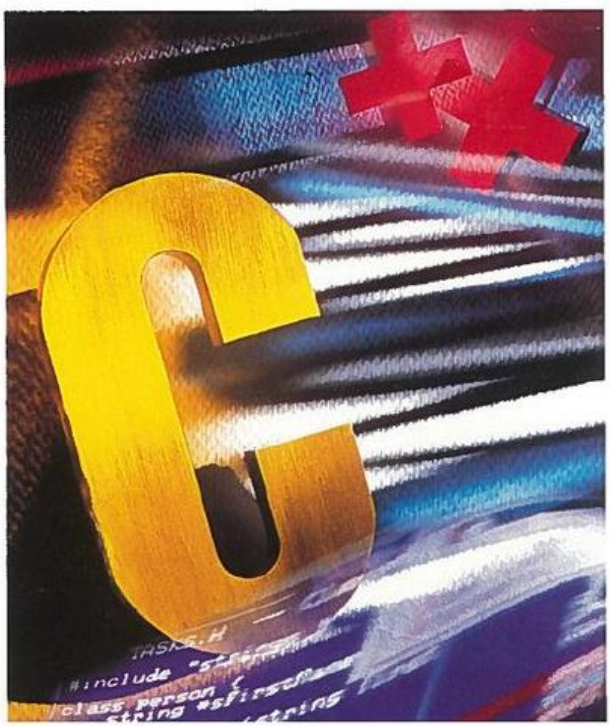
TURBO VISION® FOR C++

USER'S GUIDE



BORLAND

Turbo Vision 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. Other brand and product names are trademarks or registered trademarks of their respective holders.

R1

10 9 8 7 6 5

C O N T E N T S

T A B L E S

F I G U R E S

I N T R O D U C T I O N

If you write character-based applications that need a high-performance, flexible, and consistent interactive user interface, Turbo Vision is for you.

Turbo Vision is an application framework for DOS-based applications that provides a whole new way of looking at application development. You can use it to create a complete user interface, including windows, dialog boxes, menus, mouse support, and even a simple, fast editor, for your own application.

In this book you'll find out more about what Turbo Vision can do, how it does it, and why. Once you take the time to understand the underlying principles of Turbo Vision, you will find it a rewarding, time-saving, and productive tool: You can build sophisticated, consistent interactive applications in less time than you thought possible.

Why Turbo Vision?

With Turbo Vision and object-oriented programming, you don't have to reinvent the wheel—you can inherit ours!

After creating a number of programs with consistent user interfaces at Borland, we decided to package all that functionality into a reusable set of tools. Object-oriented programming gave us the vehicle, and Turbo Vision is the result.

Does it work? You bet! We used the Turbo Pascal version of Turbo Vision to write the integrated development environment for Turbo Pascal 6.0 in a fraction of the time it would have taken to write it from scratch. Now you can use these same tools to write your own applications.

What is Turbo Vision?

Turbo Vision saves you from endlessly recreating the basic platform on which you build your application programs.

Turbo Vision is a complete object-oriented library of classes that provide you with the basic functionality you'll need for creating windowing applications, including:

- Multiple, resizeable, overlapping windows
- Pull-down menus
- Mouse support
- Dialog boxes
- Built-in color installation
- Buttons, scroll bars, input boxes, check boxes and radio buttons
- Standard handling of keystrokes and mouse clicks
- And more!

Using Turbo Vision, all your applications can have a consistent state-of-the-art look and feel, with very little effort on your part.

What you need to know

You need to be pretty comfortable with object-oriented programming and C++ in particular in order to use Turbo Vision. Applications written in Turbo Vision make extensive use of object-oriented techniques, including inheritance and polymorphism.

What's in this book?

Because Turbo Vision is new, and because it uses some techniques that might be unfamiliar to many programmers, we have included a lot of explanatory material and a complete reference section.

- Part 1 introduces you to the basic principles behind Turbo Vision and provides a tutorial that walks you through the process of writing Turbo Vision applications.
- Part 2 gives greater detail on all the essential elements of Turbo Vision, including explanations of the members of the Turbo Vision class hierarchy and suggestions for how to write better applications.
- Part 3 is a complete reference for all the classes and other elements included in the Turbo Vision header files.

Installing Turbo Vision

Turbo Vision comes with an automatic installation program called INSTALL. Because we used file-compression techniques, you must use this program; you can't just copy the Turbo Vision files onto your hard disk. Instead, INSTALL automatically copies and uncompresses the Turbo Vision files. For reference, the README file on the installation disk includes a list of the distribution files.

We assume you are already familiar with DOS commands. For example, you'll need the DISKCOPY command to make backup copies of your distribution disks. Make a complete working copy of your distribution disks when you receive them, then store the original disks away in a safe place.

None of Borland's products use copy protection schemes. If you are not familiar with Borland's No-Nonsense License Statement, read the agreement included with your Turbo Vision package. Be sure to mail us your filled-in product registration card; this guarantees that you'll be among the first to hear about the hottest new upgrades and versions of Turbo Vision.

Using INSTALL

We recommend that you read the README file before installing.

Among other things, INSTALL detects what hardware you are using and configures Turbo Vision appropriately. It also creates directories as needed and transfers files from your distribution disks (the disks you bought) to your hard disk. Its actions are self-explanatory; the following text tells you all you need to know.

To install Turbo Vision:

1. Insert the installation disk (disk 1) into drive A (or whichever drive is appropriate for your disk size). Type the following command, then press *Enter*.

   ```
   A:INSTALL
   ```

2. Press *Enter* at the installation screen.
3. Follow the prompts.

The README and HELPME! files

Important!

When it is finished, INSTALL reminds you to read the latest about Turbo Vision in the README file, which contains important, last-minute information about Turbo Vision. There may also be a HELPME!.DOC file, which, if present, answers many common technical support questions.

The README file

To access the README file:

1. If you haven't installed Turbo Vision, insert your Turbo Vision disk into drive A. If you have installed Turbo Vision, skip to step 3 or go on to the next paragraph.
2. Type `A:` and press *Enter*.
3. Type `README` and press *Enter*. Once you are in the file, use the ↑ and ↓ keys to scroll through the file.
4. Press *Esc* to exit.

Once you've installed Turbo Vision, you can open README into an edit window in Borland C++, following these steps:

1. Start Turbo Vision by typing `BC` on the command line. Press *Enter*.
2. Press *F10*. Choose **F**ile | **O**pen. Type in `README` and press *Enter*. Borland C++ opens the README file in an edit window.
3. When you're done with the README file, you can exit Borland C++ or play with Turbo Vision.

The HELPME!.DOC file

Turbo Vision may also come with a file called HELPME!.DOC, which, if present, contains answers to problems that users commonly run into. Consult it if you find yourself having difficulties. You can use the README program to look at HELPME!.DOC. Type this at the command line:

```
README HELPME!.DOC
```

Typefaces and icons used in these books

`Monospace type` This typeface represents text as it appears onscreen or in a program. It is also used for anything you must type literally.

ALL CAPS We use all capital letters for the names of constants and files.

() Square brackets [ ] in text or DOS command lines enclose optional items that depend on your system. *Text of this sort should not be typed verbatim.*

Boldface Function, class, and structure names are shown in boldface when they appear in text (but not in program examples). This typeface is also used in text for reserved words (such as **char**, **switch**, **near**, and **cdecl**).

Italics *Italics* indicate variable names (identifiers) that appear in text. They can represent terms that you can use as is, or that you can think up new names for (your choice, usually). They are also used to emphasize certain words, such as new terms.

Keycaps This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."

How to contact Borland

Borland offers a variety of services to answer your questions about this product. Be sure to send in the registration card; registered owners are entitled to technical support and will receive information on upgrades and supplementary products.

Resources in your package

This product contains many resources to help you find solutions:

- The manual provides information on every aspect of the program. Use it as your main information source.
- Many common questions are answered in the README file (and in the HELPME! file if present); see page 4 for more on those files.

Borland resources

Borland Technical Support publishes technical information sheets on a variety of topics and is available to answer your questions on a variety of venues.

TechFax

800-822-4269 (voice)

TechFax is a 24-hour automated service that sends technical information to your FAX machine free of charge. You can use your touch-tone phone to request up to three documents per call.

Call our BBS

408-439-9096 (modem)

The Borland File Download BBS has sample files, applications, and technical information you can download with your modem. No special setup is required.

Call another BBS

Subscribers to the CompuServe, GEnie, or BIX information services can receive technical support by modem. Use the commands in the following table to contact Borland while accessing these information services.

Table 0.1
Online information services

Service	Command
CompuServe	GO BORLAND
BIX	JOIN BORLAND
GEnie	BORLAND

Address electronic messages to SYSOP or All. Don't include your serial number; messages are in public view unless sent by electronic mail. Include as much information on the question as possible; the support staff will reply to the message within one working day.

Write a letter

If you prefer, write a letter with your comments and send it to

Borland International
Technical Support Department—Turbo Vision
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95067-0001, USA

Call us

408-438-5300 (voice)

Before calling Technical Support, check your README file; it may solve the problem for you. It also contains more detailed information on reporting problems. If you decide then that you still need to call, you can contact Borland Technical Support from 6:00 a.m. to 5 p.m. Pacific standard time. Please call from a telephone near your computer, and have the program running.

Have the following information handy:

- Product name, serial number, and version number.
- The brand and model of any hardware in your system.
- Operating system and version number. (Use the DOS command `VER` to find the version number.)
- Contents of your AUTOEXEC.BAT and CONFIG.SYS files (located in the root directory (\) of your computer's boot disk).
- A daytime phone number where you can be contacted.
- If the call concerns a problem, a list of the steps required to reproduce the problem.

PART 1

Learning Turbo Vision

C H A P T E R

1

Inheriting the wheel

How much of your last application was meat, and how much was bones?

The meat of an application is the part that solves the problem the application was written to address: Calculations, database manipulations, and so on. The bones, on the other hand, are the parts that provide the basic structure upon which the meat relies for strength and shape. Menus, editable fields, error reporting, mouse handlers, and so on. If your applications are like most, you spend as much or more time writing the bones as you do the meat. And while this sort of program infrastructure can in general be applied to *any* application, out of habit most programmers just keep writing new field editors, menu managers, event handlers, and so on, with only minor differences, for each new project they begin.

You've been warned often enough to avoid reinventing the same old wheel. So here's your chance to stop reinventing the wheel—and start *inheriting* it.

The framework of a windowing application

Turbo Vision is the framework of an event-driven, windowing application. There's only a little meat as delivered—mainly a strong, flexible skeleton. You flesh the skeleton out by using the extensibility feature of C++ object-oriented programming. Turbo Vision provides you with a skeleton application class,

TApplication, from which you can derive a specialized class—call it **TMyApplication**, perhaps—to support your application. Then you add or overide members in **TMyApplication** to get your job done.

At the very highest level, that's all there is to it. The **main** function of your application program might look as simple as this:

```
int main()
{
   TMyApplication myApplication;     // create an instance...
   myApplication.run();              // and run it!
   return 0;
}
```

The underlying constructors and destructors take care of all the housekeeping chores: initializing objects and their disposal.

A new Vision of application development

Experienced C++ programmers can skip this sales pitch.

You've probably used C function libraries before, and at first Turbo Vision may sound a lot like traditional libraries. After all, libraries can be purchased to provide menus, windows, mouse event handlers, and so on. But beneath that superficial resemblance is a radical difference, one that is worth understanding to avoid running up against some high and hard conceptual walls.

The first thing to do is remind yourself that you're now in object country. In traditional structured programming, when a tool such as a menu manager doesn't quite suit your needs, you modify the tool's source code until it does. Going in and changing the tool's source code is a bold step that is difficult to reverse, unless you somehow take note of exactly what the code originally looked like. Furthermore, changing proven source code (especially source code written by somebody else) is a fine way to introduce new bugs into a proven subsystem, bugs that could propagate far beyond your area of original concern.

With Turbo Vision, you never have to modify the actual source code. You "change" Turbo Vision by extending it. The **TApplication** application skeleton remains unchanged and hidden inside TV.LIB. You add to it by deriving new classes, and change what you need to by overriding the inherited member functions with new ones for your new objects.

Also, *Turbo Vision is a hierarchy*, not just a disjoint box full of tools. If you use any of it at all, you should use *all* of it. There is a single architectural vision uniting every component of Turbo Vision: they all work together in many subtle, interlocking ways. You shouldn't try to just "pull out" mouse support and use it—the "pulling out" would be more work than writing your own mouse event handlers from scratch.

These two recommendations are the foundation of the Turbo Vision development philosophy: *Use object-oriented techniques fully*, and *embrace the entirety of Turbo Vision*. This means playing by Turbo Vision's "rules" and using its component classes as they were intended to be used. We created Turbo Vision to save you an *enormous* amount of unnecessary, repetitive work, and to provide you with a proven application framework you can trust. To get the most benefit from it, let Turbo Vision do the work.

The elements of a Turbo Vision application

Before we look at how a Turbo Vision application works, let's take a look at "what's in the box"—what tools Turbo Vision gives you to build your applications with.

Naming of parts

A Turbo Vision application is a cooperating society of *views*, *events*, and *mute objects*.

Views

Views are covered in detail in Chapter 4.

A view is any program element that is visible on the screen—and all such elements are objects. In a Turbo Vision context, if you can see it, it's a view. Fields, field captions, window borders, scroll bars, and menu bars are all views. You can combine views to form more complex elements, such as windows and dialog boxes. These "collective" views are called groups, and they operate together as though they were a single view. Groups are, in fact, specialized views that can own other views, known as subviews. Groups can even own other groups, leading to elaborate chains of views and subviews. Later in this chapter, you'll see how you can insert views into a group that represents your application.

Views are always rectangular. This includes rectangles that contain a single character, or lines which are only one character high or one character wide.

Events

Events are explained in detail in Chapter 5.

An event is some sort of occurrence to which your application must respond. Events come from the keyboard, from the mouse, or from other parts of Turbo Vision. For example, a keystroke is an event, as is a click of a mouse button. Events are queued up by Turbo Vision's application skeleton as they occur, then they are processed in order by an event handler. The **TApplication** class, which provides the framework of your application, contains an event handler. Through a mechanism that will be explained later on, events that are not serviced by **TApplication** are passed along to other views owned by the program until either a view is found to handle the event, or an "abandoned event" error occurs.

For example, an *F1* keystroke invokes the help system. Unless each view has its own entry to the help system (as might happen in a context-sensitive help system) the *F1* keystroke is handled by the main program's event handler. Ordinary alphanumeric keys or the line-editing keys, by contrast, need to be handled by the view that currently has the *focus*; that is, the view that is currently interacting with the user.

Mute objects

Mute objects are any other objects in the program that are not views. They are "mute" because they do not speak to the screen themselves. They perform calculations, communicate with peripherals, and generally do the work of the application. When a mute object needs to display some output to the screen, it must do so through the cooperation of a view. This concept is important to keeping order in a Turbo Vision application: *Only views may access the display.*

Nothing will stop your mute objects from writing to the display with **printf** or **<<** statements. However, if you write to the display "on your own," the text you write could disrupt the text that Turbo Vision writes, and the text that Turbo Vision writes (by moving or sizing windows, for example) could obliterate this "renegade" text.

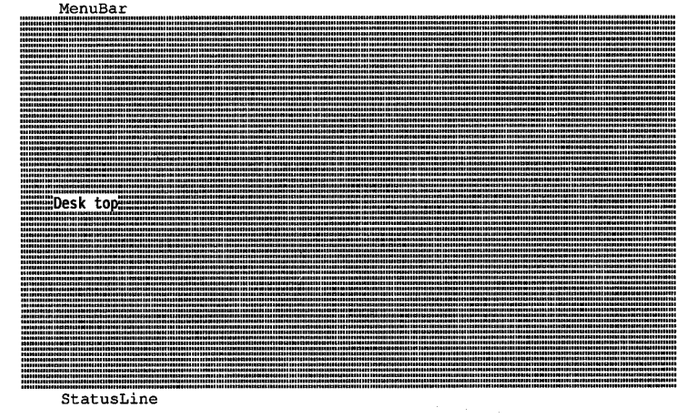
A common "look and feel"

Because Turbo Vision was designed to take a standardized, rational approach to screen design, your applications acquire a familiar look and feel. That look and feel is identical to that of Borland's language compilers, and is based on years of experience and usability testing. Having a common and well-understood look to an application is a distinct advantage to your users and to yourself: No matter how arcane your application is in terms of what it does, the way to use it will always be familiar ground, and the learning curve will be easier. Turbo Vision can provide such standard interfaces while retaining full flexibility for your own customized designs.

All these items are described in Chapter 4, "Views."

Figure 1.1 shows a collection of common objects that might appear as part of a Turbo Vision application. The desk top is the shaded background against which the rest of the application appears. Like most things in Turbo Vision, the desk top is an object. So are the menu bar at the top of the display and the status line at the bottom. Words in the menu bar represent menus, which are "pulled down" by clicking on the words with the mouse pointer or by pressing hot keys.

Figure 1.1
Turbo Vision objects onscreen



The text that appears in the status line is up to you, but typically it displays messages about the current state of your application, shows available hot keys, or prompts for commands that are currently available to the user.

When a menu is pulled down, a highlight bar slides up and down the menu's list of selections in response to movements of the mouse or cursor keys. When you press *Enter* or click the left mouse button, the item highlighted at the time of the button press is selected. Selecting a menu item transmits a command to some part of the application.

Your application typically communicates with the user through one or more windows or dialog boxes, which appear and

disappear on the desk top in response to commands from the mouse or the keyboard. Turbo Vision provides a great assortment of window mechanisms for entering and displaying information. Window interiors can be made scrollable, which enables windows to act as portals into larger data displays, such as document files. Scrolling the window across the data is done by moving a scroll bar along the bottom of the window, the right side of the window, or both. The scroll bar indicates the window's position relative to the complete background view.

Dialog boxes often contain buttons, which are highlighted words that can be selected by clicking on them (or by tabbing to the button and pressing *Spacebar*). The displayed words appear to move "downward" in response to the click (as a physical push-button would) and can be set to transmit a command to the application.

"Hello, World!" Turbo Vision style

Turbo Vision gives us a different way to say "Hello, World!"

The traditional way to demonstrate how to use any new language or user interface toolkit is to present a "Hello, World!" program written with the tools in question. This program usually consists of only enough code to display the string "Hello, World!" on the screen, and to return control to DOS.

The classic "Hello, World!" program is *not* interactive (it "talks" but it doesn't "listen"). In contrast, Turbo Vision is a tool for producing interactive programs.

The "Hello, World!" code is given in the file HELLO.CPP on your distribution disks.

The simplest Turbo Vision application is much more involved than the usual **printf** sandwiched between **{** and **}**. Compared to the classic "Hello, World!" program, Turbo Vision's HELLO.CPP does a fair number of things, including

- clearing the desk top to a halftone pattern
- displaying a menu bar and a status line at the top and bottom of the screen
- establishing a handler for keystrokes and mouse events
- building a menu object "behind the scenes" and connecting it to the menu bar
- building a dialog box, also "behind the scenes"
- connecting the dialog box to the menu

- waiting for you to take some action, through the mouse or keyboard

Nowhere in this list is there anything about displaying text to the screen. Some text has been prepared, but it's all in the background, waiting to be called up on command. That's something to keep in mind while you're learning Turbo Vision: The essence of programming with Turbo Vision is designing a custom view and teaching it what to do when it receives commands. Turbo Vision—the framework—worries about getting the proper commands to your view. You only have to worry about what to do when the keystroke, mouse click, or menu command finds its way to your view's code.

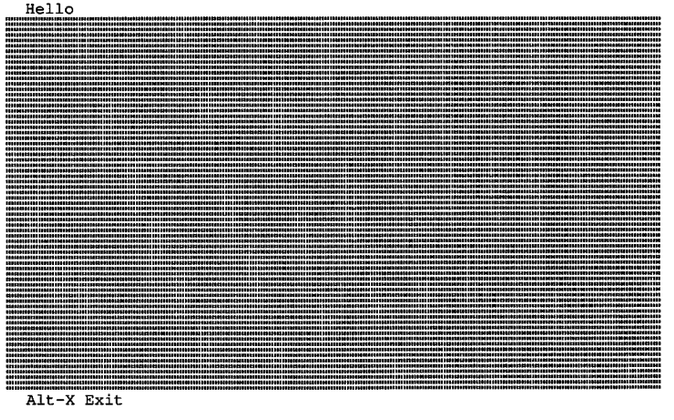
The meat of your program is the code that performs some meaningful work in response to commands entered by the user—and this "meaty" code is contained in the view objects you create.

Running HELLO.CPP

Before we dissect HELLO.CPP in detail, it would be a good idea to load the program, compile it, and follow through its execution.

When run, HELLO clears the screen and creates a desk top like that shown in Figure 1.2. No windows are open, and only one item appears in the menu bar at the top of the screen: the command **H**ello. Notice that the "**H**" in **H**ello is set off in a different color from the "ello", and that the status bar contains a message: "Alt-X Exit".

Figure 1.2
The HELLO.CPP startup screen



This is a good time to point out two general rules for programming in *any* user environment:

1. *Never put the user at a loss as to what to do next.*
2. *Always give the user a way forward and a way back.*

Before doing anything at all, the user of HELLO has two clear choices: Either select the menu item **H**ello or press *Alt-X* to leave the program entirely.

Pulling down a menu

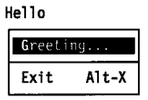
With that in mind, select **H**ello in the menu bar. There are actually three ways to do this:

- Move the mouse pointer over **H**ello and click the left button.
- Press *F10* to take the cursor to the menu bar, where **H**ello becomes highlighted. Then press *Enter* to select **H**ello.
- Press *Alt-H*, where H is the highlighted character in the menu command **H**ello.

In all three cases, a pull-down menu appears beneath the item **H**ello. This should feel familiar to you, as a Turbo C++ or Borland C++ programmer. You'll find that Turbo Vision uses all of the conventions of the Borland IDEs (integrated development environments).

The menu that appears is shown in Figure 1.3. There are only two items in the menu, separated by a single line into two separate panes. HELLO is so simple that there is only one menu item in each pane, but in fact there may be any number of items in a pane, subject to the limitations of the screen.

Figure 1.3
The HELLO.CPP menu



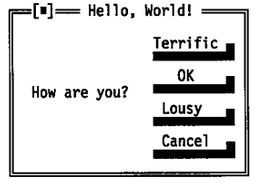
You can select a menu item either from the keyboard or with the mouse. The arrow keys move the highlight bar up and down the menu. Selecting a highlighted item from the keyboard is done by pressing *Enter* when the desired item is highlighted. More interesting is selection by mouse: You "grasp" the highlight bar by pressing the left mouse button down while the mouse pointer is on the highlight bar and holding the button down. As long as you hold the button down, you can move (drag) the bar up and down the list of menu items within the menu. You select one of the menu items by letting go of the mouse button when the highlight bar is over the menu item that you wish to select.

A dialog box

An ellipsis (...) after a menu item indicates that the item invokes a dialog box or further submenus.

However you select it, the **G**reeting item in the menu brings up a rectangular window called a dialog box, as shown in Figure 1.4. The dialog box appears in the center of the screen, but you can drag it around the screen by moving the mouse pointer to the top line of the dialog box, pressing the left mouse button, and moving the mouse while you hold the button down. As soon as you let the button up, the dialog box will stop where it is and remain there.

Figure 1.4
The Hello, World! dialog box



The dialog box has a title, "Hello, World!", and a close icon at its upper left corner. The close icon, when clicked by the mouse, closes the dialog box and makes it disappear. Inside the dialog box is a short text string: "How are you?" This is an example of static text, which is text that can be read but which contains no interactive power. In other words, static text is used to label things, but nothing happens if you click on it.

Buttons

The four rectangles on the right side of the box are the most interesting parts of the "Hello, World!" dialog box. These are called buttons, and are examples of controls. They are called controls because they resemble the controls on electronic instruments. Each button has a label, which indicates what happens when that button is pushed.

You push a button by clicking on it with the mouse, or by making the button the default (described later in this section) and then pressing *Enter*. Try pressing one of the buttons with the mouse (holding down the mouse button while the pointer is on the button) and see what happens: The body of the button moves one position to the right, and its shadow vanishes. The illusion is that of a rectangular button being pressed "downward" toward the screen. When you release the mouse button, the action specified by the button takes place.

Monochrome systems indicate the default button with "» «" characters.

Notice that the title inside the Cancel button is colored differently than the others. The difference in color indicates that the Cancel button is currently the default control within the dialog box. If you press *Enter* while Cancel is the default, you are in effect pressing the Cancel button.

The default control within a dialog box can be changed by pressing the *Tab* key. Try tabbing around in the "Hello, World!" dialog box. The distinctive default colors move from one button to the next with each press of the *Tab* key. This allows the user to press a button without using a mouse, by moving the default to the chosen button with the *Tab* key, and then pressing *Enter* or *Spacebar* to perform the actual "press of the button."

Note that mouse clicks on the menu bar have no effect while the dialog box is active. The dialog box is in control!

Getting out

Pressing any of the buttons in HELLO "puts away" the dialog box and leaves you with an empty desk top. You can pull down the **H**ello menu again, and bring up the dialog box again, any number of times. To exit the program, you can either select the E**x**it item in the **H**ello menu, or simply press the E**x**it shortcut, *Alt-X*. Note that this shortcut is presented both inside the **H**ello menu and in the status line at the bottom of the screen.

This is good practice: *Always make it easy for the user to exit the program*. Frustrated users who can't find the door are quite likely to reboot the system, preventing your application from closing files or otherwise cleaning house before shutting down.

Inside HELLO.CPP

That's what HELLO does if you run it. Now, how does it make all this happen? Much—in fact, most—of the code comprising HELLO is inherited from standard classes provided in Turbo Vision. So much is inherited that when the program runs, *how* it works may first seem a bit of a mystery. Tracing execution with the integrated debugger will not show you the whole picture, since Turbo Vision is provided as compiled library. Fortunately, if you take the time to understand what is going on, the exact *how* won't be necesssary.

To understand a Turbo Vision application, start by reminding yourself that *a Turbo Vision application is a society of objects working together*. Find the major objects and understand how they work together. Then see how the lesser objects support the major objects.

Be sure you read and understand the class declarations in the supplied *.H files before you study the implementation detail. It's important that you understand what an object contains and how it relates to the other objects in the system.

The application class

The cornerstone of any application is the **TApplication** class, although you never directly instantiate an object of this type. As a grandchild of **TGroup**, **TApplication** is a group that knows how to own and control dynamic chains of views, such as your application's menus, windows and other subviews. **TApplication** does not, in itself, implement your application. You use **TApplication** as a base class. From this you derive a class that contains the meat of your program. Only one instance of this application class is needed for a given program. In HELLO, that derived class is **THelloApp**, declared as follows:

```
class THelloApp : public TApplication
{
```

```
public:
   THelloApp();      // constructor
   virtual void handleEvent( TEvent& event );
   static TMenuBar *initMenuBar(TRect r);
   static TStatusLine *initStatusLine(TRect r);
private:
    void greetingBox();
};
```

In more complex programs, the class declarations would be placed in a separate .H file and the definitions in .CPP files.

Of course, **THelloApp** contains much more than just its constructor and the four member functions listed above. A derived class inherits everything from its immediate base class (and that base class in turn inherits from its base class, and so on). When crafting **THelloApp**, you define how the new class *differs* from its ancestor, **TApplication**. Everything that you do not redefine is inherited unchanged from **TApplication**.

We'll leave aside discussion of the constructor for the moment and review the four member functions defined in **THelloApp**. These provide the "big picture" of your entire application:

- How the application behaves is dictated by what events it responds to, and how it responds to them. You must define a **handleEvent** method to fulfill this all-important requirement. A virtual **handleEvent** member function is inherited from **TApplication** (via its base class, **TProgram**) to deal with generic events that occur within any application, but you must always override it to handle events specific to your own application.
- The **initMenuBar** static member function returns a pointer to a **TMenuBar** object. This sets up the menus behind the menu bar for your application. **TApplication** has a menu bar but no menus; if you want menus (and it would be a poor application indeed without them!) you *must* define a member function such as **initMenuBar** to provide them. You'll see later that the **THelloApp** constructor calls **initMenuBar** and inserts the resulting **TMenuBar** object into the application group.
- Similarly, the **initStatusLine** static member function returns a pointer to a **TStatusLine** object to provide the status line text at the bottom of the screen. This text typically displays messages about the current state of the application, shows the available hot keys, or notifies the user of some action to be taken. As with **initMenuBar**, the **THelloApp** constructor calls **initStatusLine** and inserts the resulting **TStatusLine** object into your application group.

- The **greetingBox** private member function brings up the dialog box in response to the menu item **G**reeting. **greetingBox** is called only from within the **handleEvent** member function, in response to the event triggered by the selection of the **G**reeting menu item by mouse or keyboard action. In more advanced applications, you would have separate member functions to respond to each of the menu items defined in the initial menu.

In short, **THelloApp**'s member functions provide what all main-program objects must provide: a constructor to set the application up, an "engine" (the event handler) to recognize relevant events, and member functions to respond to such events. These three things are, by and large, what you must add to classes derived from **TApplication**.

The dialog box object

The only other major object used in HELLO is a dialog box object. Because the dialog box doesn't have to do anything special, HELLO uses an instance of the **TDialog** class. There is usually no need to derive a special class from **TDialog**, although there are occasions when this can be useful.

TDialog itself contains no interactive elements. It is nothing more than a frame (albeit a clever frame); you provide whatever fields or controls are needed to interact with the user.

THelloApp::greetingBox builds on **TDialog** by inserting four buttons which are also Turbo Vision views. (Remember that *all* objects that display *anything* to the screen *must* be Turbo Vision views!) This is typical when using dialog boxes. Usually you just insert the controls you want to have in the dialog box. Everything else that a dialog box must have (including its event handler) is built into **TDialog**.

Flow of execution and debugging

Because Turbo Vision applications are event-driven, the code is structured somewhat differently than conventional programs. Specifically, event-driven programs separate the control structures that read and evaluate user input (and other events) from the functions that act on that input.

Conventional programs typically contain many blocks of code, each of which involves getting some input, deciding which code gets that input, calling the appropriate routine(s) to process the

input, then doing the same thing again. In addition, the code that finishes processing the input must then know where to give control for the next round of input.

Event-driven programs, on the other hand, have a central event-dispatching mechanism, so the bulk of your program does not have to worry about fetching input and deciding what to do with it. Your routines simply wait for the central dispatcher to hand them input to process. This has important implications for debugging your programs: You will probably want to rethink your debugging strategies, setting breakpoints in event-handling routines to check the dispatching of messages, and setting breakpoints in your event-responding code to check that it functions properly.

For more hints and tips on debugging Turbo Vision applications, see Chapter 10, "Hints and tips."

HELLO's main program

At the very highest level, the **main** program in all Turbo Vision applications look pretty much like HELLO:

```
int main()
{
   THelloApp helloWorld;
   helloWorld.run();
   return 0;
}
```

Let's examine the body of **main** in more detail.

Application instantiation

The first statement declares an instance of **THelloApp** called **helloWorld**. This invokes the constructor **THelloApp::THelloApp** to create and initialize the object, **helloWorld**. Ignoring the finer details, the **THelloApp** constructor triggers calls to base constructors down the Turbo Vision class hierarchy from **TObject** (the "root" class of Turbo Vision), to **TView**, to **TGroup**, to **TProgram**, to **TApplication**, and finally to **THelloApp**. This spate of off-stage activity sets up a whole slew of default conditions and mechanisms that provide standard interfaces for most applications. (Advanced programmers who want to produce nonstandard interfaces will need to study and override the appropriate class constructors detailed in Chapter 13.) The net result is that **helloWorld**, the main program object, starts life with a cleared, full-screen, halftone desk top view. The **TProgram** constructor calls **initMenuBar** and **initStatusLine** to set up and

insert the particular HELLO menu and status line that you saw earlier. The essential steps are:

```
// simplified extract from TProgram and TProgInit constructors

menuBar = initMenuBar();    // get menuBar pointer
if ( menuBar != 0 )
   insert( menuBar );       // insert it in app group's subview list

statusLine = initStatusLine();     // get statusLine pointer
if ( statusLine != 0 )
   insert( statusLine );    // insert it in app group's subview list
```

As this stage, it is sufficient to get a general feel for these operations. The source code for **THelloApp::initMenuBar** and **THelloApp::initStatusLine** (in HELLO.CPP) reveals the basic strategy. In Chapter 2, we'll explain in more detail how to create a menu bar and its associated menus and commands, together with a status line and its text and commands.

You may find it interesting to use a debugger to step over HELLO.EXE and inspect the display. After the constructor calls, the desk top, menu bar, and status line will all be laid out and complete, ready for **helloWorld.run**.

The run member function

Nearly all of the mystery in a Turbo Vision application is in the main program's **run** member function. The mystery starts when you look into **THelloApp** to find the definition of **run**. It's not there—because **run** is inherited intact from **THelloApp**'s base class, **TApplication**, via its base class, **TProgram**. **TProgram::run** calls **execute**, which is where your application will probably spend the bulk of its time. **execute** is a double nested **do..while** loop, somewhat simplified as follows:

```
ushort TGroup::execute()
{
    do {
       endState = 0;
       do {
          TEvent event;
          getEvent( event );
          handleEvent( event );
          } while ( endState == 0 );
    } while( !valid(endState) );
    return endState;
}
```

For more details on how events are handled, refer to Chapter 5.

Without spelling out all the detail, you can see that a Turbo Vision application loops through two tasks: Getting an event with **getEvent** (where an event is essentially "something to do"), and servicing that event with **handleEvent**. Eventually, one of the events resolves to some sort of quit command, and the loop terminates.

The application destructor

There is no explicit destructor defined in HELLO.CPP. When the program terminates, the object **helloApp** and all its associated views (menu bar, status line, and desk top) are disposed of by automatic calls to the destructors of the base classes, in the reverse order in which the constructors were called. Finally, Turbo Vision's error handler and drivers are shut down. In general, no special code is needed to dispose of Turbo Vision objects.

Summary

In this chapter you've had a first (intriguing, we hope) taste of what Turbo Vision is all about. You have seen objects interacting in an event-driven framework and seen a little of the tools that Turbo Vision provides.

At this point you may feel confident enough to try modifying the HELLO.CPP program to do some other things. Feel free to do so. One of the nicest features of Turbo Vision is the freedom it gives you to change your programs with very little effort.

The next chapter will take you through the steps of building a Turbo Vision program of your own from the skeleton we provide.

C H A P T E R

2

Writing Turbo Vision applications

Now that you've seen what a Turbo Vision application looks like, inside and out, you're probably itching to write one yourself. In this chapter, you'll do just that, starting with a simple framework and adding small pieces of code at each step so you can see what each of them does.

You probably have a lot of questions at this point. How exactly do views work? How does a group control its subviews? What can I do with them? How can I customize them for my applications? If Turbo Vision were a traditional run-time library, most likely you would dig into the source code to get the answers.

But Turbo Vision is already a working application. The best way to answer your questions about Turbo Vision is for you to actually try out views. As you'll see, you can initialize them with a minimum of code.

Your first Turbo Vision application

A Turbo Vision application always begins by instantiating a class derived from **TApplication**. In the following example, you will derive a class from **TApplication** called **TMyApp**. In it, you'll begin to override **TApplication** member functions and/or add new members. Next, you will declare an object of type **TMyApp**, called **myApp**. Keep in mind that **myApp** is a special kind of view known as a group, tracing its ancestry up the hierarchy via **TApplication**, **TProgram**, **TGroup**, and **TView**. From each of these classes, **myApp** has inherited a number of properties and capabilities which can be exploited without you having to write explicit code. Turbo Vision programming requires a growing knowledge of what each class contributes as "default" behavior. You'll see in this chapter that the group properties of **myApp** are particularly important. Your application group will eventually own a series a views (including other groups that own subviews) that must respond to "random" user events.

There is only one ***TApplication*** *object in a program.*

In the rest of this chapter, we will often refer to **myApp**. By that we mean your application, an instance of a class descended from **TApplication**. When you write your own Turbo Vision applications, you will probably call them something else, something indicative of the function of each application. We use **myApp** as shorthand for "the instance of the class you derived from **TApplication**."

Several stages of the example code are on your distribution disks. The file names are indicated next to the code examples.

Beginning with the following code example, you will build an example program. Rather than giving the entire program listing each time, we've only included the added or changed parts in the text. If you follow along and make all the indicated changes, you should get a good feel for what it takes to add each increment of functionality. We also strongly recommend that you try modifying the examples. You are also encouraged to study the various *.H header files on the distribution diskettes. These provide the class declarations, member function prototypes, data member definitions, and various constants and macros. Chapter 12 offers a useful header file cross-reference.

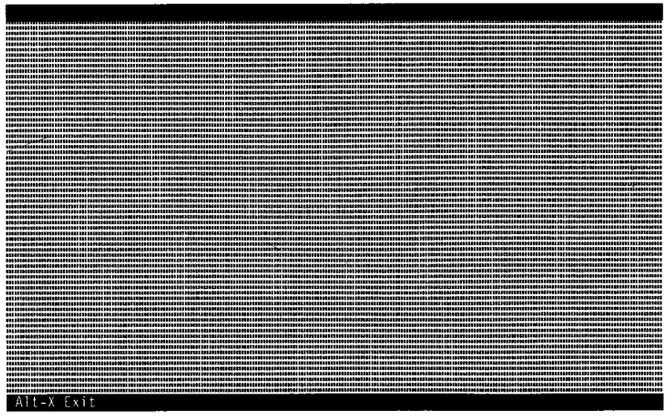
The main block of TVGUID01 (and of every Turbo Vision application) looks like this:

This program is TVGUID01.CPP, which is included with the demo programs on your distribution disks.

```
// TVGUID01.CPP

#define Uses_TApplication
#include <tv.h>

class TMyApp : public TApplication
{

public:
   TMyApp();

};

TMyApp::TMyApp() :
   TProgInit( &TMyApp::initStatusLine,
         &TMyApp::initMenuBar,
         &TMyApp::initDeskTop
         )
{
}

int main()
{
   TMyApp myApp;
   myApp.run();
   return 0;
}
```

Note that you haven't yet added any new functionality to **TMyApp**. Normally, it makes little sense to declare a derived class without adding new members or overriding inherited members. You could simply declare the object **myApp** as an instance of **TApplication**. Since you'll be adding to **myApp** later on as your Turbo Vision application evolves, you can look on **TMyApp** as a springboard for future action. For now, **myApp** will behave as a "plain vanilla" **TApplication**. The default behavior of a **TApplication** object produces a screen like that in Figure 2.1.

Figure 2.1
Default **TApplication** screen



This working program does only one thing: It responds to *Alt-X* to terminate the program. To get it to do more, you need to add to the default behavior by adding text and commands to the status line or the menu bar. In the next section, you'll do both.

The desk top, menu bar, and status line

Classes used:
TView
TGroup
TMenuBar
TMenuBox
TStatusLine
TProgInit
TProgram
TApplication
TDeskTop.

In TVGUID01.CPP, the body of the **TMyApp** constructor is empty. The **TMyApp** constructor invokes the base constructor of **TProgInit**. **TProgInit** is a virtual base for **TProgram**. The **TProgram** constructor calls the **TGroup** constructor:

```
TProgram::TProgram() : TGroup(/* args for view bounds */)

// the TGroup constructor creates a full-screen view
{
   ...
   if( createStatusLine != 0 &&
       (statusLine = createStatusLine( getExtent() ) != 0 )
       insert( statusLine );
   ...
// createStatusLine actually calls an initStatusLine defined either
// in TProgram or, if overridden, in the most-derived of its
//  children. See TProgInit constructor, Chapter 13.

// statusLine is a static data member pointing to the newly created
// status line object. insert() inserts the status line into your
// application group and draws the status line.

// getExtent() returns the rectangular bounds of the calling object
   ...
// plus similar code for createMenuBar, createDeskTop
   ...
}
```

The three *createXXX* function pointers are set to the addresses of their corresponding **initXXX** functions. This will be explained in more detail in TVGUID02 and under the **TMyApp** and **TProgInit** constructor entries in Chapter 13.

TVGUID01.CPP's bare-bones desk top, menu bar, and status line are created and inserted into the **myApp** group when the **TProgram** constructor calls the three default member functions **initDeskTop**, **initMenuBar**, and **initStatusLine**. These three functions are inherited, without change, by **TApplication** from its immediate base, **TProgram**. And, of course, **TMyApp** also inherits them unchanged (so far) from **TApplication**. As defined in **TProgram**, the three **init** functions create the minimal views shown in the Figure 2.1. In the unlikely event that you are happy with this bleak interface, you would just let the inherited **init**'s do their job. In fact, the default desk top is quite adequate for most purposes, so there is usually no need to override **initDeskTop**. The menu bar and status line, however, both need fleshing out for each application, so you always have to override **initMenuBar** and **initStatusLine**. You'll see how in this chapter.

The use of TV.H is detailed in Chapter 12.

The `#define Uses_TApplication` followed by the `#include <tv.h>` code pulls in APP.H, the include file that declares the classes **TProgram**, **TProgInit**, and **TApplication**. As the examples grow, you simply add a `#define Uses_TClassname` statement for each standard class introduced. TV.H ensures that the correct .H files are included without duplication, not only for ***TClassname***, but for all its base classes.

TProgram uses **initDeskTop**, **initMenuBar**, and **initStatusLine** to set **TProgram**'s static data members *deskTop*, *menuBar*, and *statusLine* to point to their respective objects. This implies that at any given moment there can be only one desk top object, one menu bar object, and one status line object for each application. Let's look at each of these in turn.

The desk top

initDeskTop creates a **TDeskTop** object and returns a pointer to it, as shown below. (Don't worry about the detail—the meaning of **TRect** will emerge as we proceed.)

```
TDeskTop *TProgram::initDeskTop(TRect r)
{
    r.a.y++;          // adjust top-left y coordinate
    r.b.y--;          // adjust bottom-right y coordinate
                      // gives full screen less menu bar and
                      // status line
    return new TDeskTop( r ); // create default desk top
}
```

The desk top pointer, *deskTop*, is a static data member of **TProgram** since there can be only one desk top per application. You saw earlier that the **TProgram** constructor inserts the desk top into the application group. The group **myApp** now *owns* the desk top object. But the desk top is also a group. Although it is owned by **myApp**, the desk top can own views in its own right. In fact, the desk top, as a group, plays a key role in your application by controlling all the views that appear on the desk top. For example, when the user clicks on a menu selection, **myApp** instantiates the appropriate subview and inserts it in the desk top group. Thereafter, the desk top manages the subview and any subsubviews subsequently generated. In this context, managing covers a host activities, such as event handling and drawing, hiding, and resizing views in response to such events.

The status line

The **TProgram** constructor calls **TProgInit::createStatusLine** to create a **TStatusLine** object and returns a pointer to it, *statusLine*. The actual status line created is determined by **TProgram::initStatusLine**, a program that you override to meet your application's particular status line requirements. You'll see how in a moment. You simply pass the address of your **TMyApp::initStatusLine** to the **TProgInit** constructor, and leave the rest to Turbo Vision.

As with *desk top*, *statusLine* is a static data member of **TProgram**; only one instance exists for all objects of its class. A prime function of the status line is to define and display hot key definitions. (Hot keys are keystrokes, including shift, control, Alt, and ^KF keys, that act like menu or status line items.)

The status line is displayed starting at the left edge of the screen. Any part of the bottom screen line not needed for status line items is free for other views. **statusLine* binds hot keys to commands, and the items themselves can also be clicked on with the left mouse button. To give you the flavor of **initStatusLine**, here is the default version defined in **TProgram**, giving the minimal "Alt-X" status line as shown in Figure 2.1. In addition, *F10, Alt-F3, F5* and *Ctrl-F5* are assigned to the standard IDE commands *cmMenu*, *cmClose*, *cmZoom*, and *cmResize*, but no text is displayed other than "Alt-X Exit". Only the *cmQuit* command is active (enabled) at this stage.

```
TStatusLine *TProgram::initStatusLine(TRect r) {
    r.a.y = r.b.y - 1;
    return new TStatusLine( r,
        *new TStatusDef( 0, 0xFFFF ) +
        *new TStatusItem( "~Alt-X~ Exit", kbAltX, cmQuit ) +
        *new TStatusItem( 0, kbF10, cmMenu ) +
        *new TStatusItem( 0, kbAltF3, cmClose ) +
        *new TStatusItem( 0, kbF5, cmZoom ) +
        *new TStatusItem( 0, kbCtrlF5, cmResize )
        );
}
```

*Some of the **#define Uses_TClassname** are redundant. But they do no harm and it is better to be safe than sorry.*

In the next exercise, you will get acquainted with the **initStatusLine** syntax by adding a legend to the status line. TVGUID02.CPP creates a modified status line by overriding **TApplication.initStatusLine**. Note the additional `#define Uses_TClassname` lines at the start of TVGUID02, one for each new class used. `Uses_TKeys`, for example, ensures that the various keyboard mnemonics, such as *kbF5* and *kbAltF3*, are available.

Next, add the **initStatusLine** declaration to **TMyApp**, and add the following definition:

This is part of TVGUID02.CPP.

*Don't forget to add `static TStatusLine *initStatusLine(TRect r);` to the declaration body of **TMyApp**.*

```
static TStatusLine *TMyApp::initStatusLine(TRect r)
{
   r.a.y = r.b.y - 1;     // move top to one line above bottom
   return new TStatusLine( r,
      *new TStatusDef( 0, 0xFFFF ) +
      // set range of help contexts
      *new TStatusItem( "~Alt-X~ Exit", kbAltX, cmQuit ) +
          // define an item
      *new TStatusItem( "~Alt-F3~ Close", kbAltF3, cmClose )
          // and another one
    );
}
```

Turbo Vision commands are constants. Their identifiers start with "cm."

The initialization is a sequence of calls to the **TStatusDef** and **TStatusItem** constructors (described in detail in Chapter 13). The overloaded operator **+** is used to create a linked list of **TStatusDef** and **TStatusItem** objects and to pass the list to the *items* **TStatusLine** data member:

```
TStatusDef& operator + (TStatusDef& s1, TStatusItem& s2);
```

The **TStatusItem** constructor prototype will help you understand the syntax:

```
TStatusItem::TStatusItem(const char *aText, ushort key, ushort cmd,
                         TStatusItem *aNext = 0);
```

As you can see, each status item provides a text string (set to null if no text is needed), a key code, a command code, and a pointer to the next status item (a null pointer means no more items).

The **TStatusDef** object in TVGUID02 defines a status line to be displayed for a range of help contexts from 0 through 0xFFFF. It also binds the standard Turbo Vision command *cmQuit* to the *Alt-X* keystroke, and the standard command *cmClose* to the *Alt-F3* key.

The last status item above has the text argument "~Alt-F3~ Close." The part of the string enclosed by tildes (~) will be highlighted on the screen. The user will be able to click with the left mouse button anywhere within the string to activate the command.

When you run TVGUID02, you'll notice that the *Alt-F3* status item is not highlighted, and clicking on it has no effect. This is because the *cmClose* command is disabled by default, and items that generate disabled commands are also disabled. Once you open a window, *cmClose* and the status item will be activated.

You have already defined a **TMyApp** constructor so that your own **initStatusLine** gets called rather than the default version defined in **TProgram**:

```
TMyApp::TMyApp() :
   TProgInit( &TMyApp::initStatusLine, &TMyApp::initMenuBar,
              &TMyApp::initDeskTop )
{
}

/*
Pass the addresses of your three initXXXX functions to the TProgInit
constructor. This constructor initializes three creatXXXX function
pointers (createStatusLine, etc) and uses them to intialize the
status line, menu bar, and desk top for your application.
*/
```

The technical reasons for the **TProgInit** class will be covered in Chapter 13. Briefly, **TProgram** has two base classes: **TGroup** and **TProgInit**. **TProgInit** is a public virtual base class holding pointers to the **createStatusLine**, **createDeskTop**, and **createMenuBar** functions. Virtual base constructors have their own special rules: they are called by the most-derived class and before the constructors for any derived classes. The **createStatusLine** function pointer is therefore set to `&TMyApp::initStatusLine`. Since **TMyApp** doesn't define **initMenuBar** or **initDeskTop**, *&TMyApp::initMenuBar* and *&TMyApp::initDeskTop* refer to the inherited versions of those functions.

Your status line work is over once you've initialized *statusLine*. Since you are using only predefined commands (*cmQuit* and *cmClose*), *statusLine* can handle the user's input without your further attention. (Note that since *statusLine* is a static member, references to it by classes *not* derived from **TProgram** must use the qualified form: *TProgram::statusLine*.)

Creating new commands

Note that *cmQuit* and *cmClose*, the commands you bound to the status line items, are standard Turbo Vision commands, so you don't have to define them. In order to use customized commands, you simply declare your commands as constant values. For example, you can define your own command for opening a new window:

Turbo Vision reserves some constants for its own commands. *See page 132 in Chapter 5.*

```
const cmNewWin = 199;
```

Next you can bind that command to a hot key and a status line item:

```
return statusLine = new TStatusLine ( r,
*new TStatusDef(0, 0xFFFF ) +
     *new TStatusItem( "~Alt-X~ Exit", kbAltX, cmQuit ) +
     *new TStatusItem( "~F4~ New", kbF4, cmNewWin ) +
     *new TStatusItem( "~Alt-F3~ Close", kbAltF3, cmClose)
     );
```

How to implement a function associated with *cmNewWin* will be covered when we discuss event handling later in the chapter.

The status line's initialization syntax is a good introduction to menu initialization, which is similar but somewhat more complex.

The menu bar

The default **initMenuBar** called by the **TProgram** constructor sets up a **TMenuBar** object and sets the static data member *menuBar* as follows:

```
TMenuBar *TProgram::initMenuBar( TRect r )
{
    r.b.y = r.a.y + 1;    // set bottom y coordinate of view one unit
                          // below top
    return new TMenuBar( r, 0 );
}
```

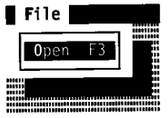
As you saw with TVGUID01, this default gives you an empty menu bar with no associated menu items. The first argument gives the view size (a rectangular strip). The 0 pointer in the second argument of `new TMenuBar` indicates that there are no menu items or legends. You must override **initMenuBar** to provide your menu hierarchy. *menuBar* is initialized with nested calls to the constructors for **TMenuItem** and **TSubMenu** using the overloaded operator **+** as seen with status lines. And, as with status lines, you must add the declaration

```
static TMenuBar *initMenuBar( TRect r);
```

to the **TMyApp** class.

A function **newLine** provides horizontal lines between menu items. Once you've initialized a menu, your work is finished. The menu bar knows how to handle the user's input without your help.

Initialize a simple menu bar, one menu containing one selection, like this:




```
const cmFileOpen = 200; // define a new command

TMenuBar *TMyApp::initMenuBar(TRect r)
{
   r.b.y = r.a.y + 1;    // set bottom 1 line below top
   return new TMenuBar( r,
   *new TSubMenu("~F~ile", kbAltF ) +
   *new TMenuItem( "~O~pen", cmFileOpen, kbF3, hcNoContext, "F3" )
   );
}
```

The single menu produced by this code is called "File," and the single menu selection is called "Open." The tildes (~) make *F* the shortcut letter in "File" and *O* the shortcut letter in "Open," while the *F3* key is bound as a hot key for "Open."

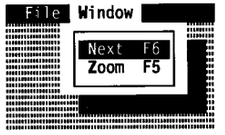
All Turbo Vision views can have a help context number associated with them. The number makes it easy for you to implement context-sensitive help throughout your application. By default, views have a context of *hcNoContext*, which is a special context

that doesn't change the current context. When you're ready to add help contexts to the menu bar, you can substitute your own values for *hcNoContext* in the **initMenuBar** code.

To add a second item to the "File" menu, you simply nest another `*new TMenuItem` call, like this:


```
return new TMenuBar( r,
  *new TSubMenu("~F~ile", kbAltF ) +
  *new TMenuItem( "~O~pen", cmFileOpen, kbF3, hcNoContext, "F3" )+
  *new TMenuItem( "~N~ew",  cmNewWin,   kbF4, hcNoContext, "F4" )
  );
```

To add a second menu, nest another `*new TSubMenu` call, like this:




```
return new TMenuBar( r,
   *new TSubMenu( "~F~ile", kbAltF )+
   *new TMenuItem( "~O~pen", cmFileOpen, kbF3, hcNoContext, "F3" )+
   *new TMenuItem( "~N~ew",  cmNewWin,   kbF4, hcNoContext, "F4" )+
   *new TSubMenu( "~W~indow", kbAltW )+
   *new TMenuItem( "~N~ext", cmNext,     kbF6, hcNoContext, "F6" )+
   *new TMenuItem( "~Z~oom", cmZoom,     kbF5, hcNoContext, "F5" )
   );
```

This code binds two more standard Turbo Vision commands, *cmNext* and *cmZoom*, to menu items and hot keys.

To add a horizontal line between menu selections, insert a call to **newLine** between the `*new TMenuItem` calls, like this (TVGUID03.CPP):


```
return new TMenuBar( r,
   *new TSubMenu( "~F~ile", kbAltF )+
   *new TMenuItem( "~O~pen", cmFileOpen, kbF3, hcNoContext, "F3" )+
   *new TMenuItem( "~N~ew",  cmNewWin,   kbF4, hcNoContext, "F4" )+
   newLine()+
   *new TMenuItem( "E~x~it", cmQuit,  kbAltX, hcNoContext, "Alt-X" )+
   );
```

You may notice that the version of TVGUID03.CPP supplied on your disk also adds a status key to the status line, binding the *F10* key to the *cmMenu* command. *cmMenu* is a standard Turbo Vision command that helps non-mouse users make use of the menu bar. In this case, the *F10* keystroke causes the menu bar to be activated, allowing menus and menu items to be selected using cursor keys. No change to **handleEvent** is needed.

You may also notice that the *F10* status item has an empty string as its text, so nothing appears on the screen for it. Although it might be nice to alert users that *F10* will activate the menus, it is

rather pointless to have an item to click on that performs that action. Clicking directly on the menu bar makes much more sense.

A note on structure

At this point, a number of commands are available, but most of them are disabled, and the *cmNewWin* and *cmFileOpen* commands don't yet perform any actions.

If your initial reaction is one of disappointment, it shouldn't be—you've accomplished a lot! In fact, what you've just discovered is one of the big advantages of event-driven programming: Separating the function of *getting* your input from the function of *responding* to that input.

With traditional programming techniques, you would need to go back into the code you've just written and start adding code to open windows and such. But you don't have to do that: You've got a solid engine that knows how to generate commands. All you need to do is write a few routines that respond to those commands. And that's just what you'll do in the next section.

The Turbo Vision application framework takes you one step beyond traditional modular programming. Not only do you break your code up into functional, reusable blocks, but those blocks can be smaller, more independent, and more interchangeable.

Your program now has several different ways to generate a command to open a window (*cmNewWin*): a status line item, a menu item, and a hot key. In a moment, you'll see how easy it is to tell your application to open a window when that command shows up. The most important thing is that the application doesn't care *how* the command was generated, and neither will the window. All that functionality is independent.

If, later on, you decide you want to change the binding of the command—move the menu selection, remap the hot keys, whatever—you don't have to think about how it will affect your other code. That's what event-driven programming buys you: It separates your user interface design from your program workings, and, as you'll see, it also allows different parts of your program to function just as independently.

Doing windows

Classes used:
TRect
TView
TWindow
TGroup
TScroller
TScrollBar.

A Turbo Vision window is an object into which is built the ability to respond to much of the user's input without you having to write a line of code. A Turbo Vision window already knows how to open, resize, drag, move, and close itself in response to outside events. But you don't write directly on a Turbo Vision window. A Turbo Vision window is a group that owns and manages other objects, but it does not draw them on the screen. The window manages the views, and your application's unique functionality is in the views that the window owns and manages. The views you create retain great flexibility about where and how they will appear.

So how do you combine the standard window tools with the things you want to put in the window? You start with a standard window, then add the features you want. As you go through the next few examples, you'll see how easy it is to flesh out the skeleton Turbo Vision provides.

The following code initializes a window and attaches it to the desk top. Remember to add the new member functions to the declaration of your **TMyApp** class. Note that again you are defining a new class (**TDemoWindow**) without adding any members to those inherited from **TWindow**, its base class. As before, you're doing that just to provide a simple platform you can easily build on. You'll add new member functions as you go.

This is from TVGUID04.CPP.

```
static short winNumber = 0;          // initialize window number

void TMyApp::handleEvent(TEvent& event)
{
  TApplication::handleEvent(event); // act like base!
  if ( event.what == evCommand )
  {
     switch( event.message.command )
     {
        case cmMyNewWin:       // but respond to additional commands
           myNewWindow();      // define action for cmMyNewWin
           break;
        default:
           return;
     }
     clearEvent( event );       // clear event after handling
  }
}
```

```
class TDemoWindow : public TWindow   // define a new window class
{

public:

   TDemoWindow( const TRect& r, const char *aTitle, short aNumber );
// declare a constructor

};

TDemoWindow::TDemoWindow( const TRect& r, const char *aTitle, short
                                                    aNumber):
              TWindow( r, aTitle, aNumber),
              TWindowInit( &TDemoWindow::initFrame )

{
}

void TMyApp::myNewWindow()
{
   TRect r( 0, 0, 26, 7 );           // set initial size and position
   r.move( random(53), random(16) ); // randomly move around screen
   TDemoWindow *window = new TDemoWindow ( r, "Demo Window",
                                                    ++winNumber);
   deskTop->insert(window); // put window into desk top and draw it
}
```

To use this window in your program, bind the command *cmNewWin* to a menu option or status line hot key, as you did earlier. When the user invokes *cmNewWin*, Turbo Vision dispatches the command to **TMyApp::handleEvent**, which responds by calling **TMyApp::myNewWindow**.

Window construction

The **TWindow** constructor needs three arguments for it to initialize itself: its size and position on the screen, a title, and a window number.

*The **TRect** class is described in detail in Chapter 4, "Views."*

The first argument, determining the window's size and position, is a **TRect**, Turbo Vision's rectangle class. **TRect** is a very simple class. Its constructors give it a size and position, based on its top-left corner and its bottom-right corner (expressed as either four integer coordinates or two **TPoint** objects). There are several member functions and overloaded operators for manipulating and comparing **TRect** objects. Consult Chapter 12, "Header file cross-reference," for complete descriptions.

In TVGUID04, the rectangle *r* is created at the origin of the desk top, then moved a random distance inside. Programs do not usually engage in random window movement, but we need a simple ploy to open lots of windows in different positions.

The second **TWindow** constructor argument is the string you want to display as the window's title.

The last argument is provides the window's *number* data member. If *number* is between 1 and 9, it will be displayed on the window frame, and the user can select a numbered window by pressing *Alt-1* through *Alt-9*.

If you don't need to assign a number to a window, just pass it the predefined constant *wnNoNumber*.

The **TDemoWindow** constructor calls both the **TWindow** and **TWindowInit** constructors. The latter takes a *&initFrame* argument, as you saw with **TProgInit** and **initStatusLine**. We haven't overridden **TWindow::initFrame** so we get a standard frame. This is more than adequate for most applications, although you can override **initFrame** for fancy effects.

The insert member function

Inserting a window into the desk top automatically makes the window appear. The **insert** member function gives a view control over another view. When you execute the instruction

```
deskTop->insert(window);
```

you are inserting *window* into the desk top. You may insert any number of views into a *group* class like the desk top. The group into which you insert a view is called the *owner* view, and the inserted view is called a *subview*. Note that a subview may itself be a group, and may have its own subviews. For instance, when you insert a window into the desk top, the window is a subview, but the window can itself own a frame, scroll bars, or other subviews.

All these relationships among views are explained in Chapter 4.

This process of establishing links between view classes creates a *view tree*, so named because the multiple linkages of views and subviews branch out from the *root* view, the application, much as limbs branch out from the trunk of a tree.

Closing a window

Clicking on a window's close icon generates the same *cmClose* command you bound to the *Alt-F3* keystroke and a status line item. By default, opening a window (with *F4* or the **File | Open** menu choice) automatically enables the *cmClose* command and the views that generate it (as well as other window-related commands like *cmZoom* and *cmNext*).

You don't have to write any new code to close the window. When the user clicks on the window's close icon or presses *Alt-F3* or clicks on the status item, Turbo Vision does the rest. By default, a window responds to the *cmClose* command by calling its destructor, which destroys the object and its title string. The window's destructor also calls the destructors of all its subviews. If you've allocated any additional memory yourself in the window's constructor, you need to make sure that you deallocate it by suitably overriding the window's destructor.

Window behavior

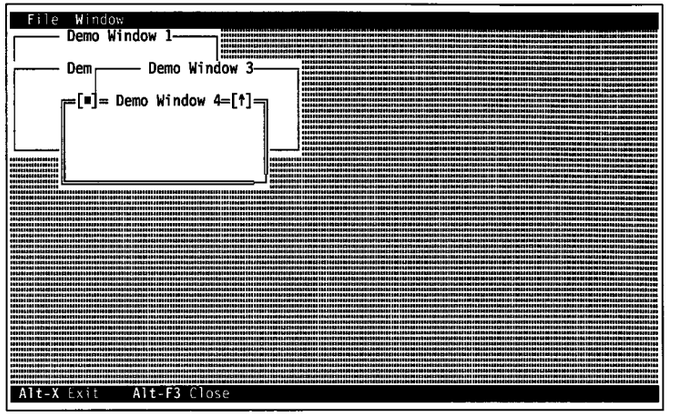
Take some time to play with the program you've written. It has a great deal of capability already. It knows how to open, close, select, move, resize, and zoom multiple windows on the desk top. Not bad for fewer than 150 lines of code!

After **TMyApp** initializes the window, it inserts it into the desk top. As you recall, **TDeskTop** is a group, which means that its purpose is to own and manage subviews, like your window. If you compile and run the code, you'll notice that you can resize, move, and close the new window. Your mouse input is being turned into a series of events and routed from the desk top to the new window, which knows how to handle them.

If you keep invoking *cmNewWin*, more windows will appear on the desk top, each with a unique number. These windows can be resized, selected, and moved over one another. Figure 2.2 shows the desk top as it appears with several windows open.

A **TWindow** object is a group that initially owns one view: a **TFrame** object. The user clicks on the frame's icons to move, resize, or close the window. The frame displays the title that it receives during the window's initialization, and it draws the window's background, just as **TBackGround** does for the desk top. All this happens, as you've seen, without you writing any code.

Figure 2.2
TVGUID04 with multiple windows open



Look through any window

If you were dealing with a traditional window here, the next step would be to write something in it. But a **TWindow** isn't a blank slate to be written on: It's a **TGroup** class, with no screen representation at all beyond its frame view. To put something "in" a window, you need to take an additional step, a step that puts tremendous power in your hands.

To make something appear in the window, you first create a view that knows how to draw itself and then you insert it into the window. This view is called an *interior*.

This first interior will entirely fill the window, but you'll find it easy later to reduce its size and make room for other views. A window can own multiple interiors and any number of other useful views—input lines, labels, buttons, or check boxes. You'll also see how easy it is to place scroll bar views on a window's frame.

You can tile, cascade or overlap the subviews within a group—how the views interact is up to you. **TDeskTop** has member functions, **tile** and **cascade**, that can tile or cascade subviews after they are initialized, but these member functions are available only to the desk top.

The interior you'll create next is a simple class derived from **TView**. Any **TView** can have a frame that operates like a traditional static window frame. By static, we mean that it does not re-

spond to mouse clicks. A **TView**'s frame is outside the clipping region: it's just a line around the view.

If your **TView** interior fills its entire owner window, it doesn't matter if it has a frame—the window's frame covers the interior's frame. If the interior is smaller than the window, the interior frame is visible. Multiple interiors within a window can then be delineated by frames, as you'll see in a later example.

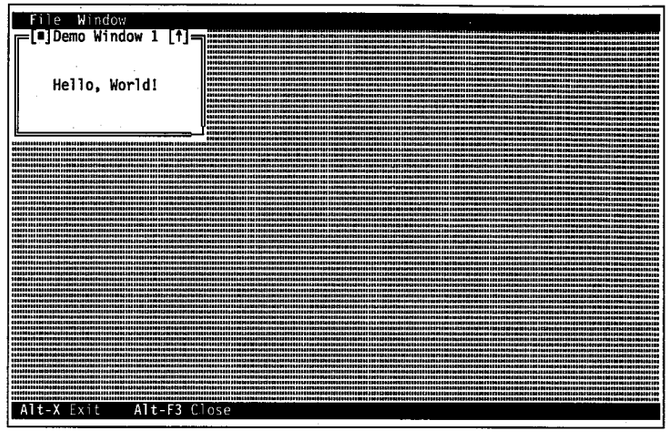
The following code writes "Hello, World!" in the demonstration window; the results are shown in Figure 2.3.

This makes TVGUID05.CPP.

```
#include <stdlib.h>
// for random()

class TInterior : public TView
{

public:

   TInterior(TRect bounds);    // constructor
   virtual void draw();        // override TView::draw
}

TInterior::TInterior(const TRect& bounds) : TView(bounds)
{
   growMode = gfGrowHiX | gfGrowHiY; // make size follow window's
   options = options | ofFramed;
}

void TInterior::draw()
{
   char *hstr = "Hello World!";
   ushort color = getColor(0x0301);
   TView::draw();
   TDrawBuffer b;
   b.moveStr( 0, hstr, color );
   writeLine( 4, 2, 12, 1, b);
}

TDemoWindow::TDemoWindow(const TRect& bounds, const char *aTitle,
                         short aNumber) :  TWindow( bounds, aTitle,
                         aNumber), TWindowInit(
                         &TDemoWindow::initFrame )

{
   TRect r = getClipRect();  // get exposed area
   r.grow(-1,-1);            // make interior fit inside window frame
   insert( new TInterior( r )); // add interior to window }
```

Note that you would need to add `#include <strstrea.h>` for the string stream operations in the previous code example.

Figure 2.3
TVGUID05 with open window



What do you see?

All Turbo Vision views know how to draw themselves. A view's drawing takes place within the member function **draw**. If you create a derived view with a new screen representation, you need to override its base class's **draw** member function and teach the new class how to represent itself on the screen. **TInterior** is derived from **TView**, and it needs a new **draw** member function.

Notice that the new **TInterior::draw** first calls the **draw** of its base class, **TView**, which in this case just clears the rectangle of the view. Normally you would not do this: Your interior view's **draw** member function should take care of its entire region, making the **TView::draw** call redundant.

If you really have something to put into a window's interior, you won't want to call the inherited **draw** member function anyway. Calling **TView::draw** will tend to cause flickering, because parts of the interior are being drawn more than once.

As an exercise, you might try recompiling TVGUID05.CPP with the call to **TView::draw** commented out. Then move and resize the window. This should make quite clear why a view needs to take responsibility for covering its entire region!

Turbo Vision calls a view's **draw** member function whenever the user opens, closes, moves, or resizes views. If you need to ask a view to redraw itself, call **drawView** instead of **draw**. **drawView**

draws the view only if it is exposed. This is important: You override **draw**, but never call it directly; you call **drawView**, but you never override it!

A better way to write

While you can make **printf**, **puts**, and the standard C++ stream I/O work in Turbo Vision, they are often the wrong tools for the job. First, if you simply write something, there's no way you can keep a window or other view from eventually coming along and obliterating it. Second, you need to write to the current view's local coordinates, and clip to the view's boundary. Third, there is the question of what color to use when writing. In addition, **TView** has several special drawing tools, such as **writeLine**, as used in TVGUID05. For example, some use a **TDrawBuffer** object while others take string and character arguments directly. Draw buffers are covered in TVGUID06.

TView::writeStr not only knows how to write with local coordinates and how to clip a view within its parent's boundaries, but also how to use the view's color palette. **writeStr** takes *x*- and *y*-coordinates, the string to be written, and a color index as arguments:

```
void writeStr( short x, short y, const char *str, uchar color);
```

Similar to **writeStr** is **writeChar**, declared in **TView** as

```
void writeChar( short x, short y, char ch, uchar color, short count);
```

Like **writeStr**, **writeChar** positions its output at x- and y-coordinates within the view, but it writes *count* copies of the character *ch*. Both functions write in the color indicated by the *color*'th entry in the view's palette.

These **write*XXX*** member functions should be called only from within a view's **draw** member function. That's the only place you need to write anything in Turbo Vision.

A simple file viewer

In this section you'll add some new functionality to your window and put something real in the interior. You'll add member functions to read a text file from disk and display it in the interior.

Warning! This program will display some "garbage" characters. Don't worry—we did that on purpose.

This is from TVGUID06.CPP.

```
const char *fileToRead = "tvguid06.cpp";
const int maxLineLength = maxViewWidth + 1;
const int maxLines = 100;
char *lines[maxLines];
int lineCount = 0;

void readFile( const char *fileName )
{
 ...
// read fileName into the lines string array
 ...
}

void TInterior::draw()
{
    for( int i = 0; i < size.y; i++ )
writeCStr( 0, i, lines[i], 1 );
}

int main()
{
    readFile( fileToRead );
    TMyApp myApp;
    myApp.run();
    deleteFile();      // delete the lines string array
    return 0;
}
```

Reading a text file

Your application needs to call **readFile** to load the text file into the *lines* array.

Buffered drawing

You will notice that when you run this program, there are "garbage" characters displayed on the screen where there should be empty lines. That's a result of the incomplete **draw** member function. It violates the principle that a view's **draw** member function needs to cover the entire area for which the view is responsible.

Also, the text array *lines* is not really in the proper form to be displayed in a view. Text typically consists of variable length strings, many of which will be of zero length. Because the **draw** member function needs to cover the entire area of the interior, the text lines need to be padded to the width of the view.

The draw buffer

To take care of this, create a new **draw** that assembles each line in a buffer before writing it in the window. The class **TDrawBuffer** offers various **move*XXX*** and **put*XXX*** member functions to manipulate **draw** buffers. **TView** has corresponding **write*XXX*** functions to display (write) a **draw** buffer on the screen.

TDrawBuffer objects hold alternating attribute and character bytes. You can write a **draw** buffer to the screen using **TView::writeBuf**, declared as follows in VIEWS.H:

```
void writeBuf(short x, short y, short w, short h, const void far *b);
void writeBuf(short x, short y, short w, short h, const TDrawBuffer&
              b);
```

These two functions write the buffer given by *b* to the screen starting at the coordinates (*x*,*y*), and filling the region of width *w* and height *h*. The first form draws from a character/attribute array of words (with the character in the low byte and the attribute in the high byte); the other draws from an instance of **TDrawBuffer**. You should only call **writebuf** from within **draw** member functions.

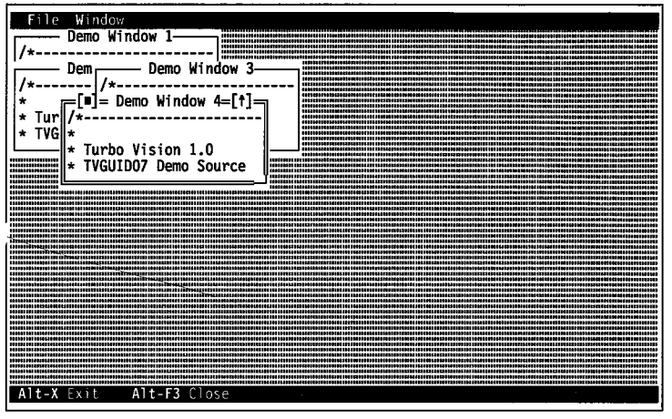
The new **TInterior::draw** looks like this:

This is from TVGUID07.CPP.

```
void TInterior::draw()
{
    ushort color = getColor(0x0301);
    for( int i = 0; i < size.y; i++ )
        {
        TDrawBuffer b;
        b.moveChar( 0, ' ', color, size.x );
        // fill line buffer with spaces
        if( lines[i] )
            {
            char s[maxLineLength];
            strncpy( s, lines[i], size.x );
            s[size.x] = EOS;
            b.moveStr( 0, s, color );
            }
        writeLine( 0, i, size.x, 1, b);
        }
}
```

Figure 2.4 shows *TVGUID07* with several windows open.

Figure 2.4
Multiple file views



draw first uses **TDrawBuf::moveChar** to move *size.x* spaces (the width of your interior) of the proper color into **b**, a **TDrawBuffer** object. Now every line it writes will be padded with spaces to the width of the interior. Next, **draw** uses **b.moveStr** to copy a text line into **b**, then displays it with a **writeLine** call.

Moving text into a buffer

TDrawBuffer has four member functions for moving text into a **TDrawBuffer**: **moveStr**, **moveChar**, **moveCStr**, and **moveBuf**. They move strings, characters, control strings (strings with tildes for menus and status items), and other buffers, respectively, into the calling buffer. All these functions are explained in detail in Chapter 13.

Writing buffer contents

*All the **writeXXX** variants are explained in detail in chapter 13.*

TView has five member functions for writing fixed strings or the contents of a buffer to a view. Two of them, **writeBuf** and **writeLine**, take a parameter of type **TDrawBuffer.** (The other three are **writeChar**, **writeStr**, and **writeCStr**.) All five functions are unbuffered and must only be used within a **draw** member function. You've already seen **writeBuf**, **writeStr**, and **writeChar**. Here are the prototypes for **writeLine** and **writeCStr**:

```
void writeLine(short x, short y, short w, short h,
               const TDrawBuffer& b);
void writeLine(short x, short y, short w, short h, const void far
               *b);
void writeCStr(short x, short y, char far *str, uchar color);
```

In **TInterior::draw**, **writeLine** writes a **TDrawBuffer** object on one line. If the fourth argument, *h* (for height), is greater than 1, **writeLine** repeats the buffer on subsequent lines. Thus, if *buf* holds "Hello, World!", `writeLine( 0,0,13,4,buf )` will write

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

`writeBuf( x, y, w, h, buf )` will also write to a rectangular area of the screen. *w* and *h* refer to the width and height of the buffer. If *buf* holds "ABCDEFGHIJKLMNOP", `writeBuf( 0,0,4,4,buf )` will write

```
ABCD
EFGH
IJKL
MNOP
```

Unlike **writeStr**, **writeCStr**, and **writeChar**, **writeBuf** and **writeLine** do not take a color palette argument. This is because colors are specified when the text is moved into the buffer, meaning that text with differing attributes can appear in the same buffer. The **writeCStr** variant copes with the highlighting of characters surrounded by tildes.

Knowing how much to write

Note that **TInterior.draw** draws just enough of the file to fill the interior. Otherwise, **draw** would spend much of its time writing parts of the file that would just end up being clipped by the boundaries of **TInterior**.

If a view requires a lot of time to draw itself, you can first call **getClipRect** declared as follows:

```
TRect TView::getClipRect();
```

getClipRect returns the rectangle that is exposed within the owning view, so you only need to draw the part of the view that is exposed. For example, when the user moves a complex dialog box in order to look at something behind it, calling **getClipRect** before drawing would save having to redraw the parts of the dialog box that are temporarily off the screen. Don't confuse **getClipRect** with **getExtent**. **getExtent** returns the current bounds of a view, regardless of how much of it is exposed.

Scrolling up and down

Obviously, a file viewer isn't much use if you can only look at the first few lines of the file. So next you'll change the interior to a scrolling view, and give it scroll bars, so that **TInterior** becomes a scrollable window on the textfile. You'll also change **TDemoWindow**, giving it a **makeInterior** member function to separate that function from the mechanics of opening the window.

This is TVGUID08.CPP.

```
class TDemoWindow : public TWindow   // define a new window class
{

public:

    TDemoWindow( const TRect& bounds, const char *aTitle, short
                 aNumber );
    void makeInterior();

};

class TInterior : public TScroller
{

public:

    TInterior( const TRect& bounds, TScrollBar *aHScrollBar,
               TScrollBar *aVScrollBar ); // constructor
    virtual void draw();              // override TView::draw };

// TInterior definitions

TInterior::TInterior( const TRect& bounds, TScrollBar *aHScrollBar,
                      TScrollBar *aVScrollBar ) :
      TScroller( bounds, aHScrollBar, aVScrollBar ) {
    growMode = gfGrowHiX | gfGrowHiY;
    options = options | ofFramed;
    setLimit( maxLineLength, maxLines );
}

void TInterior::draw()       // modified for scroller {
    ushort color = getColor(0x0301);
    for( int i = 0; i < size.y; i++ )
        // for each line:
        {
        TDrawBuffer b;
        b.moveChar( 0, ' ', color, size.x );
        // fill line buffer with spaces
        int j = delta.y + i;       // delta is scroller offset
        if( lines[j] )
            {
            char s[maxLineLength];
```

Note that you have changed the base class of ***TInterior****.*

```
            if( delta.x > strlen(lines[j] ) )
                s[0] = EOS;
            else
                {
                strncpy( s, lines[j]+delta.x, size.x );
                s[size.x] = EOS;
                }
            b.moveStr( 0, s, color );
            }
        writeLine( 0, i, size.x, 1, b);
        }
}

// TDemoWindow definitions

void TDemoWindow::makeInterior()
{
    TScrollBar *vScrollBar =
        standardScrollBar( sbVertical | sbHandleKeyboard );
    TScrollBar *hScrollBar =
        standardScrollBar( sbHorizontal |  sbHandleKeyboard );
    TRect r = getClipRect();    // get exposed view bounds
    r.grow( -1, -1 );           // shrink to fit inside window frame
    insert( new TInterior( r, hScrollBar, vScrollBar )); }

TDemoWindow::TDemoWindow( const TRect& bounds, const char *aTitle,
                          short aNumber) :
        TWindow( bounds, aTitle, aNumber),
        TWindowInit( &TDemoWindow::initFrame )
{
    makeInterior(); // creates scrollable interior and inserts into
                    window
}
```

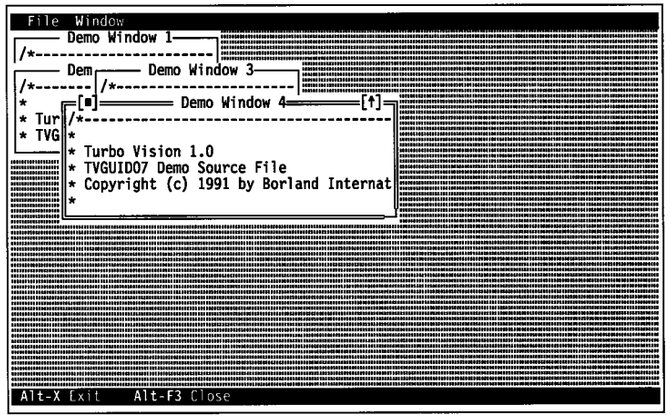


Figure 2.5
File viewer with scrolling interior

The horizontal and vertical scroll bars are initialized and inserted in the group, and are then passed to ***TScroller*** *during its initialization.*

A scroller is a view designed to display part of a larger virtual view. A scroller and its scroll bars cooperate to produce a scrollable view with remarkably little work by you. All you have to do is provide a **draw** member function for the scroller so it displays the proper part of the virtual view. The scroll bars automatically control the scroller values *delta.x* (the column to begin displaying) and *delta.y* (the first line to begin displaying).

You must override a **TScroller**'s **draw** member function in order to make a useful scroller. The *delta* values will change in response to the scroll bars, but it won't display anything by itself. The **draw** member function will be called whenever *delta* changes, so that is where you need to put the response to *delta*.

Multiple views in a window

In the next exercise, you duplicate the interior and create a window with two scrolling views of the text file. The mouse or the tab key automatically selects one of the two interior views. Each view scrolls independently and has its own cursor position.

To do this, you add a section to the **makeInterior** member function so it knows which side of the window the interior is on (since the different sides behave slightly differently), and you make two calls to **makeInterior** in **TDemoWindow**'s constructor.

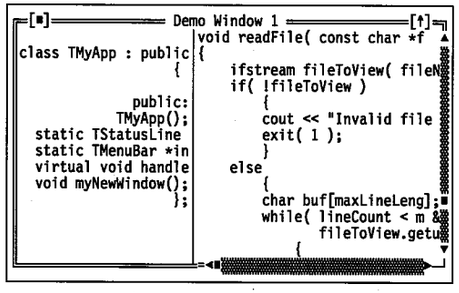
This is TVGUID09.CPP.

Be sure to change the declaration of ***makeInterior****.*

```
// new TDemoWindow constructor
TDemoWindow::TDemoWindow( const TRect& bounds, const char *aTitle,
            short aNumber) :
        TWindow( bounds, aTitle, aNumber),
        TWindowInit( &TDemoWindow::initFrame )
{
        bounds = getExtent();
    TRect r( bounds.a.x, bounds.a.y, bounds.b.x/2+1, bounds.b.y );
    TInterior *lInterior = makeInterior( r, True );
    lInterior->growMode = gfGrowHiY;
    insert( lInterior );
    // creates left-side scrollable interior and inserts into window
    r = TRect( bounds.b.x/2, bounds.a.y, bounds.b.x, bounds.b.y );
    TInterior *rInterior = makeInterior( r, False );
    rInterior->growMode = gfGrowHiX | gfGrowHiY;
    insert( rInterior );
    // likewise for right-side scroller
}

// new makeInterior member function
TInterior *TDemoWindow::makeInterior( const TRect& bounds, Boolean
                                      left )
```

```
{
    TRect r = TRect( bounds.b.x-1, bounds.a.y+1, bounds.b.x,
                     bounds.b.y-1 );
    TScrollBar *vScrollBar = new TScrollBar( r );
    if( vScrollBar == 0 )
        {
        cout << "vScrollbar init error" << endl;
        exit(1);
        }
        // production code would display error dialog box
    vScrollBar->options |= ofPostProcess;
    if( left )
        vScrollBar->growMode = gfGrowHiY;
    insert( vScrollBar );

    r = TRect( bounds.a.x+2, bounds.b.y-1, bounds.b.x-2, bounds.b.y
);
    TScrollBar *hScrollBar = new TScrollBar( r );
    if( hScrollBar == 0 )
        {
        cout << "hScrollbar init error" << endl;
        exit(1);
        }
    hScrollBar->options |= ofPostProcess;
    if( left )
        hScrollBar->growMode = (gfGrowHiY | gfGrowLoY);
    insert( hScrollBar );

    r = bounds;
    r.grow( -1, -1 );
    return new TInterior( r, hScrollBar, vScrollBar ); }
```

Figure 2.6
Window with multiple panes



If you shrink down the windows in TVGUID09.CPP, you'll notice that the vertical scroll bar gets overwritten by the left interior view if you move the right side of the window too close to the left. To get around this, you can set a limit on how small you're allowed to make the window. You do this by overriding the **TWindow** virtual member function **sizeLimits**.

*This is TVGUID10.CPP. Remember to add the **sizeLimits** declaration to **TDemoWindow**.*

```
void TDemoWindow::sizeLimits( TPoint& minP, TPoint& maxP ) {
    TWindow::sizeLimits( minP, maxP );
    minP.x = lInterior->size.x+9;
}
```

Note that you do not have to call **TDemoWindows::sizeLimits** explicitly. You just override it, and it will be called at the appropriate times. This is the same thing you did with the **draw** member function: You told the view *how* to draw itself, but not *when*. Turbo Vision already knew when to call **draw**. The same applies to **sizeLimits**: You set the limits, and the view knows the appropriate times to check them. Since **sizeLimits** is virtual, the correct version for each view type is always called.

Where to put the functionality

You've now created a window with a number of views: a frame and two scrolling interiors, each with two scroll bars. You're on your way to creating a window that can carry out specific functions in an application.

How do you proceed? Suppose you want to turn your window into a fully-fledged text editor. Since the window has two views, you may be tempted to put some of the text-editing functionality into the group, and then have the group communicate with the two views. After all, a group's job is to manage views. Isn't it natural for it to be involved in all the work?

While a group is as capable of being extended as any view, and you can put any functionality in it that you wish, your Turbo Vision applications will be more robust and flexible if you follow these two guidelines: *keep classes as autonomous as possible*, and *keep groups (such as windows) as dumb and devoid of additional functionality as possible.*

Thus, you'd build the text editor by putting all the functionality into the interior view: Create a text editor view type. Views can be easily reusable if you design them properly, and moving your text editor into a different environment wouldn't be very easy if its editing functionality were divided between a group and a view.

Making a dialog box

Classes used:
TView
TGroup
TDialog
TCluster
TCheckBoxes
TRadioButtons
TLabel
TInputLine.

A dialog box is just a special kind of window. In fact, **TDialog** is derived from **TWindow**, and though you *can* treat it as just another window, you will usually do some things differently.

Building on your demonstration program, you'll add a new menu item that generates a command to open a dialog box, add a member function to your application that knows how to do that, and add a line to the application's **handleEvent** member function to link the command to the action.

Note that you do not need to derive a new class from **TDialog** as you did with **TWindow** (to produce **TDemoWindow**). Rather than creating a special dialog box type, add the intelligence to the application: Instead of instantiating a dialog box class that knows what you want it to do, instantiate a generic dialog box and *tell* it what you want it to do.

You will rarely find it necessary to derive a class from **TDialog**, since the only difference between any two dialog boxes is what they contain, not how the dialog boxes themselves work.

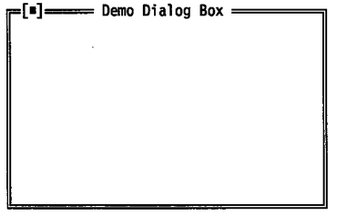
This is TVGUID11.CPP.

```
// added for dialog menu
const int cmNewDialog  = 202;

TMenuBar *TMyApp::initMenuBar( TRect r )
{
    r.b.y = r.a.y + 1;    // set bottom line 1 line below top
                          // line
    return new TMenuBar( r, *new TSubMenu( "~F~ile", kbAltF )+
       *new TMenuItem( "~O~pen", cmMyFileOpen, kbF3, hcNoContext,
                       "F3" )+
       *new TMenuItem( "~N~ew", cmMyNewWin, kbF4, hcNoContext, "F4"
                       )+
       newLine()+
       *new TMenuItem( "E~x~it", cmQuit, cmQuit, hcNoContext,
                       "Alt-X" )+
       *new TSubMenu( "~W~indow", kbAltW )+
       *new TMenuItem( "~N~ext", cmNext, kbF6, hcNoContext, "F6" )+
       *new TMenuItem( "~Z~oom", cmZoom, kbF5, hcNoContext, "F5" )+
       *new TMenuItem( "~D~ialog", cmNewDialog, kbF2, hcNoContext,
                       "F2" )
       // new dialog menu added here
       );
}

// TMyApp needs newDialog member
```



```
void TMyApp::newDialog()
{
    TRect r( 0, 0, 40, 13 );
    r.move( random(39), random(10) );
    deskTop->insert( new TDialog( r, "Demo Dialog" )); }

// modify handleEvent to handle cmNewDialog
void TMyApp::handleEvent(TEvent& event)
{
    TApplication::handleEvent(event);
    if( event.what == evCommand )
        {
        switch( event.message.command )
            {
            case cmMyNewWin:
                newWindow();
                break;
            case cmNewDialog:
                newDialog();
                break;
            default:
                return;
            }
        clearEvent( event );      // clear event after handling
        }
}
```

Figure 2.7
Simple dialog box



There are few differences between this dialog box and your earliest windows, except for the following:

- The default color of the dialog box is gray instead of blue.
- The dialog box is not resizable or zoomable.
- The dialog box has no window number.

Note that you can close the dialog box by clicking on its close icon, clicking the *Alt-F3* status line item, or pressing the *Esc* key. By default, the *Esc* key cancels the dialog box.

This is an example of what is called a non-modal (or "modeless") dialog box. Dialog boxes are usually *modal*, which means that they

define a mode of operation. Usually when you open a dialog box, the dialog box is the only thing active: it is the *modal view*. Clicking on other windows or the menus will have no effect as long as you are in the dialog box's mode. There may be occasions when you want to use non-modal dialog boxes, but in the vast majority of cases, you will want to make your dialog boxes modal.

Modal views are discussed in Chapter 4, "Views."

Executing a modal dialog box

So how do you make your dialog box modal? It's really very easy. Instead of inserting the dialog box class into the desk top, you execute it, by calling the **deskTop->execView** function:

This is from TVGUID12.CPP.

```
// changed from tvguid11: now calls execView
void TMyApp::newDialog()
{
   TRect r( 0, 0, 40, 13 );
   r.move( random(39), random(10) );
   deskTop->execView( new TDialog( r, "Demo Dialog" )); }
```

A **TDialog** object already knows how to respond to an *Esc* key event (which it turns into a *cmCancel* command) and an *Enter* key event (which is handled by the dialog box's default **TButton**). A dialog box always closes in response to a *cmCancel* command.

Calling **execView** inserts the dialog box into the group and makes the dialog box modal. Execution remains in **execView** until the dialog box is closed or canceled. **execView** then removes the dialog box from the group and exits. For the moment, you can ignore the value returned by the **execView** function and stored in *control*. You'll make use of this value in *TVGUID16*.

Taking control

Of course, a dialog box with nothing in it is not much of a dialog box! To make this interesting, you need to add *controls*. Controls are various elements within a dialog box that allow you to manipulate information. The important thing to remember about controls is that they only affect things within the dialog box.

Command handling is explained more in Chapter 5, "Event-driven programming."

The only exception to this rule is the case of a button in a modeless dialog box. Because buttons generate commands, those commands will spread downward from the current modal view. If the dialog box is not the modal view, those commands will go to places outside the dialog box, which may have unintended effects.

In general, when setting up controls in a dialog box, you can separate the visual presentation from the handling of data. This means you can easily design an entire dialog box without having to create the code that sets up or uses the data provided in the dialog box, just as you were able to set up menus and status items without having code that acted on the commands generated.

Button, button...

One of the simplest control classes is the **TButton**. It works very much like a fancy status line item: It's a colored region with a text label on it, and if you click on it, it generates a command. There is also a shadow "behind" the button, so that when you click on the button it gives a sort of three-dimensional movement effect.

Most dialog boxes have at least one or two buttons. The most common are buttons for "OK" (meaning "I'm done. You may close the dialog box and accept the results.") and "Cancel" (meaning "I want to close the dialog box and ignore any changes made in it."). A Cancel button will usually generate the same *cmCancel* command that the close icon produces.

There are five standard dialog commands that can be bound to a **TButton**: *cmOk, cmCancel, cmYes, cmNo,* and *cmDefault*. The first four commands also close the dialog box by having **TDialog** call its **endModal** member function, which restores the previous modal view to modal status.

You can also use buttons to generate commands specific to your application.

This is from TVGUID13.CPP.

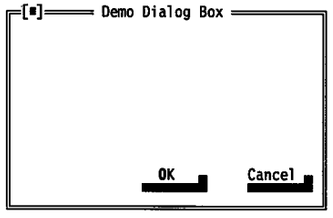
```
// changed from tvguid12: add buttons
void TMyApp::newDialog()
{
   TDialog *pd = new TDialog( TRect( 20, 6, 60, 19), "Demo Dialog" );
   if( pd )
   {
      pd->insert( new TButton( TRect( 15, 10, 25, 12 ), "~O~K", cmOk,
                  bfDefault ));
      pd->insert( new TButton( TRect( 28, 10, 38, 12 ), "~C~ancel",
                  cmCancel, bfNormal ));
      deskTop->execView( pd );
    }
    destroy( pd );
}
```

Creating a button requires four arguments for the constructor:

1. The region the button will cover (Remember to leave room for the shadow!).
2. The text that will appear on the button.
3. The command to be bound to the button.
4. A flag indicating the type of button (normal or default).

After the user has closed or canceled the dialog box, execution leaves **execView**, and the dialog box object is eliminated. Note the use of **destroy** rather than the usual C++ **delete** you might expect to find after a **new**. All objects that have been derived from **TObject** and created with **new** must be disposed of using the **destroy** member function inherited from **TObject**. Just like **delete**, **destroy** takes a single pointer-to-object argument. If you are interested in the whys and wherefores, you can read more about Turbo Vision's memory management in Chapter 6 and also in Chapter 13 (see **TVMemMgr** and **TObject** class references). For now, it is sufficient to know that **destroy** takes care of all the delicate internal housekeeping needed when you dispose of objects derived from **TObject**, ensuring that the states of all objects related to the one you are destroying are reset in the proper manner.

Figure 2.8
Dialog box with buttons



Notice that you didn't highlight the "C" in "Cancel" because there is already a hot key (*Esc*) for canceling the dialog box. This leaves *C* available as a shortcut for some other control.

Normal and default buttons

Whenever you create a button, you give it a flag, either *bfNormal* or *bfDefault*. Most buttons will be *bfNormal*. A button flagged with *bfDefault* will be the default button, meaning that it will be "pressed" when you press the *Enter* key. Turbo Vision does not check to ensure that you have only one default button—that is your responsibility. If you designate more than one default control, the results will be unpredictable.

Usually, the "OK" button in a dialog box is the default button, and users become accustomed to pressing *Enter* to close a dialog box and accept changes made in it.

Focused controls

When a dialog box is open, one of the controls in it is always highlighted. That control is called the active, or *focused*, control. Focus of controls is most useful for directing keyboard input and for activating controls without a mouse. For example, if a button has the focus, the user can "press" the button by pressing *Spacebar*. Characters can only be typed into an input line if the input line has the focus.

Labels are discussed later in this chapter.

The user can press the *Tab* key to move the focus from control to control within the dialog box. Labels won't accept the focus, so the *Tab* key skips over them.

Tab order is important!

You will want the user to be able to *Tab* around the dialog box in some logical order. The *Tab* order is the order in which the objects were inserted into the dialog box. Internally, the objects owned by the dialog box are maintained in a circular linked list, with the last object inserted linked to the first object.

By default, the focus ends up at the last object inserted. You can move the focus to another control either by using the dialog box's **selectNext** member function or by calling the control's **select** member function directly. **selectNext** allows you to move either forward or backward through the list of controls. The code `selectNext(False)` moves you forward through the circular list (in *Tab* order); `selectNext(True)` moves you backward.

Take your pick

Often, the choices you want to offer your users in a dialog box are not simple ones that can be handled by individual buttons. Turbo Vision provides several useful standard controls for allowing the user to choose among options. Two of the most useful are check boxes and radio buttons.

Check boxes and radio buttons function almost identically, with the exception that you can pick as many (or as few) of the check boxes in a set as you want, but you can pick one, and only one, radio button. The reason the two sets appear and behave so similarly is that they are both derived from a single Turbo Vision class, **TCluster**.

Creating a cluster

There is probably no reason you would ever want to create an instance of a plain **TCluster**. Since the process for setting up a check box cluster is the same as that for setting up a cluster of radio buttons, you only need to look at the process in detail once.

Add the following code to the **TMyApp::newDialog** member function, *after* the dialog box is created but *before* the buttons are added. Keep the buttons as the last items inserted so they will also be last in *Tab* order.


```
TView *b = new TCheckBoxes( TRect(3, 3, 18, 6),
           new TSItem( "~H~varti",
           new TSItem( "~T~ilset",
           new TSItem( "~J~arlsberg", 0) )));
pd->insert(b);
```

The initialization is quite simple. You designate a rectangle to hold the items (remembering to allow room for the check boxes themselves), and then create a linked list of pointers to strings that will show up next to the check boxes, terminated by 0.

Check box values

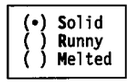
The preceding code creates a set of check boxes with three choices. You may have noticed that you gave no indication of the settings for each of the items in the list. By default, they will all be unchecked. But often you will want to set up boxes where some or all of the entries are already checked. Rather than assigning values when you set up the list, Turbo Vision provides a way to set and store values easily, outside the visual portion of the control.

A set of check boxes may have as many as 16 entries. A **ushort** data member called *value* maps its 16 bits to the items to be checked.

After you finish constructing the dialog box as a whole, you will look at how to set and read the values of all the controls. For now, concentrate on getting the proper controls in place.

One more cluster

Before moving on, however, add a set of radio buttons to the dialog box so you can compare them with check boxes. The following code sets up a set of three radio buttons next to your check boxes:




```
b = new TRadioButtons( TRect( 22, 3, 34, 6),
    new TSItem( "~S~olid",
    new TSItem( "~R~unny",
    new TSItem( "~M~elted", 0 )  )));
pd->insert( b );
```

The main differences between the check boxes and the radio buttons are that you can only select one radio button in the group, and the first item in the list of radio buttons is selected by default.

Since you don't need to know the state of every radio button (only one can be on, so you only need to know *which* one it is), radio button data is not bitmapped. This means you can have more than just 16 radio buttons, if you choose, but since the button state is still stored in 16 bits, you are limited to 65,536 radio buttons per cluster. This should not be a serious impediment to your design. A value of zero indicates the first radio button is selected, a one indicates the second button, a two the third, and so on.

Labeling the controls

Of course, setting up controls may not be sufficient. Simply offering a set of choices may not tell the user just *what* he is choosing! Turbo Vision provides a handy member function for labeling controls in the form of another control, the **TLabel**.

There's more to the **TLabel** class than appears at first glance. A **TLabel** object not only displays text, it is also bound to another view. Clicking on a label will move the focus to the bound view. You can also define a shortcut letter for a label by surrounding the letter with tildes (~).

To label your check boxes, add the following code right after you insert the check boxes into the dialog box:

```
pd->insert( new TLabel( TRect( 2, 2, 10, 3), "Cheeses", b));
```

You can now activate the set of check boxes by clicking on the word "Cheeses." This also lets the uninformed know that the items in the box are, in fact, cheeses.

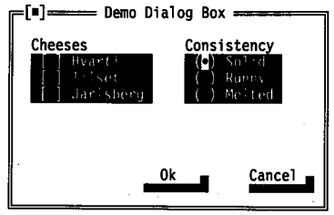
Similarly, you can add a label to your radio buttons with the following code:

This is TVGUID14.CPP.

```
insert( new TLabel( TRect( 21, 2, 33, 3), "Consistency", b));
```

Resulting in the following dialog box:

Figure 2.9
Dialog box with labeled clusters added



The input line class

There is one other fairly simple kind of control you can add to your dialog box: an item for editing string input, called an input line. Although the workings of the input line are fairly complex, from your perspective as a programmer **TInputLine** is a very simple class to use.

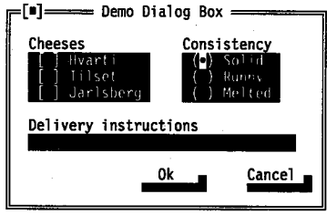
Add the following code after the code for labeling the radio buttons and before you execute the dialog box:

This is TVGUID15.CPP.

```
// add input line
b = new TInputLine( TRect( 3, 8, 37, 9 ), 128 );
pd->insert( b );
pd->insert( new TLabel( TRect( 2, 7, 24, 8 ), "Delivery
                        Instructions", b ));
```

Setting up an input line is simplicity itself: You assign a rectangle that determines the length of the input line within the screen. The only other argument required is one defining the maximum length of the string to be edited. That length may exceed the displayed length because the **TInputLine** class knows how to scroll the string forward and backward. By default, the input line can handle keystrokes, editing commands, and mouse clicks and drags.

Figure 2.10
Dialog box with input line added



The input line also has a label for clarity, since unlabeled input lines can be even more confusing to users than unlabeled clusters.

Setting and getting data

Now that you have constructed a fairly complex dialog box, you need to figure out how to *use* it. You have set up the user interface end; now you need to set up the program interface. Having controls isn't much help if you don't know how to get information from them!

There are basically two things you need to be able to do: Set the initial values of the controls when the dialog box is opened, and read the values back when the dialog box is closed. Note that you don't want to modify any data outside the dialog box until you successfully close the box. If the user decides to cancel the dialog box, you have to be able to ignore any changes made while the dialog box was open.

Luckily, Turbo Vision facilitates doing just that. Your program passes a packet of information to a dialog box when it is opened. When the user ends the dialog box, your program needs to check to see if the dialog box was canceled or closed normally. If it was canceled, you can simply proceed without modifying the structure. If the dialog box closed successfully, you can read back a structure from the dialog box in the same form as the one given to it.

The virtual member functions **setData** and **getData** copy data to and from a view. Every view has both a **setData** and **getData** member function.

When a group (such as **TDialog**) is initialized through a **setData** call, it passes the data along by calling each of its subviews' **setData** member functions.

When you call a group's **setData**, you pass it a data structure that contains the data for each view in the group. You need to arrange each view's data in the same order as the group's views were inserted.

You also need to make the data the proper size for each view. Every view has a member function called **dataSize** which returns the size of the view's data space. Each view copies **dataSize** amount of data from the data structure, then advances a pointer to tell the next view where to begin. If a subview's data is the wrong size, each subsequent subview will also copy invalid data.

If you create a new view and add data fields to it, don't forget to override **dataSize**, **setData**, and **getData** so they handle the proper values. The order and sizes of the data in the data structure is entirely up to you. The compiler will return no errors if you make a mistake.

After the dialog box executes, your program should first make sure the dialog box wasn't canceled, then call **getData** to copy the dialog box's information back into your application.

So, in your example program, you initialize in turn a cluster of check boxes, a label, a cluster of radio buttons, a label, an input line of up to 128 characters, a label, and two buttons (Ok and Cancel). Table 2.1 summarizes the data requirements for each of these.

Table 2.1
Data for dialog box controls

Control	Data required
check boxes	**ushort**
label	none
radio buttons	**ushort**
label	none
input line	**string**[128]
label	none
button	none
button	none

Views that have no data (such as labels and buttons) use the **getData** member function they inherit from **TView**, which does nothing at all, so you don't need to concern yourself with them here. This means that when getting and setting data, you can skip over labels and buttons.

Thus, you are only concerned with three of the views in the dialog box: the check boxes, the radio buttons, and the input line. As noted earlier, each of the cluster items stores its data in a **ushort** data member. The input line's data is stored in a string. You can set up a data structure for this dialog box in a global type declaration:

```
struct DialogData
{
   ushort checkBoxData;
   ushort radioButtonData;
   char inputLineData[128];
};
```

Now all you have to do is initialize the structure when you start up the program (**myApp**'s constructor is a good place), set the data

when you enter the dialog box, and read it back when the dialog box closes successfully. Once you've declared the type as we did here, you declare a pointer:

```
DialogData *demoDialogData;
```

then add one line before executing the dialog box and one after:

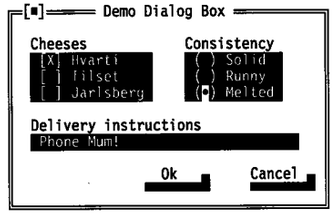
```
// we save the dialog data:
pd->setData( demoDialogData );
ushort control = deskTop->execView( pd );
// and read it back when the dialog box is successfully closed
if( control != cmCancel )
    pd->getData( demoDialogData );
```

and add three lines to **TMyApp**'s constructor to set the initial values for the dialog box:

This is from TVGUID16.CPP.

```
demoDialogData->checkboxData = 1;
demoDialogData->radioButtonData = 2;
strcpy( demoDialogData->inputLineData, "Phone Mum!");
```

Figure 2.11
Dialog box with initial values set



Now any changes you make to the dialog box should be there when you reopen it, as long as you didn't cancel the dialog.

One of the things we learned as we wrote the Borland integrated environments was that it is a good idea to have your program store information that gets altered by a dialog box in the form of a structure that can be used for setting or getting data from the dialog box. This keeps you from having to construct lots of data structures from discrete variables every time you want to open a dialog box, and from having to disperse the information returned from a dialog box to various variables when it's done.

Hot keys and conflicts

By default, labels, check boxes, and radio buttons can respond to hot keys even when the focus is elsewhere within the dialog. For example, when your demo dialog box first opens, the focus is in

the check boxes, and the cursor is on the first check box. Pressing an *M* for "Melted" will immediately move the focus to the **M**elted radio button and turn it on.

While you obviously want hot keys to be as mnemonic as possible, there are only 26 letters and 10 digits available. This may cause some conflicts. For example, in your demo dialog box it would make sense to have *C* as the hot key for "Cheeses," "Consistency," and maybe a cheese called "Cheddar." There are a couple of ways to deal with such situations.

First, while it is nice to have the first letter of a word be the hot key, it is not always possible. You can resolve the conflict between "Cheeses" and "Consistency," for example, by making the letter *O* the hot key for "Consistency," but the result is not as easy to remember. Another way, of course, is to relabel something. Instead of the label "Cheeses," you could label that cluster "Kind of Cheese," with *K* as the hot key.

This sort of manipulation is the only way around conflicts of hot keys at the same level. However, there is another approach you can take if the conflict is between, say, a label and a member of a cluster: Hot keys can be made local within a dialog box item. In the previous example, for example, if you localize the hot keys within each cluster, pressing *M* when the check boxes are focused will *not* activate the "Consistency" buttons or the "Melted" button. *M* would only function as a hot key if you clicked or tabbed into the "Consistency" cluster first.

The options data member and the ofPostProcess bit are both explained in Chapter 4.

By default, all hot keys are active over the entire dialog box. If you want to localize hot keys, change the default *options* data member for the object you are about to insert into the dialog box. For example, if you want to make the hot keys in your check boxes local, you would add another line before inserting into the dialog box:

```
TView *b = new TCheckBoxes( TRect( 3, 3, 18, 6),
    new TSItem( "~H~varti",
    new TSItem( "~T~ilset",
    new TSItem( "~J~arlsberg", 0 )
    )));
(b->options) &= (!ofPostProcess); // turn it off
pd->insert( b );
```

Now the *H*, *T*, and *J* hot keys only operate if you click or *Tab* into the "Cheeses" cluster first. *Alt-H*, *Alt-T*, and *Alt-J* will continue to function as before, however.

See 129 in Chapter 5 for more explanation.

Keep in mind that a label never gets the focus. Therefore, a label must have its *ofPostProcess* bit on for its hot key to operate.

Having *ofPostProcess* set means that the user can enter information in a dialog box quickly. However, there are some possible drawbacks. A user may press a hot key expecting it to go to one place, but because of a conflict it goes somewhere else. Similarly, if the user *expects* hot keys to be active, but they're only active locally, it could be confusing to have a hot key do nothing when it is pressed outside the area where it is active.

The best advice we can give you is to test your dialog boxes carefully for conflicts. Avoid having duplicate hot keys when possible, and always make it clear to the user which options are available.

Other dialog box controls

There are some additional ready-made dialog mechanisms that weren't used in this example. They are used in the same way as the items you did use: You create a new instance, insert it into the dialog box, and include any appropriate data in the data structure. This section briefly describes the functions and usage of each one. Much more detail is contained in Chapter 13, "Class reference."

Static text

TStaticText is a view that simply displays the string passed to it. The string is word wrapped within the view's rectangle. The text will be centered if the string begins with a *Ctrl-C*; line breaks can be forced with *Ctrl-M*. By default, the text can't get the focus, and the object gets no data from the data structure.

List viewer

A **TListViewer** displays a single- or multiple-column list from which the user can select items. A **TListViewer** can also communicate with two scroll bars.

TListViewer is meant to be a building block, and is not usable by itself. It has the ability to handle a list, but does not itself contain a list. Its abstract member function **getText** loads the list members

for its **draw** member function. A working class derived from **TListViewer** needs to override **getText** to load actual data.

List box

TListBox is a working class derived from **TListViewer**. It owns a **TCollection** that is assumed to be pointers to strings. **TListBox** only supports one scroll bar. An example of a list box is the file selection list in the Borland integrated environment, or the sample program FILEVIEW.CPP provided on your diskettes.

Getting and setting data with list boxes is greatly facilitated by the use of the **TListBoxRec** structure type, which holds a pointer to a collection containing the list of strings to be displayed and a word indicating which item is currently selected in the list.

History

THistory implements an object that works together with an input line and a related list box. By clicking on the arrow icon next to the input line, the user brings up a list of previous values given for the input line, any of which can then be selected. This reduces or even eliminates repetitive typing.

THistory classes are used in many places in the Borland integrated environment, such as the **File** | **O**pen dialog box and in the **S**earch | **F**ind dialog box.

Standard dialog boxes

The STDDLG module contains a pre-built dialog object called **TFileDialog**. You use this dialog box in the integrated environment when you open a file. **TFileDialog** uses other classes in STDDLG which you may find useful:

```
class TFileInputLine: public TInputLine
class TFileCollection: public TSortedCollection
class TSortedListBox: public TListBox
class TFileList: public TSortedListBox
class TFileInfoPane: public TView
```

Chapter 15, "Implementing standard dialog boxes," provides reference material on this set of classes.

PART 2

Using Turbo Vision

C H A P T E R

3

The class hierarchy

This chapter assumes that you have read Part 1 of this book to get an overview of Turbo Vision's philosophy, capabilities, and terminology. Remember also that a good working knowledge of C++ is essential to the use of Turbo Vision.

After some general comments on OOP and hierarchies, this chapter takes you quickly through the Turbo Vision class hierarchies, stressing how the classes are related through the inheritance mechanism. By learning the main properties of each standard class (many of which are related to the class's name in an obvious way), you will gain an insight into how the inherited and new members of each class combine to provide the derived class's functionality.

The overall hierarchy

Chapter 13, "Class reference," describes in depth the member functions and data members of each standard class, but until you acquire an overall feel for how the class hierarchies are structured, you can easily become overwhelmed by the mass of detail. This chapter presents an informal browse through the Turbo Vision hierarchies. The remainder of Part 2 gives more detailed explanations of the components of Turbo Vision and how to use them. Part 3 provides reference material on each class and its members.

Although **TObject** is the root class for almost all classes in Turbo Vision, **TView** (directly derived from **TObject**) is the class from which the majority of the classes directly spring.

The major aspects of the **TObject/TView** hierarchy are shown in Figures 3.1 and 3.1. You'll find that these figures merit careful study. For example, knowing that **TDialog** is derived from **TWindow**, which is derived from **TGroup**, which is derived from **TView**, reduces your learning curve considerably. Each new derived class you encounter already has familiar inherited properties; you simply study whatever additional data members and properties it has over its parent.

There are several examples of multiple inheritance in the hierarchies; for example, **TProgram** is derived from both **TGroup** and **TProgInit**. As is customary in C++, the arrows point *toward* the base class. Classes that represented in these diagrams but are not connected to anything are not part of the hierarchy but are related to the classes they are near.

Figure 3.1
Class hierarchy overview

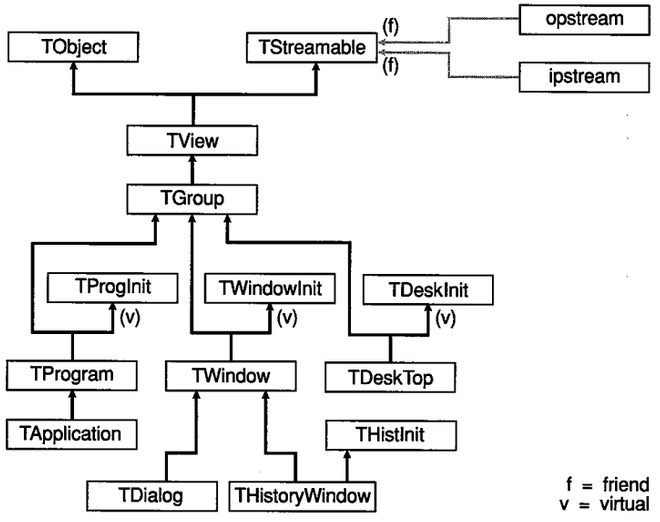
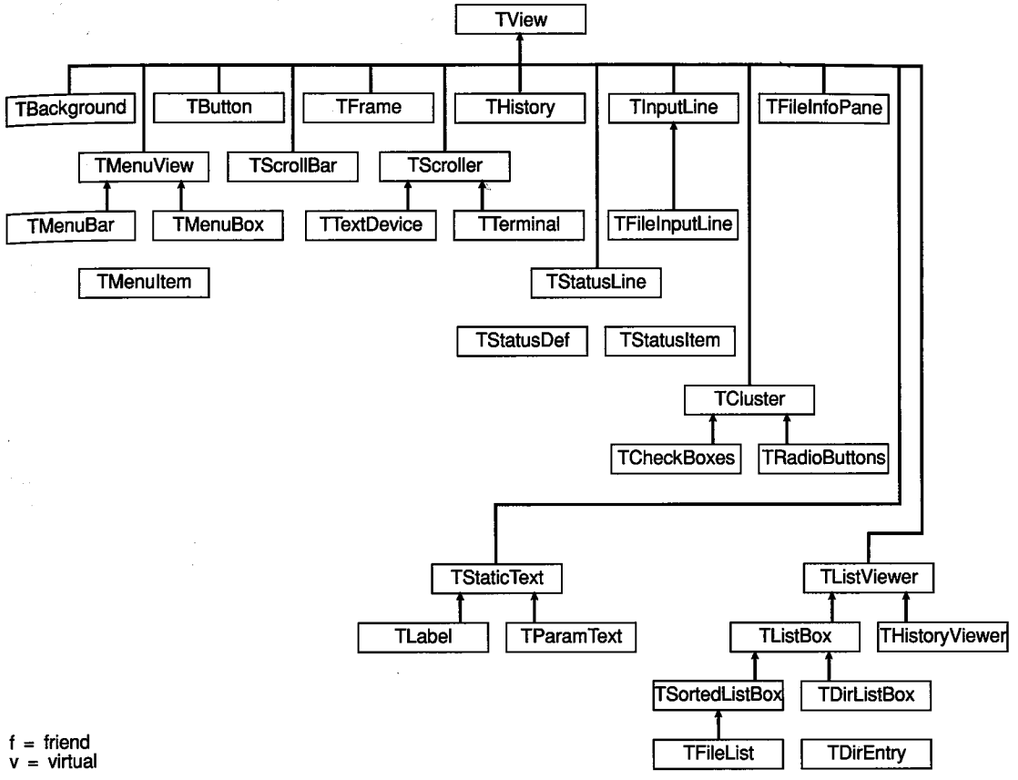


Figure 3.2: TView class hierarchy



A word on creating hierarchies

There is no "perfect" hierarchy for any application. Every class hierarchy is something of a compromise obtained by careful experimentation (and a fair amount of intuition acquired with practice). By noticing how we set things up, you can benefit from our experience in developing your own class hierarchies. Naturally, you can also create your own base classes to achieve special effects beyond the standard classes provided in Turbo Vision.

Mastering the minute details will come later, but as with all OOP projects, the initial overall planning of your new classes is the key to success.

Class typology

Not all classes are created equal in Turbo Vision. You can separate their functions into three distinct groups: primitive classes, view classes, and mute classes. Each of these is described in a separate section of this chapter.

Within each of these groups there are also different sorts of classes, some of which are useful classes that you can instantiate (that is, create objects of that class) and use, and others of which are "seed" classes that serve only as the basis for deriving related, useful classes. Before we look at the classes in the Turbo Vision hierarchy, it will be helpful to understand a little about these "seed" classes. Seed classes are sometimes called "abstract classes" in OOP circles. The strict meaning of a C++ abstract class is one that has at least one pure virtual function. A C++ abstract class *cannot* be instantiated and exists only as a base for abstract or non-abstract derivations. Turbo Vision has several seed classes that cannot usefully be instantiated but are not abstract in the technical C++ sense.

Seed classes

Many classes exist as seeds from which more specialized and immediately useful classes can be derived. The reason for having seed types is partly conceptual but largely serves the practical aim of reducing coding effort.

Take the **TRadioButtons** and **TCheckBoxes** types, for example. They could each be derived directly from **TView** without difficulty. However, they share a great deal in common. This commonality warrants a seed class called **TCluster**. **TRadioButtons** and **TCheckBoxes** are then derived from **TCluster** with the addition of a few specialized member functions to each to provide their individual functionalities.

Seed classes are never usefully instantiated. An instance of **TCluster**, **MyCluster**, for example, would not have a useful **draw** member function: It inherits **TView::draw** without overriding, so **MyCluster::draw** would simply display an empty rectangle of the default color. If you want a fancy cluster of controls with properties different from radio buttons or check boxes, you could derive a **TMyCluster** from **TCluster**, although it might be easier to derive your special cluster from **TRadioButtons** or **TCheckBoxes**; it depends on which is closer to your needs. In all cases, you would add data members, and add or override member functions, with the least possible effort. If your plans include a whole family of fancy clusters, you might find it convenient to create an intermediate seed class.

Empty member functions

Whether you can usefully instantiate a class depends entirely on the circumstances. Many of Turbo Vision's standard classes have empty member functions that *must* be defined in derived classes. Standard classes may also have skeleton member functions offering minimal default actions that may suit your purposes—if not, a derived type will be needed.

A general rule is that as you travel down the Turbo Vision hierarchy, the standard classes become more specialized. Their names reveal the functionality encapsulated in their data members and member functions. For most applications there will be obvious base classes from which you can create a "standard" interface: a desktop, a menu bar, a status line, some dialog boxes, and so on.

Class instantiations and derivations

With any given class there are two basic operations available: You can create an instance of that class (instantiate it), or you can derive a new class. In the latter case, you have a class on which

the previous two operations can again be applied. Let's examine these operations in more detail.

Instantiation

Creating an instance of a class (also known as an object) is usually accomplished by a variable declaration, either static or dynamic:

```
myScrollBar(TRect r);
TButton *someButton = new TButton( ... );
```

MyScrollBar would be initialized by **TScrollBar**'s constructor with certain default data member values. These can be found by consulting **TScrollBar**'s constructor entry in Chapter 13, "Class reference." Since **TScrollBar** is derived from **TView**, **TScrollBar**'s constructor calls **TView**'s constructor to set the data members inherited from **TView**. Similarly, **TView**'s constructor is derived from **TObject**, so it calls **TObject**'s constructor to allocate memory. **TObject** has no parent, so the buck stops there.

The **myScrollBar** object now has default data member values which you may need to change. It also has all the member functions of **TScrollBar** plus the member functions (possibly overridden) of **TView** and **TObject**. To make use of **myScrollBar**, you need to know what its member functions do, especially **handleEvent** and **draw**. If the required functionality is not defined in **TScrollBar**, you need to derive a new type.

Derivation

You can easily derive a new class from an existing one:

```
class TNewScrollBar: public TScrollBar
{
// declare new members here
};

// define new members here
```

You do not yet have any instances of this new class. Before creating any **TNewScrollBar** objects, you need to define new member functions or override some of **TScrollBar**'s member functions and possibly add some new data members; otherwise there would be no reason to derive a new scroll bar class. The new or revised members you define constitute the process of adding functionality to **TScrollBar**. Your new constructor must determine the default values for your new scroll bar classes.

Member functions

Turbo Vision class member functions can be characterized in four (possibly overlapping) ways, each described here.

Empty member functions

In the base class, an empty member function has no defining body (or a body containing a statement to trap illegal calls). Empty member functions are always virtual and *must* be defined by a derived class before they can be used.

Default member functions

In the base class, a default member function has a minimal action defined. It will almost always be overridden by a derived class to be useful, but the member function provides a reasonable default for all classes in the inheritance chain. An example is **TSortedCollection::compare**.

Virtual member functions

Virtual member functions use the **virtual** keyword in their class declarations. A virtual member function can be overridden in derived classes. The overriding function must match the original member function's argument profile and return type exactly. The resulting function is itself virtual whether you repeat the **virtual** keyword or not. Virtual functions need not be overridden, but the usual intention is that they *will* be overridden sooner or later. Examples are **TView::dataSize** and **TGroup::handleEvent**. C++ virtual functions implement the concept of *polymorphism*: functions and the member functions they invoke can be bound to objects dynamically; that is to say, at run time.

Non-virtual member functions

A non-virtual member function is not usually overridden. A derived class can overload or hide a member function by defining a member function with the same name using different arguments and return types, if necessary, but non-virtual member functions do not operate polymorphically. This is most critical when you call member functions. For example, if *pGeneric* is a pointer variable of type **TView***, you can also assign to *pGeneric* a pointer

of type ***TClassName****, where ***TClassName*** is any class derived from **TView**. However, when you dereference the pointer and call a non-virtual member function, the member function called will always be **TView**'s, since that is the type of the pointer as determined at compile time. In other words, *pGeneric->non_virtual-MemberFunction* is *always* equivalent to **TView::non_virtual-MemberFunction**, even if you have assigned a pointer of some other type to *pGeneric*. Contrast this with virtual functions: *pGeneric->virtual_MemberFunction* will take note of the run-time pointer type and invoke the function defined for the object **pGeneric*. This may well be of a type derived from **TView**.

Static members

Static members (both data and functions) are declared using the **static** keyword in the class body declaration. Such members have only one copy shared by all objects in the class. Static member functions do not have a **this** pointer, and they cannot be virtual. Static members play an important role in Turbo Vision. For example, you'll see that **TProgram::initStatusLine** is a static member function that creates the static data member *TProgram::statusLine*.

Turbo Vision data members

If you take an important trio of classes: **TView**, **TGroup**, and **TWindow**, a glance at their data members reveals inheritance at work, and also tells you quite a bit about the growing functionality as you move down the hierarchy (recall that class trees grow downward from the root!).

Table 3.1
Inheritance of view data members

TView data members	**TGroup data members**	**TWindow data members**
cursor	cursor	cursor
dragMode	dragMode	dragMode
eventMask	eventMask	eventMask
growMode	growMode	growMode
helpCtx	helpCtx	helpCtx
next	next	next
options	options	options
origin	origin	origin
owner	owner	owner
size	size	size
state	state	state
	buffer	buffer
	clip	clip
	current	current
	endState	endState
	last	last
	lockFlag	lockFlag
	phase	phase
		flags
		frame
		number
		palette
		title
		zoomRect

Notice that **TGroup** inherits all the data members of **TView** and adds several more that are pertinent to group operation, such as pointers to the current and last views in the group. **TWindow** in turn inherits all of **TGroup**'s data members and adds even more members, which are needed for window operations (such as the title, a pointer to its frame, and the number of the window).

In order to fully understand **TWindow**, keep in mind that a window is both a group and a view.

Primitive classes

Turbo Vision provides several simple classes that exist primarily to be used by other classes or to act as the basis of a hierarchy of more complex classes. **TPoint** and **TRect** are used by all the visible classes in the Turbo Vision hierarchy. **TStatusItem** and **TMenuItem** are used to build status lines and menus. **TObject** is the basis of the view hierarchy.

Note that classes of these types are not directly displayable. **TPoint** is simply a screen-position class (x, y coordinates). **TRect** sounds like a view class, but it just supplies upper-left and lower-right rectangle bounds and several non-display utility member functions and overloaded operators.

TPoint

This class represents a point. Its data members, *x* and *y*, define the cartesian (x,y) coordinates of a screen position. The point (0,0) is the topmost, leftmost point on the screen. *x* increases horizontally to the right; *y* increases vertically downwards. **TPoint** defines several overloaded operators, such as **+**, **==**, and **+=**, that work in a natural way. Other classes have member functions that convert between global (whole screen) and local (relative to a view's origin) coordinates.

TRect

This class represents a rectangle. Its data members, *a* and *b*, are **TPoint** objects defining the rectangle's upper-left *(a.x, a.y)* and lower-right *(b.x, b.y)* points. **TRect** has member functions and operators such as **move**, **grow**, **intersect**, **Union**, **contains**, **==**, **!=**, and **isEmpty**. **TRect** constructors take either four integers or two **TPoint** objects. **TRect** objects are not visible views and cannot draw themselves. However, all views are rectangular: Their constructors take one **TRect** argument, called *bounds*, to determine the region they will cover.

TObject

TObject is the base class for most of the Turbo Vision classes. It provides two member functions beside a destructor: **destroy** and **shutDown**. **destroy** takes the place of the standard C++ operator **delete**; **shutdown** is used internally by **destroy** to ensure correct destruction of derived and related objects.

TObject's derived classes fall into one of two families: views or non-views. Views are derived from **TView**, which gives them special properties not shared by non-views. Views can draw themselves and handle events sent to them. The non-view classes provide a host of utilities for handling streams and collections of other classes, including views, but they are not directly "viewable."

Views

The displayable classes derived from **TObject** give you objects known as views. Such classes are derived from **TView**, a class immediately derived from **TObject**. You should distinguish "visible" from "displayable," since there may be times when a view is wholly or partly hidden by other views.

Views overview

A view is any object that can be drawn (displayed) in a rectangular portion of the screen. A view class must be derived from **TView**. **TView** itself is a seed class representing an empty rectangular screen area. Having **TView** as a base class, though, ensures that each derived view has at least a rectangular portion of the screen and a minimal virtual **draw** member function (forcing all immediate derived classes to supply a specific **draw** member function).

Most of your Turbo Vision programming will use the more specialized derived classes of **TView**, but the functionality of **TView** permeates the whole of Turbo Vision, so you'll need to understand what it offers.

Groups

The importance of **TView** is apparent from the hierarchy chart shown in Figures 3.1 and 3.1 on pages 74 and 75. Everything you can see in a Turbo Vision application derives in some way from **TView**. But some of those visible classes are also important for another reason: They allow several classes to act in concert.

The TGroup class

TGroup lets you handle dynamically chained lists of related, interacting subviews via a designated view called the *owner* of the group. Each view has an owner data member of type **TGroup* that points to the owning **TGroup** class. A null pointer means that the view has no owner. A data member called *next* provides a link to the next view in the view chain. Since a group is a view, there can be subviews that are groups owning their own subviews, and so on.

The state of the chain is constantly changing as the user clicks and types during an application. New groups can be created and subviews can be added to (inserted into) and deleted from a group.

During its lifespan, a subview can be hidden or exposed by actions performed on other subviews, so the group needs to coordinate many activities.

Desktops

TDesktop is the normal startup background view, providing the familar user's desktop, usually surrounded by a menu bar and status line. Typically, **TApplication** will be the owner of a group containing the **TDesktop**, **TMenuBar**, and **TStatusLine** classes. Other views (such as windows and dialog boxes) are created, displayed, and manipulated in the desktop in response to user actions (mouse and keyboard events). Most of the actual work in an application goes on inside the desktop.

Programs

TProgram provides a set of virtual member functions for its derived class, **TApplication**. **TProgram** has two base classes (multiple inheritance): **TGroup** and **TProgInit**.

Applications

TApplication provides a program template class for your Turbo Vision application. It is derived from **TGroup** (via **TProgram**). Typically, it will own **TMenuBar**, **TDesktop**, and **TStatusLine** subviews. **TApplication** has member functions for creating and inserting these three subviews. The key member function of **TApplication** is **TApplication::run**, which executes the application's code.

Windows

TWindow objects, with help from associated **TFrame** objects, provide the popular bordered rectanglar displays that you can drag, resize, and hide using member functions inherited from **TView**. A data member called *frame* points to the window's **TFrame** object. A **TWindow** object can also zoom and close itself using its own member functions. **TWindow** handles the *Tab* and *Shift-Tab* key actions for selecting the next and previous selectable subviews in a window. **TWindow's** event handler takes care of close, zoom, and resize commands. Numbered windows can be selected with *Alt-n* hot keys.

Dialog boxes

TDialog, which is derived from **TWindow**, is used to create dialog boxes to handle a variety of user interactions. Dialog boxes typically contain controls such as buttons and check boxes. The parent's **execView** member function is used to save the previous context, insert a **TDialog** class into the group, and then make the dialog box modal. The **TDialog** class then handles user-generated events such as button clicks and keystrokes. The *Esc* key is treated specially as an exit (*cmCancel*). The *Enter* key is specially treated as a broadcast *cmDefault* event (usually meaning that the default button has been selected). Finally, **execView** restores the previously saved context.

Terminal views

Terminal views are all views that are not groups. That is, they cannot own other views. They are therefore the endpoints of any chains of views.

Frames

TFrame provides the displayable frame (border) for a **TWindow** class together with icons for moving and closing the window. **TFrame** objects are never used on their own, but always in conjunction with a **TWindow** object.

Buttons

A **TButton** object is a titled box used to generate a specific command event when "pushed." Buttons are usually placed inside (owned by) dialog boxes, offering such choices as "Ok" or "Cancel." The dialog box is usually the modal view when it appears, so it traps and handles all events, including its button events. **TButton's** event handler offers several ways of pushing a button: mouse-clicking in the button's rectangle, typing the hot key, or selecting the default button with the *Enter* key.

Clusters

TCluster is a seed class used to implement check boxes and radio buttons. A cluster is a set of controls that all respond in the same way. Cluster controls are often associated with **TLabel** objects, letting you select the control by selecting on the adjacent explanatory label. Additional data members are *value*, giving a user-defined value, and *sel*, indexing the selected control of the cluster. Member functions for drawing text-based icons and "mark" characters are provided. The cursor keys or mouse clicks can be used to mark controls in the cluster.

Radio buttons are special clusters in which only one control can be selected. Each subsequent selection deselects the current one (as with some radio station selectors). Check boxes are clusters in which any number of controls can be marked (selected).

Menus

TMenuView and its two derived classes, **TMenuBar** and **TMenuBox**, provide the basic classes for creating pull-down menus and submenus nested to any level. You supply text strings for the menu selections (with optional highlighted hot keys) together with the commands associated with each selection. The **handleEvent** member functions take care of the mechanics of mouse and/or keyboard (including shortcut and hot key) menu selection.

Menu selections are displayed using a **TMenuBar** class, usually owned by a **TApplication** class. Menu selections are displayed using objects of type **TMenuBox**.

For most applications, you will not be involved directly with menu classes. By overriding **TApplication::initMenuBar** with a suitable set of nested **new TMenuBar**, **new TSubMenu**, **new TMenuItem** and **newLine** calls, Turbo Vision builds, displays, and interacts with the required menus.

Histories

The seed class **THistory** implements a generic pick-list mechanism. Its two additional data members, *link* and *historyId*, give each **THistory** object an associated **TInputLine** and the ID of a list of previous entries in the input line. **THistory** works in conjunction with **THistoryWindow** and **THistoryViewer**.

Input lines

TInputLine is a specialized view that provides a basic input-line string editor. It handles all the usual keyboard entries and cursor movements (including *Home* and *End*). It offers deletes and inserts with selectable insert and overwrite modes and automatic cursor shape control. You can drag the mouse to highlight blocks of text.

List viewers

The **TListViewer** class is another seed class from which you can derive list viewers of various kinds, such as **TListBox**. **TListViewer**'s members let you display linked lists of strings with control over one or two scroll bars. The event handler permits mouse or key selection (with highlight) of items on the list. The **draw** member function copes with resizing and scrolling. **TListViewer** has an empty, virtual **getText** member function, so you need to supply the mechanism for creating and manipulating the text of the items to be displayed.

TListBox, derived from **TListViewer**, implements the most commonly used list boxes, namely those displaying lists of strings, such as file names. **TListBox** objects represent displayed lists of such items in one or more columns with an optional vertical scroll bar. The horizontal scroll bars of **TListViewer** are not supported. The inherited **TListViewer** member functions let you select (and highlight) items by mouse and keyboard cursor actions. **TListBox** has an additional member function called **list** that points to a **TCollection** class. This provides the items to be listed and selected. The contents of the collection are your responsibility, as are the actions to be performed when an item is selected.

Scrolling classes

A **TScroller** class is a scrollable view that serves as a portal onto another, larger "background" view. Scrolling occurs in response to keyboard input or actions in the associated **TScrollBar** objects. Scrollers have two data members, *hScrollbar* and *vScrollbar*, identifying their controlling horizontal and vertical scroll-bars. The *delta* data member in **TScroller** determines the unit amount of x and y scrolling in conjunction with data members in the associated scroll bars.

TScrollBar classes provide either vertical or horizontal control. The key data members are *value* (the position of the scroll bar indicator), *pgStep* (the amount of scrolling needed in response to mouse clicks and *PgUp*, *PgDn* keys) and *arStep* (the amount of scrolling needed in response to mouse clicks and arrow keys).

A scroller and its scroll bars are usually owned by a **TWindow** class leading to a complex set of events to be handled. For example, resizing the window must trigger appropriate redraws by the scroller. The values of the scroll bar must also be changed and redrawn.

Text devices

TTextDevice is a scrollable TTY-type text viewer/device driver. Apart from the data members and member functions inherited from **TScroller**, **TTextDevice** defines virtual functions for writing strings to the device. **TTextDevice** exists solely as a base type for deriving real terminal drivers. **TTextDevice** uses **TScroller**'s constructor and destructor.

TTerminal implements a "dumb" terminal with buffered string writes. The size of the buffer is determined at initialization.

Static text

TStaticText classes are simple views used to display fixed strings provided by the data member *text*. They ignore any events sent to them. The **TLabel** class adds the property that the view holding the text, known as a label, can be selected (highlighted) by mouse click, cursor key, or *Alt-letter* keys. The additional data member *link* associates the label with another view, usually a control view that handles all label events. Selecting the label selects the linked control and selecting the linked control highlights the label as well as the control.

Status lines

The **TStatusLine** class is intended for various status and hint (help) displays, usually at the bottom line of the screen. A status line is a one-character-high strip of any length up to the screen width. The class offers dynamic displays reacting with events as the application unfolds. Items on the status line can be mouse- or hot-key selected, rather like **TLabel** classes. Most application classes will start life owning a **TMenuBar** class, a **TDesktop** class, and a **TStatusLine** object. The added data members for **TStatusLine** provide an *items* pointer and a *defs* pointer.

The *items* data member points to the current linked list of **TStatusItem** objects. These hold the strings to be displayed, the hot key mappings, and the associated *command* word. The *defs* data member points to a linked list of **TStatusDef** objects used to determine the current help context, allowing you to display short hints. **TStatusLine** is instantiated and initialized using a suitably defined **TApplication::initStatusLine**.

Streams

A stream is a generalized object for handling input and output. In traditional device and file I/O, separate sets of functions must be devised to handle the extraction and conversion of different data types. With Turbo Vision streams, you can create polymorphic I/O functions, using the overloaded operators **<<** and **>>**, which know how to process their own particular stream contents.

Turbo Vision provides a general stream manager based on the class **TStreamableClass**. This lets you write and inject objects to streams and read and extract them back from streams in a type-safe manner. There are several functions that every streamable class defines. These functions are used by the stream manager to read and write objects of that class, and they must be linked into

any application that will be using the stream manager. Linkage is accomplished by calling the **__link** macro; for example,

```
__link(RClassName...);
```

This indicates your intention to read and write objects of type **T*ClassName*** to and from a stream. This process is known as *registering* a class for stream applications; we also refer to this as *stream registration*. Each registered class takes care of registering any other classes it needs, so dependencies are automatically figured out for you.

Object streams work with the standard C++ **streambuf** classes used for **iostream**s. This means that if you have an **iostream** variant that can read and write to a modem, you'll also be able to transmit objects across the modem.

Collections

TCollection implements a general set of items, including arbitrary objects of different types. Unlike the arrays, sets, and lists of non-OOP languages, a Turbo Vision collection allows for dynamic sizing. **TCollection** is a seed base for more specialized collections, such as **TSortedCollection**. The chief data member is *items*, a generic (**void****) pointer to a collection of items. Apart from the indexing, insertion, and deletion member functions, **TCollection** offers several iterator routines. A collection can be scanned for the first or last item that meets a condition specified in a user-supplied test function. With the **forEach** member function you can also trigger user-supplied actions on each item in the collection.

Note **TNSCollection** is the base class to **TCollection**, but you should be using **TCollection**, which is why we focus on it here. The same applies to **TNSSortedCollection** and **TSortedCollection**.

Sorted collections

TSortedCollection is an abstract class implementing collections that are sorted by keys. Sorting is defined via a pure virtual **compare** function. Your derived classes must therefore specify a particular ordering for collections of classes of any type. The **insert** member function adds items to maintain this ordering, and keys can be located quickly with a binary **search** member function. Duplicate keys can be accommodated if the Boolean data member *duplicates* is set to *True*.

String collections

TStringCollection is a simple extension of **TSortedCollection** for handling sorted collections of strings. The secret ingredient is the overriding of the **compare** member function to provide alphabetical ordering. A **freeItem** member function removes a given string item from the collection.

Resources

A resource file is a special kind of stream where generic classes ("items") can be indexed via string keys. Rather than derive resource files from **TStream**, **TResouceFile** has a data member, *stream*, associating a stream with the resource file. Resource items are accessed with **get**(*key*) calls, where *key* is the string index. Other member functions provided are **put** (store an item with a given key), **keyAt** (get the index to a given item), **flush** (write all changes to the stream), **remove** (erase the item at a given key), and **count** (return the number of items on file).

Resource collections

TResourceCollection implements a collection of sorted resource indexes used by resource files. The **TStringCollection** member functions **freeItem** and **keyOf** are all overriden to handle resources.

String lists

TStringList implements a special kind of string resource in which strings can be accessed via a numerical index using the **get** member function. **TStringList** simplifies internationalization and multilingual text applications. String lists can be read from a stream using the stream manager. To create and add to string lists, use **TStrListMaker**.

TStringList offers access only to existing numerically indexed string lists. **TStrListMaker** supplies the **put** member function for adding a string to a string list, and **<<** and **>>** operators for stream I/O.

C H A P T E R

4

Views

From reading Chapters 1 and 2 you should have a sense of what a Turbo Vision application looks like from the outside. But what's behind the scenes? That's the subject of Chapters 4 and 5.

"We have taken control of your TV..."

One of the adjustments you make when you use Turbo Vision is that you give up writing directly to the screen. Instead of using **stdlib** functions such as **printf** and **puts** to convey information to the user, you give the information to Turbo Vision, which makes sure the information appears in the right places at the right time.

The basic building block of a Turbo Vision application is the *view*. A view is an object that manages a rectangular area of the screen. For example, the menu bar at the top of the screen is a view. Any program action in that area of the screen (for example, clicking the mouse on the menu bar) will be dealt with by the view that controls that area.

Menus are views, as are windows, the status line, buttons, scroll bars, dialog boxes, and usually even a simple line of text. In general, anything that shows up on the screen of a Turbo Vision program *must* be a view, and the most important property of a view is that it knows how to represent itself on the screen. So, for example, when you want to make a menu system, you tell Turbo

Vision that you want to create a menu bar containing certain menus, and Turbo Vision handles the rest.

The most visible example of a view, but one you probably would not think of as a view, is the program itself. It controls the entire screen, but you don't notice that because the program sets up other views (its subviews) to handle its interactions with the user. As you will see, what appears to the user as a single object (like a window) is often a group of related views.

Simple view objects

As you can see from the hierarchy chart on page 74, all Turbo Vision views have **TObject** as an ancestor. **TObject** is little more than a common ancestor for all the objects. Turbo Vision itself really starts at **TView**.

A **TView** object itself just appears on the screen as a blank rectangle. There is little reason to instantiate a **TView** itself unless you want to create a blank rectangle on the screen for prototyping purposes. But even though **TView** is visually simple, it contains all of Turbo Vision's basic screen management member functions and data members.

There are two things any **TView**-derived object must be able to do:

1. Draw itself at any time. **TView** defines a virtual member function called **draw**, and each object derived from **TView** must also have a **draw** member function. This is important, because often a view will be covered or overlapped by another view, and when that other view goes away or moves, the view must be able to show the part of itself that was hidden.
2. Handle any events that come its way. As noted in Chapter 1, Turbo Vision programs are event-driven. This means that Turbo Vision gathers input from the user and parcels it out to the appropriate objects in the application. Views need to know what to do when events affect them. Event handling is covered in detail in Chapter 5.

Setting your sights

Before discussing what view objects *do*, you need to learn a bit about what they *are*—how they represent themselves on the screen.

TPoint is described in the next section.

The location of a view is determined by two points: its top left corner (called its origin) and its bottom right corner. Each of these points is represented in the class by a data member of the type **TPoint**. The *origin* data member is a **TPoint** indicating the origin of the view, and the *size* data member represents the lower right corner.

Note that *origin* is a point in the coordinate system of the owner view: If you open a window on the desk top, its *origin* data member indicates the *x*- and *y*-coordinates of the window relative to the origin of the desk top. The *size* data member, on the other hand, is a point relative to the origin of its own object. It tells you how far the lower right corner is from the origin point, but unless you know where the view's origin is located within another view, you can't tell where that corner really is.

Getting the TPoint

The **TPoint** type is *extremely* simple. It has only two data members, called *x* and *y*, which are its coordinates. It has two member functions, the overloaded operators **+=** and **-=**, allowing you to increment or decrement the *x*, *y* coordinates as follows:

```
TPoint& operator +=( const TPoint& adder );
TPoint& operator -=( const TPoint& subber );
```

So if *p1* and *p2* are points with coordinates (*x1*, *y1*) and (*x2*, *y2*), you can write expressions such as

```
p1 += p2;                // p1 now = (x1 + x2, y1 + y2)
p1 -= p2;                // p1 now = (x1 - x2, y1 - y2)
```

There are also four **TPoint** friend operators **–**, **+**, **==**, and **!=**, that work with two point arguments in the obvious way:

```
TPoint p1(1,2), p2(3,4), p3;
p3 = p1 + p2;            // p3 = (4,6)
p3 = p2 - p1;            // p3 = (2,2)
if (p1 == p2) { // points are equal...}
if (p3 != p1) { // points are not the same...}
```

To sum up, the **TPoint** class allows views to specify and manipulate their coordinates as a single data member.

Getting into a TRect

For convenience, **TPoint** objects are rarely dealt with directly in Turbo Vision. Since each view has both an origin and a size, they are usually handled together in a class called **TRect**. **TRect** has two data members, *a* and *b*, each of which is a **TPoint**. When specifying the boundaries of a view object, those boundaries are passed to the constructor in a **TRect**.

TRect and **TView** both provide useful member functions for manipulating the size of a view. For example, if you want to create a view that fits just inside a window, you can get the window to tell you how big it is, then shrink that size and assign it to the new inside view.

***ThisWindow** and **InsideView** are just made up for this example.*

```
void ThisWindow::makeInside()
{
   TRect r = getExtent();      // sets r to size of ThisWindow
   InsideView*inside;
   r.grow(-1, -1);       // shrinks the rectangle by 1, both ways
   inside = new InsideView(r);    // creates an inside view
   insert(inside);     // insert the new view into the window
// you could also insert(new InsideView(r)); if the pointer inside is
// not needed explicitly within this definition
}
```

getExtent is a **TView** member function that returns a **TRect** value equal to the coordinates of a rectangle covering the entire view. **grow** is a **TRect** member function that increases (or with negative parameters, decreases) the horizontal and vertical sizes of a rectangle.

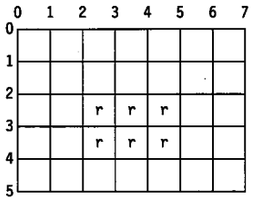
Turbo Vision coordinates

Turbo Vision's method for assigning coordinates may be different from what you're used to. The difference is that, unlike most coordinate systems that designate the character spaces on the screen, Turbo Vision coordinates specify the grid *between* the characters.

For example, if *r* is a **TRect** object, `r.assign(0,0,0,0)` designates a rectangle with no size—it is only a point. The smallest rectangle that can actually contain anything would be created with `r.assign(0,0,1,1)`.

Figure 4.1 shows a **TRect** created by `r.assign(2,2,4,5)`.

Figure 4.1
Turbo Vision coordinate system



Thus, `r.assign(2,2,4,5)` produces a rectangle that contains six character spaces. Although this coordinate system is slightly unconventional, it makes it *much* easier to calculate the sizes of rectangles, the coordinates of adjacent rectangles, and some other things as well.

Making an appearance

The appearance of a view object is determined by its **draw** member function. Nearly every class derived from **TView** will need to have its own **draw**, since it is usually the appearance of a view that distinguishes it from other views.

There are a couple of rules that apply to all views with respect to appearance. A view must

- cover the entire area for which it is responsible, and
- be able to draw itself at any time.

Both of these properties are very important and deserve some discussion.

Territoriality

There are good reasons for each view to take responsibility for its own territory. A view is assigned a rectangular region of the screen. If it does not fill in that whole area, the contents of the unfilled area are undefined: Just about anything could show up there, and you would have no control over it. The program TVGUID06.CPP in Chapter 2 demonstrated what happens if a view leaves some of its appearance to chance.

Drawing on demand

In addition, a view must *always* be able to represent itself on the screen. That's because other views may cover part of it but then be removed, or the view itself might move. In any case, when called upon to do so, a view must always know enough about its present state to show itself properly.

Note that this may mean that the view does nothing at all: It may be entirely covered, or it may not even be on the screen, or the window that holds it might have shrunk to the point that the view is not visible at all. Most of these situations are handled automatically, but it is important to remember that your view must always know how to draw itself.

This is different from a lot of other windowing schemes, where the writing on a window, for example, is persistent: You write it there and it stays, even if something covers it up then moves away. In Turbo Vision, you can't assume that a view you uncover is correct—after all, something may have told it to change while it was covered!

Putting on your best behavior

The behavior of a view is almost entirely determined by a member function called **handleEvent**. **handleEvent** is passed an event record, which it must process in one of two ways. It can either perform some action in response to the event and then mark the event as having been handled, or it can pass the event along to the next view (if any) that should see it.

Event handling is covered in detail in Chapter 5, "Event-driven programming."

The key to behavior is how the view responds to certain events. For example, if a window receives an event containing a *cmClose* command, the expected behavior is that the window would close. It is possible that you might devise some other response to that command, but not likely.

Complex views

You've already learned something about the most important immediate descendant of **TView**, the **TGroup**. **TGroup** and its derived classes are collectively referred to as groups. Views not derived from **TGroup** are called terminal views.

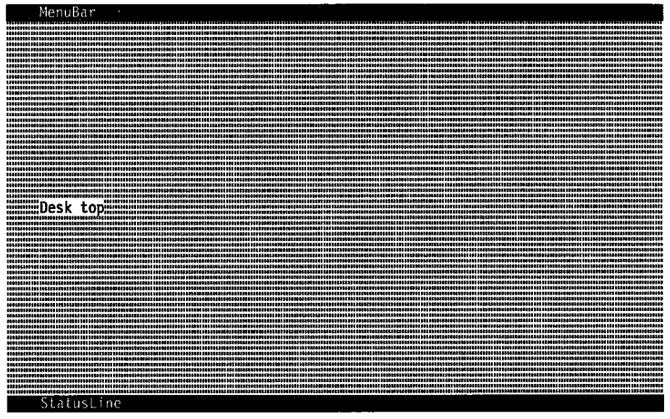
Basically a group is just an empty box that contains and manages other views. Technically, it is a view, and therefore responsible for all the things that any view must be able to do: manage a rectangular area of the screen, visually represent itself at any time, and handle events in its screen region. The difference is really in *how* it accomplishes these things: most of it is handled by *subviews*.

Groups and subviews

A subview is a view that is owned by another view. That is, some view (a group) has delegated part of its region on the screen to be handled by another view, called a subview, which it will manage.

An excellent example is **TApplication**. **TApplication** is a view that controls a region of the screen—the whole screen, in fact. **TApplication** is also a group that owns three subviews: the menu bar, the desk top, and the status line. The application delegates a region of the screen to each of these subviews. The menu bar gets the top line, the status line gets the bottom line, and the desk top gets all the lines in between. Figure 4.2 shows a typical **TApplication** screen.

Figure 4.2
TApplication screen layout



Notice that the application itself has no screen representation—you don't *see* the application. Its appearance is entirely determined by the views it owns.

Getting into a group

How does a subview get attached to a group? The process is called insertion. Subviews are created and then inserted into groups. In the previous example, the **TApplication** constructor creates three objects and inserts them into the application, as shown in the following pseudocode:

```
createStatusLine( cStatusLine ),
createMenuBar( cMenuBar ),
```

```
createDeskTop( cDeskTop )
if createDeskTop != 0  insert(deskTop);
if createStatusLine != 0  insert(statusLine);
if createMenuBar != 0  insert(menuBar);
```

Only when they have been inserted are the newly created views part of the group. In this particular case, **TApplication** has divided its region into three separate pieces and delegated one to each of its subviews. This makes the visual representation fairly straightforward, as the subviews do not overlap at all.

There is no reason, however, why views should not overlap. Indeed, one of the big advantages of a windowed environment is the ability to have multiple, overlapping windows on the desk top. Luckily, groups (including the desk top) know how to handle overlapping subviews.

Groups keep track of the order in which subviews are inserted. That order is referred to as *Z-order*. Z-order determines the order in which subviews get drawn and the order in which events get passed to them.

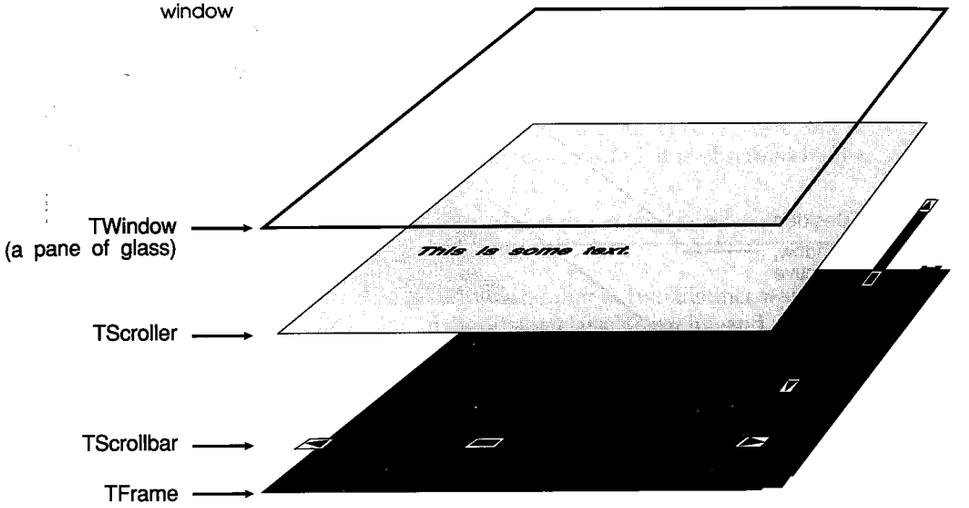
Another angle on Z-order

The term Z-order refers to the fact that subviews have a three-dimensional spatial relationship. As you've already seen, every view has a position and size within the plane of the view as you see it (the *x* and *y* dimensions), determined by its *origin* and *size* data members. But views and subviews can overlap, and in order for Turbo Vision to know which view is in front of which others, we have to add a third dimension, the Z-dimension.

Z-order, then, refers to the order in which you encounter views, starting closest to you and moving back "into" the screen. The last view inserted is the "front" view.

Rather than thinking of the screen as a flat plane with things written on it, think of it as a pane of glass providing a portal onto a three-dimensional world of views. Indeed, every group can be thought of as a "sandwich" of views, as illustrated in Figure 4.3.

Figure 4.3
Side view of a text viewer window



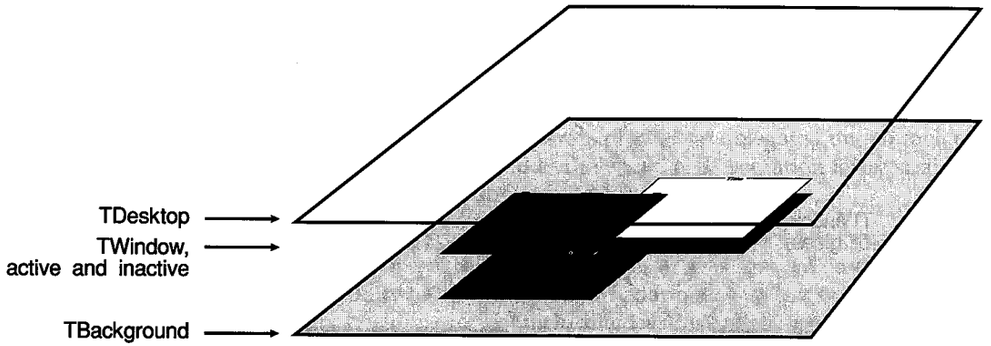
The window itself is just a pane of glass covering a group of views. Since all you see is a projection of the views behind the glass on the screen, you can't see which views are in front of others unless they overlap.

By default, a window has a frame, which is inserted before any other subviews. It is therefore the "background" view. In creating a scrolling interior, two scroll bars get overlaid on the frame. To you, in front of the whole scene, they look like part of the frame, but from the side, you can see that they actually float "above" the frame, obscuring part of the frame from view.

Finally, the scroller itself gets inserted, covering the entire area inside the border of the frame. Text is written on the scroller, not on the window, but you can see it when you look through the window.

On a larger scale, you can see the desk top as just a larger pane of glass, covering a larger sandwich, many of the contents of which are also smaller sandwiches, as shown in Figure 4.4.

Figure 4.4
Side view of the desk top



Again, the group (this time the desk top) is a pane of glass. Its first subview is a **TBackground** object, so that view is "behind" all the others. This view also shows two windows with scrolling interior views on the desk top.

Group portraits

A group is an exception to the rule that views must know how to draw themselves, because a group does not draw itself *per se*. Rather, a **TGroup** asks its subviews to draw themselves.

The subviews are called upon to draw themselves in Z-order: The first (most rear) subview inserted into the group is the first one drawn. That way, if subviews overlap, the one most recently inserted will be in front of any others.

The subviews owned by a group must cooperate to cover the entire region controlled by the group. A dialog box, for example, is a group, and its subviews—frame, interior, controls, and static text—must combine to fully "cover" the full area of the dialog box view. Otherwise, "holes" in the dialog box would appear, with unpredictable results.

When the subviews of a group draw themselves, their drawing is automatically clipped at the borders of the group. Because subviews are clipped, when you initialize a view and give it to a group, the view needs to reside at least partially within the group's boundaries. (You can grab a window and move it off the desk top until only one corner remains visible, for example, but something must remain visible for the view to be useful.) Only the part of a subview that is within the bounds of its owner group will be visible.

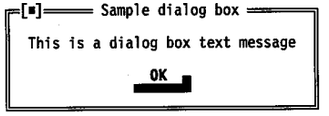
You may wonder where the desk top gets its visible background if it is a **TGroup** object. At its initialization, the desk top creates and owns a subview of type **TBackGround**, whose sole purpose is to draw a uniform background for the whole screen. Since the background is the first subview inserted, it is obscured by the other views drawn in front of it.

Relationships between views

Views are related to each other in two distinct ways: They are objects in the Turbo Vision class hierarchy, and they are members of the view tree. The distinction is of vital importance.

For example, consider the simple dialog box in Figure 4.5. It has a frame, a one-line text message, and a single button that closes the dialog box. In view tree terms, you have a **TDialog** view that owns a **TFrame**, a **TStaticText**, and a **TButton**.

Figure 4.5
A simple dialog box



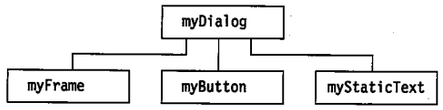
The class hierarchy

One way views are related is as base and derived class in the class hierarchy. Notice in the hierarchy diagram on page 75 that **TButton** is derived from **TView**. **TButton** actually *is* a **TView**, but it has additional members that make it a button. **TDialog** is also derived from **TView** (through **TGroup** and **TWindow**), so it has much in common with **TButton**. The two are distant "cousins" in the Turbo Vision hierarchy.

Ownership

The other way that views are related is in a view tree. In the view tree diagram (Figure 4.6), the **TDialog** object called **myDialog** owns the **TButton** object called **myButton**. Here the relationship is not between classes (**TDialog** is *not* a base class for **TButton**), but between objects (instances of classes), between owner and subview.

Figure 4.6
A simple dialog box's view tree



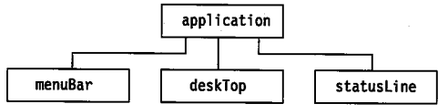
As you program, you'll need to make the **myButton** object interact with its owner in the view tree (the **myDialog** object). At the same time, **myButton** will draw upon attributes inherited from its ultimate ancestor (**TView**). Don't confuse the two relationships of ownership and inheritance.

A running Turbo Vision application looks like a tree, with views instantiating and owning other views. As your Turbo Vision application opens and closes windows, the view tree grows and shrinks as objects are inserted and removed. Of course, the class hierarchy only grows when you derive new classes from the standard class.

Subviews and view trees

As noted earlier, the **TApplication** group-view owns and manages the three subviews that it creates and inserts. You can think of this relationship as forming a view tree. Four static members of **TApplication** called *application*, *menuBar*, *deskTop*, and *statusLine* point to these objects. (For brevity, we often refer to *application* as though it were the main program object—strictly speaking the object is **application*.) You can picture *application* as the trunk, and *menuBar*, *deskTop*, and *statusLine* as the branches, as shown in Figure 4.7. Recall that there is only one instance of a static data member for all objects of its class.

Figure 4.7
Basic Turbo Vision view tree



Remember, the relationship illustrated in Figure 4.7 is *not* a class hierarchy, but a model of a data structure. The links indicate *ownership*, not inheritance.

Owners and subviews

In a typical application, as users click with the mouse or type via the keyboard, they create more views. These views will normally appear on the desk top, and so form further branches of the tree.

It is important to understand these relationships between owners and subviews, as both the appearance and the behavior of a view depend a great deal on who owns the view.

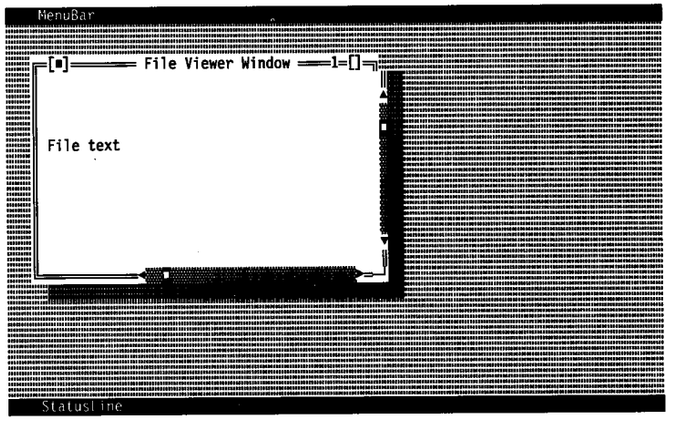
Let's follow the process. Suppose that the user clicks on a menu selection that calls for a file viewer window. The file viewer window will be a view. Turbo Vision will create the window and attach it to the desk top.

This same set of objects is depicted somewhat differently in Figure 4.3.

A window will most likely own a number of subviews: a **TFrame** object (the frame around the window), a **TScroller** object (the interior view that holds a scrollable array of text), and a couple of **TScrollbar** objects. When the window is called into being, it creates, inserts, owns, and manages its subviews.

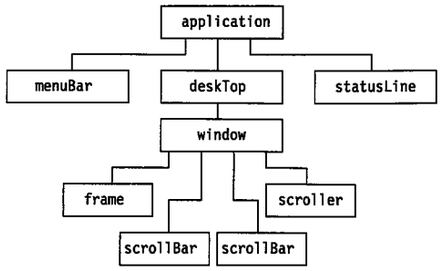
More views are now attached to our growing application, which now looks something like Figure 4.8.

Figure 4.8
Desk top with file viewer added



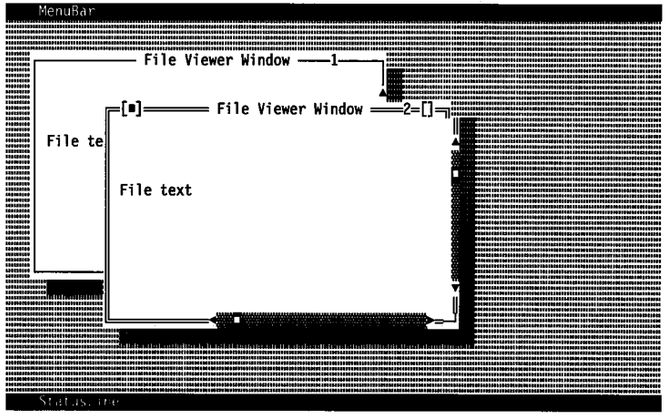
The view tree has also become somewhat more complex, as shown in Figure 4.9. (Again, these are ownership links.)

Figure 4.9
View tree with file viewer added



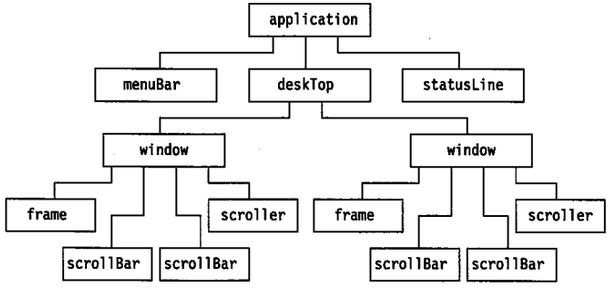
Now suppose the user clicks on the same menu selection and creates another file viewer window. Turbo Vision creates a second window and attaches it to the desk top, as shown in Figure 4.10.

Figure 4.10
Desk top with file viewer added



The view tree also becomes correspondingly more complex, as shown in Figure 4.11. (Again, these are ownership links.)

Figure 4.11
View tree with two file viewers added



As you'll see in Chapter 5, program control flows down this view tree. In the preceding example, suppose you click on a scroll bar in the file viewer window. How does that click arrive at the right place?

The application object (your program) sees the mouse click, realizes that it's within the area controlled by the desk top, and passes it to the desk top object. The desk top in turn sees that the click is within the area controlled by the file viewer, and passes it off to that view. The file viewer now sees that the click was on the scroll bar, and lets the scroll bar view handle the click, generating an appropriate response.

Event routing is explained in Chapter 5.

The actual mechanism for this is unimportant at this point. The important thing to remember is how views are connected. No matter how complex the structure becomes, all views are ultimately connected to your application object.

If the user clicks on the second file viewer's close icon or on a Close Window menu item, the second file viewer closes. Turbo Vision then takes it off the view tree and disposes it. The window destroys all of its subviews, then is destroyed itself.

Eventually, the user will trim the views down to just the original four, and will indicate at some point that he is finished by pressing *Alt-X* or by selecting Exit from a menu. The **TApplication** object will destroy its three subviews, then destroy itself.

Selected and focused views

Within each group of views, one and only one subview is selected at any given moment. For example, immediately following the creation and insertion of the menu bar, desk top, and status line, the desk top is the selected view, because that is where further work will take place.

When you have several windows open on the desk top, the selected window is the one in which you're currently working. This is also called the active window (typically the topmost window).

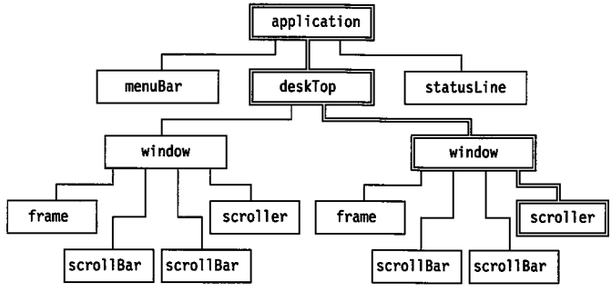
The focused view is the end of the chain of selected views that starts at the application.

Within the active window, the selected subview is called the focused view. You can think of the focused view as being the one you're looking at, or the one where action will take place. In an editor window, the focused view would be the interior view with

the text and active cursor in it. In a dialog box, the focused view is the highlighted control.

In the application diagrammed in Figure 4.11, *application* is the modal view, and *desktop* is its selected view. Within the desk top, the second (more recently inserted) window is selected, and therefore active. Within that window, the scrolling interior is selected, and because it is a terminal view (that is, it's not a group), it is the end of the chain, the focused view. Figure 4.12 depicts ownership within the same view tree with the chain of focused views highlighted by double-lined boxes.

Figure 4.12
The focus chain



Among other things, knowing which view is focused tells you which view gets information from the keyboard. For more information, see the section on focused events in Chapter 5, "Event-driven programming."

Finding the focused view

On monochrome displays, Turbo Vision adds arrow characters to indicate the focus.

The currently focused view is usually highlighted in some way on the screen. For example, if you have several windows open on the desk top, the active window is the one with the double-lined frame; the others' frames will be single-lined. Within a dialog box, the focused control (controls are views, too!) is brighter than the others, indicating that it is the one that will be acted upon if you press *Enter*. The focused control is therefore the default control as well.

How does a view get the focus?

A view can get the focus in two ways, either by default when it is created, or through some action by the user.

When a group of views gets created, the owner view specifies which of its subviews is to be focused by calling that subview's **select** member function. This establishes the *default focus*.

The user may wish to change which view currently has the focus. A common way to do this is to click the mouse on a different view. For instance, if you have several windows open on the desk top, you can select different ones simply by clicking on them. In a dialog box, you can move the focus among views by pressing *Tab*, which cycles through all the available views, or by clicking the mouse on a particular view, or by pressing a hot key.

Note that there are some views that are not selectable, including the background of the desk top, frames of windows, and scroll bars. When you create a view, you may designate whether that view is selectable, after which the view will determine whether it lets itself be selected. If you click on the frame of a window, for example, the frame does not get the focus, because **TFrame** objects are not selectable. In other words, frames know that they cannot be the focused view.

The focus chain

See Chapter 5, "Event-driven programming," for a full explanation.

If you start with the main application and trace to its selected subview, and continue following to each subsequent selected subview, you will eventually end up at the focused view. This chain of views from the **TApplication** object to the focused view is called the *focus chain*. The focus chain is used for routing focused events, such as keystrokes.

Modal views

A *mode* is a way of acting or functioning. A program may have a number of modes of operation, usually distinguished by different control functions or different areas of control. The Borland IDE (integrated development environment), for example, has an editing and debugging mode, a compile mode, and a run mode. Depending on which of these modes is active, keys on the keyboard may have different effects (or no effect at all).

A Turbo Vision view may define a mode of operation, in which case it is called a *modal view*. The classic example of a modal view is a dialog box. Usually, when a dialog box is active, nothing outside it functions. You can't use the menus or other controls not

owned by the dialog box. In addition, clicking the mouse outside the dialog box has no effect. The dialog box has control of your program until closed. (Some dialog boxes can be non-modal, but these are rare exceptions.)

When you instantiate a view and make it modal, only that view and its subviews can interact with the user. You can think of a modal view as defining the "scope" of a portion of your program. A modal view determines what behaviors are valid within it—events are handled only by the modal view and its subviews. Any part of the view tree that is not the modal view or owned by the modal view is inactive.

The status line is always "hot," no matter what view is modal.

There is actually one exception to this rule, and that is the status line. Turbo Vision "cheats" a little, and keeps the status line available at all times. That way you can have active status line items, even when your program is executing a modal dialog box that does not own the status line. Events and commands generated by the status line, however, will be handled as if they were generated within the modal view.

There is *always* a modal view when a Turbo Vision application is running. When you start the program, and often for the duration of the program, the modal view is the application itself, the **TApplication** object at the top of the view tree.

Modifying default behavior

Up to this point, you have seen mostly the default behavior of the standard views. But sometimes you will want to make your views look or act a little different, and Turbo Vision provides for that. This section explains the ways you can modify the standard views.

Every Turbo Vision view has four flags, bitmapped data members that you can set in various ways in order to control the behavior of the view. Three of these flags are covered here: **ushort** *options* (16 bits), **uchar** *growMode* (8 bits), and **uchar** *dragMode* (8 bits). The fourth flag, **ushort** *eventMask*, is covered in Chapter 5, "Event-driven programming."

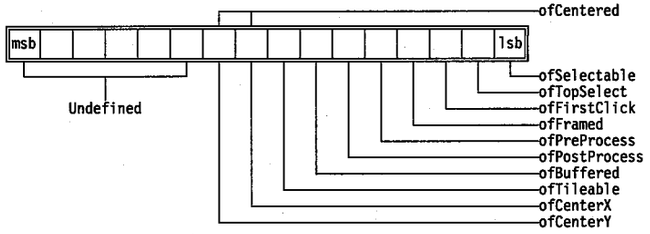
There is also a **ushort** *state* member (16 bits) that contains information about the current state of the view. Unlike the others, *state* is essentially "read only." Its value should only be changed by the **setState** member function. For more details, see page 113.

The options flag

options is a bitmapped flag used in every view. Various classes derived from **TView** have different bits set in *options* to determine their default behavior.

The *options* bits are defined mnemonically in Figure 4.13; explanations of the possible *options* follow.

Figure 4.13
options bit flags



ofSelectable If this bit is set (to 1), the user can select the view with the mouse. If the view is in a group, the user can select it with the mouse or *Tab* key. If you put a purely informational view on the screen, you might not want the user to be able to select it. Static text objects and window frames, for example, are usually not selectable.

ofTopSelect If set, the view is moved to the top of the owner's subviews if the view is selected. This option is designed primarily for windows on the desk top. You shouldn't use it for views in a group.

ofFirstClick If set, the mouse click event that selects the view is sent on to the view. If a button is clicked, you definitely want the process of selecting the button and operating it to happen with one click, so a button has *ofFirstClick* set. But if someone clicks on a window, you may or may not want the window to respond to the selecting mouse click other than by selecting itself.

ofFramed If set, the view has a visible frame around it. This is useful if you create multiple "panes" within a window, for example.

ofPreProcess If set, allows the view to process focused events before the focused view sees them. See page 129 in Chapter 5 for more details.

ofPostProcess If set, allows the view to handle focused events after they have been seen by the focused view, assuming the focused view has not cleared the event. See page 129 in Chapter 5 for more details.

ofBuffered When this bit is set, groups can speed their output to the screen. When a group is first asked to draw itself, it automatically stores the image of itself in a buffer if this bit is set and if enough memory is available. The next time the group is asked to draw itself, it copies the buffered image to the screen instead of asking all its subviews to draw themselves. If **new** or some other memory request exhausts available memory, Turbo Vision's memory manager will begin disposing of these group buffers until the memory request can be satisfied.

If a group has a buffer, a call to **TView::lock** stops all writes of the group to the screen until **TView::unlock** is called. When **unlock** is called, the group's buffer is written to the screen. Locking can decrease flicker during complicated updates to the screen. For example, the desk top locks itself when it is tiling or cascading its subviews.

ofTileable If set, the desk top can tile or cascade the windows that are currently open. If you don't want a window to be tiled, you must clear this bit. Such windows will then stay in the same position, while the tileable windows will be automatically tiled.

Tiling or cascading views from **TApplication::handleEvent** is simple:

An example of this can be found in FVIEWER.CPP, included in your distribution diskettes.

```
switch( event.message.command )
{
    ...
    case cmTile:
        tile();
        break;
    case cmCascade:
        cascade();
        break;
    ...
}
clearEvent( event );
```

If there are too many views to be successfully cascaded, the desk top will do nothing.

ofCenterX If set, the view is centered horizontally (in the *x* direction) when inserted in the group.

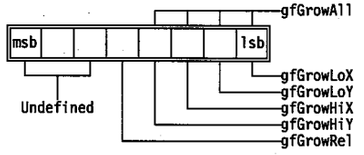
ofCenterY If set, the view is centered vertically (in the *y* direction) when inserted in the group. You may find this an important step in making a window work well with both 25- and 43-line text modes.

ofCentered If set, the view is centered both horizontally and vertically when it is inserted in the group.

The growMode flag

A view's *growMode* data member determines how the view will change when its owner group is resized. The *growMode* bits are defined as follows:

Figure 4.14
growMode bit flags



gfGrowLoX If set, the left side of the view maintains a constant distance from its owner's left side.

gfGrowLoY If set, the top of the view maintains a constant distance from the top of its owner.

gfGrowHiX If set, the right side of the view maintains a constant distance from its owner's right side.

gfGrowHiY — If set, the bottom of the view maintains a constant distance from the bottom of its owner.

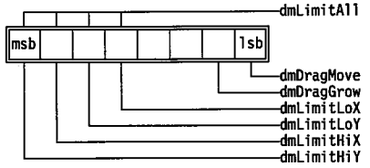
gfGrowAll — If set, the view will always remain the same size, and moves with the lower right corner of the owner.

gfGrowRel — If set, the view maintains its size relative to the owner's size. You should only use this option with **TWindow** objects (or objects of classes derived from **TWindow**) that are attached to the desk top. The window maintains its relative size when the user switches the application between 25- and 43/50-line mode. This flag isn't intended for use with views within a window.

The dragMode flag byte

A view's *dragMode* data member determines how the view behaves when it is dragged. The *dragMode* bits are defined as follows:

Figure 4.15
dragMode bit flags



dmDragMove — If this bit is set, when you click on the top of a window's frame, you can drag it.

dmDragGrow — When this bit is set, the view can grow.

dmLimitLoX — If set, the left side of the view cannot go out of the owner view.

dmLimitLoY — If set, the top of the view is not allowed to go out of the owner view.

dmLimitHiX — If set, the right side of the view cannot go out of the owner view.

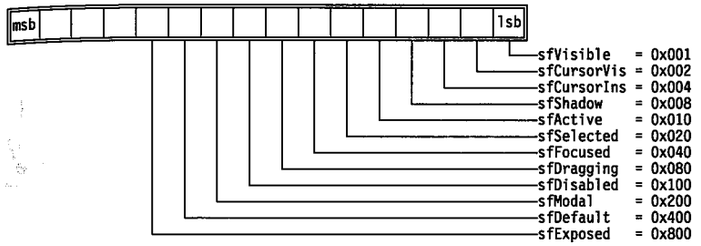
dmLimitHiY — If set, the bottom of the view cannot go out of the owner view.

dmLimitAll — If set, no part of the view can go out of the owner view.

state flag and setState

A view also has a bitmapped flag called *state*, which keeps track of various aspects of the view, such as whether it is visible, disabled, or being dragged. The *state* flag bits are defined in Figure 4.16.

Figure 4.16: *state* flag bit mapping



The meaning of each of the state flags (which should be clear from the mnemonics) is covered in detailed Chapter 16, "Global reference," under "*sfXXXX* state flag constants." This section focuses on the mechanics of manipulating the *state* data member.

Turbo Vision changes a view's *state* flag through a **setState** virtual member function. If the view gets the focus, gives up the focus, or becomes selected, Turbo Vision calls **setState**. This differs from the way the other bitmapped flags are handled, because those are usually set on initialization and remain unchanged (if a window is resizable, it is *always* resizable, for example). The *state* of a view, however, will often change during the time it is on the screen. Because of this, Turbo Vision provides a mechanism in **setState** that allows you not only to change the state of a view, but also to react to those changes in state.

setState takes two arguments: a state (*aState*) and a Boolean flag (*enable*) indicating whether the state is being set or cleared. If *enable* is *True* (nonzero), the bits in *aState* are set in *state*. If *enable* is *False* (zero), the corresponding *state* bits are cleared. That much is essentially what you would do with any bitmapped data member. The difference comes when you want a view to *do* something when you change its state.

Acting on a state change

Views often take some action when **setState** is called, depending on the resulting state flags. A button, for example, watches *state* and changes its color to cyan when it gets the focus. Here's a typical **setState** for a descendant of **TView**:

```
virtual void TButton::setState(ushort aState, Boolean enable)
{
  TView::setState(aState, enable);      // set or clear state bits
  if (aState & (sfSelected | sfActive))
      drawView();
  if (aState & sfFocused)
      makeDefault(enable);
};
```

Note the call **TView::setState** in the body of **TButton::setState**. You should always call **TView::setState** from within an overriding **setState** member function. **TView::setState** does the actual setting or clearing of the state flags. You then define any special actions based on the resulting state of the view. **TButton::setState** checks to see if it is in an active or selected window in order to decide whether to draw itself. It also checks to see if it has the focus, in which case it calls its **makeDefault** member function, which grabs or releases the focus, depending on the *enable* parameter.

If you need to make changes in the view or the application when the state of a particular view changes, you should do it by overriding the view's **setState**.

You and Turbo Vision often cooperate when the state changes. Suppose, for instance, that you want a block cursor to appear in your text editor when the editor's insert mode is toggled on.

First, the editor insert mode will have been bound to a keystroke—say, the *Ins* key. When the text editor is the focused view and the *Ins* key is pressed, the text editor receives the *Ins* key event. The text editor's **handleEvent** responds to the *Ins* event by calling the **toggleInsMode** member function. Turbo Vision does the rest. **toggleInsMode** calls the view's **setState** to reverse the *sfCursorIns* state:

```
void TEditor::toggleInsMode
{
    overwrite = Boolean(!overwrite);
    setState( sfCursorIns, Boolean(!getState(sfCursorIns)) );
}
```


The **getState** member function returns *True* if the command argument is set in *state*. The *sfCursorIns* bit in *state* determines the shape of the cursor: solid block for insert mode, normal for overwrite mode. Command sets, such as *state*, are easily manipulated using objects of type **TCommandSet**, which has several overloaded operators. For example,

```
TCommandSet s, command;
...
s += command;
```

adds the set *command* to the set *s*. Similarly, **-=** removes commands from a command set. We also describe such actions as *enabling* or *disabling* commands. The operators **==** and **!=** can test two command sets for equality.

What color is your view?

No one ever seems to agree on what colors are "best" for any computer screen. Because of this, Turbo Vision allows you to change the colors of the views you put on the screen. In order to facilitate this, Turbo Vision provides you with the **TPalette** class for handling color palettes.

Color palettes

Palettes for all standard views are listed in Chapter 13, "Class reference."

When a Turbo Vision view draws itself, it asks to be drawn, not with a specific color, but with a color indicated by a position in its palette. For example, the palette for **TScroller** looks like this:

```
#define cpScroller "\x06\x07"
```

Color palettes are **TPalette** objects. You will not normally be concerned with the internal structure of palettes, since the member functions provided attend to all the housekeeping. You can look on *cpScroller* as a two-character string, providing two palette entries. The layout of the **TScroller** palette is defined as

```
// Palette layout
// 1 = Normal
// 2 = Highlight
```

but it might be more useful to look at it this way:

Figure 4.17
TScroller's default color palette

getColor is a TView member function.

This means there are two kinds of text a scroller object knows how to display: normal and highlighted. The default color of each is determined by the palette entries. When displaying normal text, the **draw** member function needs to call **getColor(1)**, meaning it wants the color indicated by the first palette entry. To show highlighted text, the call would be **getColor(2)**.

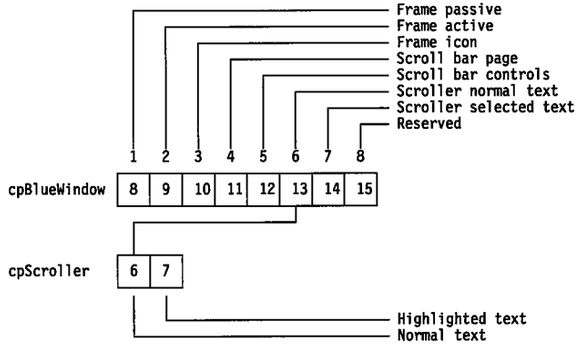
Selecting non-default colors is described in the next section.

If all you want to do is display the default colors, that's really all you need to know. The palettes are set up so that any reasonable combination of objects should produce decent looking colors.

Inside color palettes

Palette entries are actually indexes into their owner's palette, not the colors themselves. If a scroller is inserted into a window, you get normal text by calling for the normal text color in the scroller's palette, which contains the number 6. To translate that into a color, you find the sixth entry in the owner's palette. Figure 4.18 shows **TWindow**'s palette.

Figure 4.18
Mapping a scroller's palette onto a window



The sixth entry in **TWindow**'s palette is 13, which is an index into the palette of the window's owner (the desk top), which in turn indexes into the palette of its owner, the application. **TDeskTop** has a null palette, meaning that it doesn't change anything—you can think of it as a "straight" or "transparent" palette, with the first entry being the number 1, the second being 2, and so on.

The application does have a palette, a large one containing 64 entries for all the elements you might insert into a Turbo Vision application. Its thirteenth element is 0x1E. The application is the end of the line (it has no owner), so the palette mapping stops there.

*Don't think of palettes as **colors**. They are **kinds of things to display**.*

So now you are left with 0x1E, which is a text attribute byte corresponding to background color 1 and foreground color 0xE (or 14), which produces yellow characters on a blue background. Again, don't think of this in terms of yellow-on-blue, but rather say that you want your text displayed as the normal color for window text.

The getColor member function

Color palette mapping is done by the virtual member function **TView::getColor**. **getColor** climbs up the view tree from the object being drawn to its owner, to the owner's owner, and so on, until it gets to the application object. At each object along that chain, **getColor** calls **getPalette** for that object. The end result is a color attribute.

A view's palette contains offsets into its owner's palette, except for the application, whose palette contains color attributes.

Overriding the default colors

The obvious way to change colors is to change the palette. If you don't like your scroller's normal text color, your first instinct might be to change entry 1 (the normal text entry) in the scroller's palette, perhaps from 6 to 5. Normal scroller text is then mapped onto the window entry for scroll bar controls (blue on cyan, by default). Remember: *5 is not a color!* All you've done is tell the scroller that its normal text should look like the scroll bars around it!

So what if you don't want bright yellow on blue? Change the palette entry for normal window text in **TApplication**. Since that is the last non-null palette, the entries in the application palette determine the colors that will appear in all views within a window. Make this your color mantra: Colors are not absolute, but are determined by the owner's palettes.

This makes sense: Presumably you want your windows to look similar. You certainly don't want to have to tell every single window what color it should be. If you change your mind later (or

you allow users to customize colors) you would have to change the entries for *each* window.

Also, a scroller or other interior does not have to worry about its colors if it is inserted into some window other than the one you originally intended. If you put a scroller into a dialog box instead of a window, for example, it will not (by default) come up in the same colors, but rather in the colors of normal text in a dialog box.

To change a view's palette, override the virtual member function **TView::getPalette**. To create a new scroller class that draws itself in the window's frame color instead of the normal text color, the declaration and implementation of the class would include the following:

```
class TMyScroller : public TScroller
{
    ...
    virtual TPalette& getPalette() const;
    ...
};

virtual TPalette& TMyScoller::getPalette() const
{
    static TPalette palette( cpFrame, sizeof( cpFrame ) - 1 );
    return palette;
}
```

This **TPalette** constructor takes two arguments: a palette string and its length (less one for the first length byte), and returns the new palette object. The default **getPalette** for each view class simply returns the standard palette object assigned to that class.

Adding new colors

You may want to add colors to the window class, which will allow for a variety of colors to be used for new views you create. For example, you might decide you want a third color in your scroller for a different type of highlight, such as the one used for the breakpoints in the IDE editor. This can be done by deriving a new class from the existing **TWindow**, and adding to the default palette, as shown here:

```
#define cpNewPalette cpBlueWindow "\x54"

class TMyWindow : public TWindow
{
    ...
    virtual TPalette& getPalette() const;
    ...
}

virtual TPalette& TMyWindow::getPalette() const
{
    static TPalette palette (cpNewPalette, sizeof( cpNewPalette) -
1);
    return palette;
}
```

Now **TMyWindow** has a new palette entry that contains this new type of highlight. *cpBlueWindow* is a string constant containing **TWindow**'s default palette:

```
#define cpBlueWindow "\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F"
```

You will have to change **MyScroller::getPalette** to take advantage of the new window palette:

```
virtual TPalette& TMyScroller::getPalette() const
{
    const char *cpMyScroller = "\x06\x07\x09";
    static TPalette palette( cpMyScroller, sizeof(cpMyScroller)-1 );
    return palette;
}
```

The scroller's palette entry 3 is now the new highlight color (in this case bright white on red). If you use this new **getPalette** using the *cpMyScroller* that accesses the ninth element in its owner's palette, be sure that the owner is indeed using the *cpMyWindow* palette. If you try to access the ninth element in an eight-element palette, the results are undefined.

CHAPTER 5

Event-driven programming

The purpose of Turbo Vision is to provide you with a working framework for your applications so you can focus on creating the "meat" of your applications. The two major Turbo Vision tools are built-in windowing support and the handling of events. Chapter 4 explained views; this chapter deals with how to build your programs around events.

Bringing Turbo Vision to life

We have already described Turbo Vision applications as being event-driven, and briefly defined events as being occurrences to which your application must respond.

Reading the user's input

In a traditional procedural programs, you typically write a loop of code that reads the user's keyboard, mouse, and other input, and you make decisions based on that input within the loop. You'll call functions, or branch to a code loop somewhere else that again begins reading the user's input:

```
quit = False;
do
{
  B = ReadKey();
  switch( B )
  {
    case 'i':
    case 'I':
        invertArray();
        break;
    case 'e':
    case 'E':
        editArray();
        break;
    case 'g':
    case 'G':
        graphicDisplay();
        break;
    case 'q':
    case 'Q':
        quit = True;
        break;
    default:
        break;
   }
} while (!quit);
```

An event-driven program is not really structured very differently from this. In fact, it is hard to imagine an interactive program that doesn't work this way. However, an event-driven program looks different to you, the programmer.

In a Turbo Vision application, you no longer have to read the user's input because Turbo Vision does it for you. It packages the input into *events* (objects of type **struct TEvent**), and dispatches them to the appropriate views in the program. That means your code only needs to know how to deal with relevant input, rather than sorting through the input stream looking for things to handle.

For instance, if the user clicks on an inactive window, Turbo Vision reads the mouse action, packages it into an event record, and sends the event record to the inactive window.

If you come from a traditional programming background, you might be thinking at this point, "Okay, so I don't need to read the user's input anymore. What I'll be doing instead is learning how to read a mouse click event, and how to tell an inactive window to become active." In fact, there's no need for you to write even that much code.

Views can handle much of a user's input by themselves. A window knows how to open, close, move, be selected, resize, and more. A menu knows how to open, interact with the user, and close. Buttons know how to be pushed, how to interact with each other, and how to change color. Scroll bars know how to be operated. The inactive window can make itself active without any attention from you.

So what is your job as programmer? You will define new views with new actions, which will need to know about certain kinds of events that you'll define. You'll also teach your views to respond to standard commands, and even to generate their own commands ("messages") to other views. The mechanism is already in place: All you have to do is generate commands and teach views what to do when they see them.

But what exactly do events look like to your program, and how does Turbo Vision handle them for you?

The nature of events

Events can best be thought of as little packets of information describing discrete occurrences to which your application needs to respond. Each keystroke, each mouse action, and certain conditions generated by other components of the program, constitute separate events. Events cannot be broken down into smaller pieces; thus, the user typing in a word is not a single event, but a series of individual keystroke events.

At the core of every **TEvent** object is a **ushort** data member named *what*. The numeric value of *what* describes the kind of event that occurred, and the remainder of the event structure holds specific information about that event: the keyboard scan code for a keystroke event, information about the position of the mouse and the state of its buttons for a mouse event, and so on.

Because different kinds of events get routed to their destination objects in different ways, we need to look first at the kinds of events recognized by Turbo Vision.

Kinds of events

Let's look more closely at the possible values of **TEvent::what**. There are basically four types of event: mouse events, keyboard events, message events, and "nothing" events. Each type has a mask defined, so your objects can determine quickly which general type of event occurred without worrying about what specific sort it was. For instance, rather than checking for each of the four different kinds of mouse events, you can simply check to see if the event flag is in the mask. Instead of

```
TEvent event;
 ...
if (event.what & (evMouseDown | evMouseUp | evMouseMove |
                  evMouseAuto)) != 0 { ... }
```

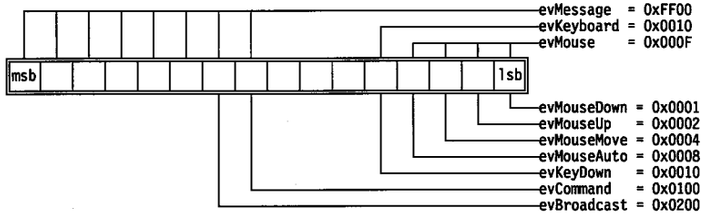
you can use

```
if (event.what & evMouse) != 0  { ...
```

The masks available for separating events are *evNothing* (for "nothing" events), *evMouse* for mouse events, *evKeyboard* for keyboard events, and *evMessage* for messages.

The event mask bits are defined in Figure 5.1.

Figure 5.1
TEvent.what data member bit mapping



Mouse events

There are basically four kinds of mouse events: an up or down click with either button, a change of position, or an "auto" mouse event. Pressing down a mouse button results in an *evMouseDown* event. Lifting (releasing) the button generates an *evMouseUp* event. Moving the mouse produces an *evMouseMove* event. And if you hold down the button, Turbo Vision will periodically generate an *evMouseAuto* event, allowing your application to perform such actions as repeated scrolling. All mouse event records include the position of the mouse, so an object that processes the event knows where the mouse was when it happened.

Keyboard events

Keyboard events are even simpler. When you press a key, Turbo Vision generates an *evKeyDown* event, which keeps track of which key was pressed.

Message events

Message events come in three flavors: commands, broadcasts, and user messages. The difference is in how they are handled, which is explained later. Basically, commands are flagged in the *what* data member by *evCommand*, broadcasts by *evBroadcast*, and user-defined messages by some user-defined constant.

"Nothing" events

A "nothing" event is really a dead event. It has ceased to be an event, because it has been completely handled. If the *what* data member in an event record contains the value *evNothing*, that event record contains no useful information that needs to be dealt with.

When a Turbo Vision object finishes handling an event, it calls a member function called **clearEvent**, which sets the *what* data member back to *evNothing*, indicating that the event has been handled. Objects should simply ignore *evNothing* events, as they have already been dealt with by another object.

Events and commands

Ultimately, most events end up being translated into commands of some sort. For example, clicking the mouse on an item in the status line generates a mouse event. When it gets to the status line object, that object responds to the mouse event by generating a command event, with the *command* data member value determined by the command bound to the status line item. A mouse click on `Alt-X Exit` generates the *cmQuit* command, which the application interprets as an instruction to shut down and terminate.

Routing of events

Turbo Vision's views operate on the principle "Speak only when spoken to." That is, rather than actively seeking out input, they wait passively for the event manager to tell them that an event has occurred to which they need to respond.

In order to make your Turbo Vision programs act the way you want them to, you not only have to tell your views what to do when certain events occur, you also need to understand how events get to your views. The key to getting events to the right place is correct *routing* of the events. Some events get broadcast all over the application, while others are directed rather narrowly to particular parts of the program.

Where do events come from?

As noted in Chapter 1, "Inheriting the wheel," the main processing loop of a **TApplication**, the **run** member function, calls **TGroup::execute**, which is basically a repeat loop that looks something like this:

```
TEvent event;
event.what = evNothing;   // indicate no event has occurred
do {
   if (event.what != evNothing) eventError(event);
   getEvent(event);        // get an event
   handleEvent(event);     // route the event to the right place
} while (endState == continue); // until the quit flag is set
```

getEvent, handleEvent *and* ***eventError*** *are described in greater detail on pages 138, 133, and 137, respectively.*

Essentially, **getEvent** looks around and checks to see if anything has happened that should be an event. If it has, **getEvent** sets appropriate values in the event object. **handleEvent** then routes the event to the proper views. If the event is not handled (and cleared) by the time it gets back to this loop, **eventError** is called to indicate an abandoned event. By default, **eventError** does nothing.

Where do events go?

Events *always* begin their routing with the current modal view. For normal operations, this usually means your application object. When you execute a modal dialog box, that dialog box object is the modal view. In either case, the modal view is the one that initiates event handling. Where the event goes from there depends on the nature of the event.

Events are routed in one of three ways, depending on what kind of event they are. The three possible routings are positional, focused, and broadcast. It is important to understand how each kind of event gets routed.

Positional events

Positional events are virtually always mouse events (*evMouse*).

Z-order is explained on page 98 in Chapter 4.

The modal view gets the positional event first, and starts looking at its subviews in Z-order until it finds one that contains the position where the event occurred. The modal view then passes the event to that view. Since views can overlap, it is possible that more than one view will contain that point. Going in Z-order guarantees that the topmost view at that position will be the one that receives the event. After all, that's the one the user clicked on!

This process continues until an object cannot find a view to pass the event to, either because it is a terminal view (one with no subviews) or because there is no subview in the position where the event occurred (such as clicking on open space in a dialog box). At that point, the event has reached the object where the positional event took place, and that object handles the event.

Focused events

Focused events are generally keystrokes (*evKeyDown*) or commands (*evCommand*), and they are passed down the focus chain.

For details on focused views and the focus chain, see page 105 in Chapter 4.

The current modal view gets the focused event first, and passes it to its selected subview. If that subview has a selected subview, it passes the event to it. This process continues until a terminal view is reached: This is the focused view. The focused view receives and handles the focused event.

Non-focused views may handle focused events. See page 129.

If the focused view does not know how to handle the particular event it receives, it passes the event back up the focus chain to its owner. This process is repeated until the event is handled or the event reaches the modal view again. If the modal view does not know how to handle the event when it comes back, it calls **eventError**. This situation is an *abandoned event*. See page 137 for more on abandoned events.

Keyboard events illustrate the principle of focused events quite clearly. For example, in the Borland IDE (integrated development environment), you might have several files open in editor windows on the desk top. When you press a key, you know which file you intend to get the character. Let's see how Turbo Vision ensures it actually gets there.

Your keystroke produces an *evKeyDown* event, which goes to the current modal view, the **TApplication** object. **TApplication** sends the event to its selected view, the desk top (the desk top is always **TApplication**'s selected view). The desk top sends the event to its

selected view, which is the active window (the one with the double-lined frame). That editor window also has subviews—a frame, a scrolling interior view, and two scrollbars. Of those, only the interior is selectable (and therefore selected, by default), so the keyboard event goes to it. The interior view, an editor, has no subviews, so it gets to decide how to handle the character in the *evKeyDown* event.

Broadcast events

Broadcast events are generally either broadcasts (*evBroadcast*) or user-defined messages. Broadcast events are not as directed as positional or focused events. By definition, a broadcast does not know its destination, so it is sent to *all* the subviews of the current modal view.

The current modal view gets the event, and begins passing it to its subviews in Z-order. If any of those subviews is a group, it too passes the event to its subviews, also in Z-order. The process continues until all views owned (directly or indirectly) by the modal view have received the event.

Broadcasts can be directed to an object with the ***message*** *function.*

Broadcast events are commonly used for communication between views. For example, when you click on a scroll bar in a file viewer, the scroll bar needs to let the text view know that it should show some other part of itself. It does that by broadcasting a view saying "I've changed!", which other views, including the text, will receive and react to. For more details, see page 139.

User-defined events

As you become more comfortable with Turbo Vision and events, you may wish to define whole new categories of events, using the high-order bits in the *what* data member of the event record. By default, Turbo Vision will route all such events as broadcast events. But you may wish your new events to be focused or positional, and Turbo Vision provides a mechanism to allow this.

Manipulating bits in masks is explained in Chapter 10, "Hints and tips."

Turbo Vision defines two masks, *positionalEvents* and *focusedEvents*, which contain the bits corresponding to events in the event record's *what* data member that should be routed by position and by focus, respectively. By default, *positionalEvents* contains all the *evMouse* bits, and *focusedEvents* contains (*evKeyboard* | *evCommand*). If you define some other bit to be a new kind of event that you want routed either by position or focus, you simply add that bit to the appropriate mask.

Masking events

Every view object has a bitmapped **ushort** data member called *eventMask* which is used to determine the events the view will handle. The bits in the *eventMask* correspond to the bits in the **TEvent::what** data member. If the bit for a given kind of event is set (to 1), the view will accept that kind of event for handling. If the bit for a kind of event is cleared (to 0), the view will ignore that kind of event.

Phase

There are certain times when you want a view other than the focused view to handle focused events (especially keystrokes). For example, when looking at a scrolling text window, you might want to use keystrokes to scroll the text, but since the text window is the focused view, keystroke events go to it, not to the scroll bars that can scroll the view.

Turbo Vision provides a mechanism to allow views other than the focused view to see and handle focused events. Although the routing described in the "Focused events" section of this chapter is generally followed, there are two exceptions to the strict focus-chain routing.

When the modal view gets a focused event to handle, there are actually three "phases" to the routing:

- The event is sent to any subviews (in Z-order) that have their *ofPreProcess* option flags set.
- If the event isn't cleared by any of them, the event is sent to the focused view.
- If the event still hasn't been cleared, the event is sent (again in Z-order) to any subviews with their *ofPostProcess* option flags set.

So in the preceding example, if a scroll bar needs to see keystrokes that are headed for the focused text view, the scroll bar should be initialized with its *ofPreProcess* option flag set. If you look at the example program TVGUID09.CPP, you will notice that the scroll bars for the interior views all have their *ofPostProcess* bits set. If you modify the code to *not* set those bits, keyboard scrolling will be disabled.

Notice also that in this particular example it doesn't make much difference whether you set *ofPreProcess* or *ofPostProcess*: Either one will work. Since the focused view in this case doesn't handle the event (**TScroller** itself doesn't do anything with keystrokes), the scroll bars may look at the events either before *or* after the event is routed to the scroller.

In general, however, you would want to use *ofPostProcess* in a case like this, because it provides greater flexibility. Later on you may wish to add functionality to the interior that checks keystrokes, but if the keystrokes have been taken by the scroll bar before they get to the focused view (*ofPreProcess*), your interior will never get to act on them.

Although there are times when you will *need* to grab focused events before the focused view can get at them, it's a good idea to leave as many options open as possible so that you (or someone else) can derive something new from this object in the future.

The phase data member

Every group has a data member flag of type **enum phaseType**:

```
enum phaseType { phFocused, phPreProcess, phPostProcess}
```

By checking its owner's *phase* flag, a view can tell whether the event it is handling is coming to it before, during, or after the focused routing. This is sometimes necessary because some views look for different events or react to the same events differently, depending on the phase.

Consider the case of a simple dialog box that contains an input line and a button labeled "**A**ll right," with *A* being the hot key for the button. With normal dialog box controls, you don't really have to concern yourself with phase. Most controls have *ofPostProcess* set by default, so keystrokes (focused events) will get to them and allow them to grab the focus if it is their hot key that was typed. Pressing *A* moves the focus to the "All right" button.

But suppose the input line has the focus, so keystrokes get handled and inserted by the input line. Pressing the *A* key puts an "A" in the input line, and the button never gets to see the event, since the focused view handled it. Your first instinct might be to have the button check before the process for the *A* key, so it can handle the hot key event before the focused view handles it. Unfortunately, this would always preclude your typing the letter "A" in the input line!

The solution is actually rather simple: Have the button check for different hot keys before and after the focused view handles the event. Specifically, by default, a button will look for its hot key in *Alt*-letter form before the process, and in straight letter-only form after the process. That's why you can always use the *Alt-letter* hot keys in a dialog box, but you can only use regular letters when the focused control doesn't grab the keystrokes immediately.

This is easy to do. By default, buttons have both *ofPreProcess* and *ofPostProcess* set, so they get to see focused events both before *and* after the focused view does. But within its **handleEvent**, the button only checks certain keystrokes if the focused control has already seen the event:

```
switch (event)
{
    ...
    case evKeyDown:
    {
       c = hotKey(title);
       if ( (event.keyCode == getAltCode(c)) ||
          (owner->phase == phPostProcess) && (c != '0') &&
          (toupper(event.charCode) == c) ||
          ((state & sfFocused) != 0) &&
          (event.charCode == ' ') )
          {
             pressButton();
             clearEvent(event);
          }
    }
    break;
    ...
}
```

Commands

Most positional and focused events wind up getting translated into commands by the objects that handle them. That is, an object often responds to a mouse click or a keystroke by generating a command event.

For example, by clicking on the status line, you generate a positional (mouse) event. The application determines that the click was positioned in the area controlled by the status line, so it passes the event to the status line object, **statusLine*.

statusLine determines which of its status items controls the area where you clicked, and reads the status item record for that item. That item will usually have a command bound to it, so *statusLine* creates a pending event record with the *what* data member set to *evCommand* and the *command* data member set to whatever command was bound to that status item. It then clears the mouse event, meaning that the next event found by **getEvent** will be the command event just generated.

Defining commands

Turbo Vision has many predefined commands, and you will define many more yourself. When you create a new view, you will also create a command that will be used to invoke the view. You can name commands anything, but Turbo Vision's convention is that a command identifier should start with *cm*.

The actual mechanics of creating a command are simple—you just create a constant:

```
const cmConfuseTheCat = 100;
```

Turbo Vision reserves commands 0 through 99 and 256 through 999 for its own use. Your applications may use the numbers 100 through 255 and 1,000 through 65,535 for commands.

The reason for having two ranges of commands is that only the commands 0 through 255 can be disabled. Turbo Vision reserves some of the commands that can be disabled and some of the commands that cannot be disabled for its standard commands and internal workings. You have complete control over the remainder of the commands.

The ranges of available commands are summarized in Table 5.1.

Table 5.1
Command ranges

Range	Reserved	Can be disabled
0 to 99	Yes	Yes
100 to 255	No	Yes
256 to 999	Yes	No
1,000 to 65,535	No	No

Binding commands

When you create a menu item or a status line item, you bind a command to it. When the user chooses that item, an event record is generated, with the *what* data member set to *evCommand*, and the *command* data member set to the value of the bound command. The command may be either a Turbo Vision standard command or one you have defined. At the same time you bind your command to a menu or status line item, you may also bind it to a hot key. That way, the user can invoke the command by pressing a single key as a shortcut to using the menus or the mouse.

The important thing to remember is that defining the command does not specify what action will be taken when that command appears in an event record. You will have to tell the appropriate objects how to respond to that command.

Enabling and disabling commands

There are times when you want certain commands to be unavailable to the user for a period of time. For example, if you have no windows open, it makes no sense for the user to be able to generate *cmClose*, the standard window closing command. Turbo Vision provides a way to disable and enable sets of commands.

Specifically, to enable or disable a group of commands, you use the class **TCommandSet**, which simulates a *set* of numbers 0 through 255. (This is why only commands in the range 0 to 255 can be disabled.) The following code disables two window-related commands:

*Note the natural use of the overloaded operator **+=**. See **TCommandSet** in Chapter 13.*

```
TCommandSet windowCommands;
  ...
  windowCommands += cmNext;
  windowCommands += cmPrev;
  disableCommands(windowCommands);
```

Handling events

Once you have defined a command and set up some kind of control to generate it—for example, a menu item or a dialog box

button—you need to teach your view how to respond when that command occurs.

Every view inherits a virtual function **handleEvent** that already knows how to respond to much of the user's input. If you want a view to do something specific for your application, you need to override its **handleEvent** and teach the new **handleEvent** two things—how to respond to new commands you've defined, and how to respond to mouse and keyboard events the way you want.

A view's **handleEvent** determines how it behaves. Two views with identical **handleEvent** member functions will respond to events in the same way. When you derive a new view class, you generally want it to behave more or less like its base view, with some changes. By far the easiest way to accomplish this is to call the base **handleEvent** as part of the new object's **handleEvent** member function.

The general layout of a derived class's **handleEvent** would look like this:

```
virtual void TNewView::handleEvent( TEvent event )
{
  // code to change or eliminate base view behavior
  TBaseView::handleEvent(event);
  // code to perform additional functions
}
```

In other words, if you want your new class to handle certain events differently from its base class does (or not at all), you would trap those particular events *before* passing the event to the base **handleEvent** member function. If you want your new object to behave just like its ancestor, but with certain additional functions, you would add the code to do that *after* the call to the base **handleEvent**.

The event record

Up to this point, this chapter has discussed events in a fairly theoretical fashion. We have talked about the different kinds of events (mouse, keyboard, message, and "nothing") as determined by the event's *what* data member. We have also discussed briefly the use of the *command* data member for command events.

Now it's time to discuss what an event object actually looks like. The SYSTEM.H header file defines the **TEvent** structure as follows:

```
struct TEvent
{
   ushort what;
   union
   {
      MouseEventType mouse;
      KeyDownEvent keyDown;
      MessageEvent message;
   };
   void getMouseEvent();
   void getKeyEvent();
};
```

*Recall that in C++, a **struct** is simply a class with public access defaults.*

The two member functions **getMouseEvent** and **getKeyEvent** are called by **TProgram::getEvent**; you will not usually need to call them directly yourself. They are described in detail in Chapter 13. Briefly, **getMouseEvent** polls the mouse event queue maintained by Turbo Vision's event handler. If a mouse event has occurred, **TEvent::what** is set to the appropriate *evMouse*xxx value (where *xxx* is *Down*, *Up*, *Move* or *Auto*). You'll see shortly how the mouse data member is set to indicate button and position information. If no mouse event has ocurred, **TEvent::what** is set to *evNothing*.

getKeyEvent calls the BIOS INT 16H service to check if a keyboard event has occurred. If so, **TEvent::what** is set to *evKeyDown* and the *keyDown* data member records the scan code of the key that's been used. In the absence of a keyboard event, **TEvent::what** is set to *evNothing*.

Depending on the value of **TEvent::what**, the contents of the union within **TEvent** will be interpreted as *mouse*, *keyDown*, or *messsage* events. Let's look at each of these types in detail. First, the **MouseEventType** is another structure defined as follows

```
struct MouseEventType
{
   uchar buttons;
   Boolean doubleClick;
   TPoint where;
};
```

A mouse event object, therefore, tells you where the mouse cursor is, which buttons were used, and whether there was a double click or not.

If **TEvent::what** is *evKeyDown*, you access the event data via the **KeyDownEvent** structure. This contains a union as follows:

```
struct KeyDownEvent
{
    union
        {
          ushort keyCode;
          CharScanType charScan;
        };
};
```

where **CharScanType** is also a structure:

```
struct CharScanType
{
    uchar charCode;
    uchar scanCode;
};
```

This means that a key down event can be interpreted either as a **ushort** *keyCode* or as a pair of bytes: *charCode* and *scanCode*.

Turning to the third member of the union in **TEvent**, we have the **MessageEvent** type defined as follows:

```
struct MessageEvent
{
    ushort command;
    union
        {
          void *infoPtr;
          long infoLong;
          ushort infoWord;
          short infoInt;
          uchar infoByte;
          char infoChar;
        };
};
```

The union in **MessageEvent** can hold one of several items: a generic pointer or an integral value of the types shown. These message values are used in a variety of ways in Turbo Vision. Views can actually generate events themselves and send them to other views, and when they do, they often use the *infoPtr* data member. Communication among views and the *infoPtr* data member are both covered starting on page 139.

Clearing events

When a view's **handleEvent** has handled an event, it finishes the process by calling its **clearEvent** member function. **clearEvent** sets *event.what* to *evNothing* and *event->infoPtr* to *this*. These are the universal signals that the event has been handled. If the "nothing" event then gets passed to another view, that view should ignore it.

Abandoned events

Normally, every event will be handled by some view in your application. If no view can be found to handle an event, the modal view calls **eventError**. **eventError** calls the owner's **eventError** and so on up the view tree until **TApplication::eventError** is called.

TApplication::eventError by default does nothing. You may find it useful during program development to override **eventError** to bring up an error dialog box or issue a beep. Since the end user of your software isn't responsible for the failure of the software to handle an event, such an error dialog box would probably be inappropriate in a shipping version.

clearEvent also helps views communicate with each other. For now, just remember that you haven't finished handling an event until you call **clearEvent**.

Modifying the event mechanism

At the heart of the current modal view is a loop that looks something like this:

```
TEvent event;
   event.what = evNothing;
   do {
      if (event.what != evNothing) eventError(event);
      getEvent(event);
      handleEvent(event);
   } (while endState != continue);
```

Centralized event gathering

One of the greatest advantages of event-driven programming is that your code doesn't have to know where its events come from. A window object, for example, just needs to know that when it sees a *cmClose* command in an event, it should close. It doesn't care whether that command came from a click on its close icon, a menu selection, a hot key, or a message from some other object in the program. It doesn't even have to worry about whether that command is intended for it. All it needs to know is that it has been given an event to handle, and since it knows how to handle that event, it does.

The key to these "black box" events is the application's **getEvent** member function. **getEvent** is the only part of your program that has to concern itself with the source of events. Objects in your application simply call **getEvent** and rely on it to take care of reading the mouse, the keyboard, and the pending events generated by other objects.

If you want to create new kinds of events (for example, reading characters from a serial port), you would simply override **TApplication::getEvent** in your derived application class. In **TApplication::getEvent**, the **getEvent** loop polls for mouse and keyboard events and then calls **idle**. To insert a new source of events, you could either override **idle** to look for characters from the serial port and generate events based on them, or override **getEvent** itself to add a **getComEvent(***event***)** call to the loop, where **getComEvent** returns an event record if there is a character available at the designated serial port.

Overriding getEvent

The current modal view's **getEvent** calls its owner's **getEvent**, and so on, all the way back up the view tree to **TApplication::getEvent**, which is where the next event is always actually fetched.

Because Turbo Vision always uses **TApplication::getEvent** to fetch events, you can modify events for your entire application by overriding just this one member function. For example, to implement keystroke macros, you could watch the events returned by **getEvent**, grab certain keystrokes, and unfold them into macros.

As far as the rest of the application would know, the stream of events would be coming straight from the user.

```
virtual void TMyApp::getEvent(TEvent& event)
{
  TApplication::getEvent(event);
  // { special processing here }
```

Using idle time

Another benefit of **TApplication::getEvent**'s central role is that it calls **TApplication::idle** if no event is ready. **TApplication::idle** is a dummy (empty) member function that you can override in order to carry out processing concurrent with that of the current view.

An example of a heap viewer is included in the example programs on your distribution disks.

Suppose, for example, you define a view called **THeapView** that uses a member function called **update** to display the currently available heap memory. If you override **TApplication::idle** with the following code, users will be able to see a continuous display of the available heap memory, no matter where they are in your program.

```
virtual void TMyApp::idle()
{
  THeapView::update();
}
```

Inter-view communication

Suppose an object needs to exchange information with another object within your program. In a traditional program, that would probably just mean copying information from one data structure to another. In an object-oriented program, that may not be so easy, since the objects may not know where to find one another.

If you need to do inter-view communication, the first question to ask is if you have divided the tasks up between the two views properly. It may be that the problem is one of poor class design. Perhaps the two views really need to be combined into one view, or part of one view moved to the other view.

Intermediaries

If indeed the program design is sound, and the views still need to communicate with each other, it may be that the proper path is to create an intermediary view.

For example, suppose you want to be able to paste something from a spreadsheet into a word processor, and vice-versa. In a Turbo Vision application, you can accomplish this with direct view-to-view communication. But suppose that at a later date you wanted to add, say, a database to this application, and to paste to and from the database. You will now need to extend the communication you established between the first two objects to cover all three.

A better solution is to establish an intermediary view—in this case, say, a clipboard. An object would then need to know only how to copy something to the clipboard, and how to paste something from the clipboard. No matter how many new objects you add to the group, the job will never become any more complicated than this.

Messages among views

If you've analyzed your situation carefully and are certain that your program design is sound and that you don't need to create an intermediary, you can implement simple communication between just two views.

Before one view can communicate with another, it may first have to find out where the other view is, and perhaps even make sure that the other view exists at the present time.

First, a straightforward example. The STDDLG library contains a dialog box called **TFileDialog**. **TFileDialog** has a **TFileList** object that shows you a disk directory, and above it, a **TFileInputLine** object that displays the file currently selected for loading. Each time the user selects another file from the **TFileList** object, it needs to tell the **TFileInputLine** object to display the new file name.

In this case, the **TFileList** object can be sure that the **TFileInputLine** object exists, because they are both initialized within the same **TFileDialog** object. How does **TFileList** tell **TFileInputLine** that the user has just selected a new name? **TFileList** creates and sends a message. Here's **TFileList::focusItem**, which sends the event, and **TFileInputLine's handleEvent**, which receives it:

```
void TFileList::focusItem( short item )
{
  TSortedListBox::focusItem( item );
  // call base member function first
  message(owner, evBroadcast, cmFileFocused, list()->at(item));
}

void TFileInputLine::handleEvent( TEvent& event )
{
    TInputLine::handleEvent(event);
    if( event.what == evBroadcast &&
       event.message.command == cmFileFocused &&
       !(state & sfSelected))
    {
       if( (((TSearchRec *)event.message.infoPtr)->attr & FA_DIREC)
           != 0 )
       {
          strcpy( data, ((TSearchRec *)event.message.infoPtr)->name );
          strcat( data, "\\" );
          strcat( data, ((TFileDialog *)owner)->wildCard );
       }
        else
          strcpy( data, ((TSearchRec *)event.message.infoPtr)->name );
          drawView();
    }
}
```

message is a function that generates a message event and returns a pointer to the object (if any) that handled the event.

Who handled the broadcast?

Suppose you need to find out if there is a window open on the desk top before you perform some action. How can you find this out? The answer is to have your code send off a broadcast event that windows know how to respond to. The "signature" left by the object that handles the event will tell you which window, if any, handled it. The process can be seen in the implementation of **TView::clearEvent**:

```
void TView::clearEvent( TEvent& event )
{
    event.what = evNothing;
    event.message.infoPtr = this;
}
```

The view object *v*, say, signals the successful handling of a broadcast event by calling **clearEvent**, so *infoPtr* is set to *&v*, allowing you to determine *v* by derefencing.

Is anyone out there?

Here's a concrete example from the Borland integrated development environment (IDE). If the user attempts to open a watch window, the code needs to check if there is a watch window already open. If not, it opens one; if there is a watch window open, it brings it to the front.

You broadcast a message using the **message** function:

```
areYouThere = message(deskTop, evBroadcast, cmFindWindow, 0);
```

A watch window's **handleEvent** responds to *cmFindWindow* by clearing the event:

```
switch (event.message.command)
{
  ...
  case cmFindWindow:
     clearEvent(event);
     break;
  ...
}
```

clearEvent, remember, not only sets the event record's *what* data member to *evNothing*, it also sets *infoPtr* to **this**. **message** reads these data members, and if the event has been handled, it returns a pointer to the object that handled the message event. In this case, that would be the watch window. So following the line that sends the broadcast, include

```
if (areYouThere == 0)
  createWatchWindow();     // if there is none, create one
else areYouThere->select;  // otherwise bring it to the front
```

As long as a watch window is the only object that knows how to respond to the *cmFindWindow* broadcast, your code can be assured that when it finishes, there will be one and only one watch window at the front of the views on the desk top.

Who's on top?

Using the same techniques outlined earlier, you can also determine, for example, which window is the topmost view of its type on the desk top. Because a broadcast event is sent to each of the modal view's subviews in Z-order (reverse insertion order), the most recently inserted view is the view "on top" of the desk top.

Consider for a moment the situation encountered in the Borland IDE when the user has a watch window open on top of the desk top while stepping through code in an editor window. The watch window can be the active window (double-lined frame, top of the stack), but the execution bar in the code window needs to keep tracking the executing code. If you have multiple editor windows open on the desk top, they might not overlap at all, but the IDE needs to know which one of the editors it is supposed to be tracking in.

The answer, of course, is the front, or topmost editor window, which is defined as the last one inserted. In order to figure out which one is "on top," the IDE broadcasts a message that only editor windows know how to respond to. The first editor window to receive the broadcast will be the one most recently inserted; it will handle the event by clearing it, and the IDE will then know which window to use for code tracking by reading the result returned by **message**.

Calling handleEvent

You can also create or modify an event, then call **handleEvent**. You can call any view's **handleEvent**; that view will pass the event on down the tree.

Help context

Turbo Vision has built-in tools that help you implement context-sensitive help within your application. You can assign a help context number to a view, and Turbo Vision ensures that whenever that view becomes focused, its help context number will become the application's current help context number.

To create global context-sensitive help, you can implement a **helpView** that knows about the help context numbers that you've defined. When **helpView** is invoked (usually by the user pressing *F1* or some other hot key), it should ask its owner for the current help context by calling the member function **getHelpCtx**. **helpView** can then read and display the proper help text. Several examples are included in the demonstration programs.

Context-sensitive help is probably one of the last things you'll want to implement in your application, so **TView** objects are initialized with a default context of *hcNoContext*. This is a prede-

fined context that doesn't change the current context. When the time comes, you can work out a system of help numbers, then plug the right number into the proper view by setting the view's *helpCtx* data member right after you construct the view.

Help contexts are also used by the status line to determine which views to display. Remember that when you create a status line, the **TStatusDef** constructor creates a set of status items for a given range of help context values. When a new view receives the focus, the help context of that item determines which status line is displayed.

C H A P T E R

6

Writing safe programs

Handling errors in an event-driven, interactive user interface is much more complicated than in a command line utility. In a non-interactive application, it is quite acceptable (and indeed, expected) that errors cause the program to display an error message and terminate the program. In an interactive setting, however, the program needs to recover from errors and leave the user in an acceptable state. Errors should not be allowed to corrupt the information the user is working on, nor should they terminate the program, regardless of their nature. A program that meets these programming criteria can be considered "safe."

Turbo Vision facilitates writing safe programs. It promotes a style of programming that makes it easier to detect and recover from errors, especially the wily and elusive "Out of memory" error. It does this by promoting the concept of atomic operations.

All or nothing programming

An atomic operation is an operation that cannot be broken down into smaller operations as far as error-handling is concerned. Or, more specific to our use, it is an operation that either completely fails, or completely succeeds. Making operations atomic is especially helpful when dealing with memory allocation.

Typically, programs allocate memory in many small chunks. For example, when constructing a dialog box, you allocate memory

for the dialog box, then allocate memory for each of the controls. Each of these allocations could potentially fail, and each possible failure requires a test to see if you should proceed with the next allocation or stop. If any allocation does fail, you need to deallocate any memory allocated successfully. Ideally, you would allocate everything and then check to see if any of your allocations failed. Enter the safety pool.

The safety pool

Turbo Vision sets aside a fixed amount of memory (4K by default) at the end of the heap, called the safety pool. If allocating memory on the heap reaches into the safety pool, the function **lowMemory** returns *True*. This indicates that further allocations are not safe and might fail.

For the safety pool to be effective, the pool must be as large as the largest atomic allocation. In other words, it needs to be large enough to make sure that all allocations between checks of **lowMemory** will succeed; 4K should suffice in most applications.

The **TVMemMgr** and **TBufListEntry** classes provide low-level memory management for Turbo Vision. Turbo Vision overrides the operator **new** so that if there is insufficient room on the heap for a memory allocation, certain steps are taken before **new** aborts. First, if there are any cache buffers assigned for group draw operations, the first such is freed and the allocation is tried again. If this allocation fails, **new** frees the next cache buffer, and so on. If the allocation still fails after all the cache buffers have been relinquished, the safety pool is tested. If the safety pool is exhausted, the **new** call aborts. Otherwise, the safety pool is freed and the allocation is attempted for the last time. **new** aborts if this attempt fails.

Doing it the old, hard way

Using the traditional approach to memory allocation, constructing a dialog box would look something like this:

```
Boolean OK = True;

TDialog *pd = new TDialog( TRect(20,3,60,10), "My dialog");
if pd != 0
{
    TStaticText *control = new TStaticText(TRect(2,2,32,3),
                        'Do you really wish to do this?');
    if (control !=0) insert(control)
    else OK = False;

    TButton *control = new TButton(TRect(5,5,14,7), '~Y~es', cmYes);
    if (control !=0) insert(control)
    else OK = False;

    TButton *control = new TButton(TRect(16,6,25,7), '~N~o', cmNo);
    if (control !=0) insert(control)
    else OK = False;

    TButton *control = new TButton(TRect(27,5,36,7), '~C~ancel',
                                   cmCancel);
    if control !=0 insert(control)
    else OK = False;
}
  if (!OK) destroy(pd);
```

Note that the variable *OK* is used to indicate if any of the allocations failed. If any did, the whole dialog box needs to be disposed. Remember, disposing of a dialog box also disposes of all its subviews.

Doing it the new, easier way

On the other hand, with a safety pool this entire block of code can be treated as an atomic operation, changing the code to this:

```
TDialog *pd = new TDialog( TRect(20,3,60,10), "My dialog");
TStaticText *control = new TStaticText(TRect(2,2,32,3),
                        'Do you really wish to do this?');
TButton *control = new TButton(TRect(5,5,14,7), '~Y~es', cmYes);
TButton *control = new TButton(TRect(16,6,25,7), '~N~o', cmNo);
TButton *control = new TButton(TRect(27,5,36,7), '~C~ancel',
                               cmCancel);

if (lowMemory())            // check if we dipped into the safety
pool
{
  destroy(pd);
  outOfMemory();          // report out-of-memory error
  doIt = False;
}
else
  doIt = (desktop->execView(pd) == cmYes);
```

Since the safety pool is large enough to allocate the entire dialog box, which takes up much less than 4K, the code can assume that all the allocations succeeded. After the dialog box is completely allocated, **lowMemory** is checked, and if *True*, the entire dialog box is disposed of; otherwise, the dialog box is used.

The validView member function

Since the **lowMemory** check is done quite often, **TApplication** has a method called **validView** that can be called to perform the necessary check. Using **validView**, the **lowMemory** test in the last eight lines of the code can be condensed into two:

```
doIt = (validView(pd) != 0) && (desktop->execView(pd) == cmYes);
```

validView returns either a pointer to the view passed or 0 if the view was invalid. If **lowMemory** returns *True*, **validView** takes care of disposing the view in question and calling **outOfMemory**.

Delete and destroy

In the previous code, you may have noticed that the dialog object was deleted by calling **destroy**(*pd*) rather than the traditional C++ **delete** operator. Because of the many class and object inter-dependencies in Turbo Vision, objects derived from **TObject** must be deleted in specific sequences. To this end, **TObject** has a static member function called **destroy** that takes a single **TObject** pointer argument. You should call this rather than the usual **delete** operator when discarding objects derived directly or ultimately from **TObject**.

Behind the scenes, as it were, **destroy** calls a virtual function called **shutDown**. The **shutDown** member function of **TObject** is overridden in many of **TObject**'s derived classes, performing appropriate object deletion in the correct sequence. For example, **TGroup::shutDown** destroys each of its subviews, frees its buffer, and so on, and finally calls **TView::shutDown** to delete the view itself. You need not be concerned with this mechanism unless you venture into advanced modifications of Turbo Vision's memory management system. In normal applications, just use **new** in the natural way, and remember to use **destroy** rather than **delete** when disposing directly of an object derived from **TObject**.

Non-memory errors

Of course, not all errors are memory related. For example, a view could be required to read a disk file for some information, and the file might be missing or invalid. This type of error must also be reported to the user. Fortunately, **validView** has a built-in "hook" for handling non-memory errors: It calls the view's **valid** member function.

TView::valid returns *True* by default. **TGroup::valid** only returns *True* if *all* the subviews owned by the group return *True* from their **valid** functions. In other words, a group is valid if all the subviews of the group are valid. When you create a view that may encounter non-memory errors, you will need to override **valid** for that view to return *True* only if it has been successfully instantiated.

You can use **valid** to indicate that a view should not be used for any reason; for example, if the view could not find its file. Note that what **valid** checks for and how it checks are entirely up to you. A typical **valid** function might look something like this:

```
virtual Boolean TMyView::valid(ushort command)
{
   if ( command == cmValid && errorEncountered )
   {
      reportError();
      return False;
   }
   else
      return True;
}
```

When a view is first instantiated, you should call its **valid** member function with a *command* argument of *cmValid* to check for any non-memory related errors involved in the creation of the view. `validView(X)` calls `X::valid(cmValid)` automatically, as well as checking the safety pool, so calling **validView** before using any new view is a good idea.

valid is also called whenever a modal state terminates, with the *command* argument being the command that terminated the modal state (see Chapter 4, "Views"). This gives you a chance to trap for conditions like unsaved text in an editor window before terminating your application.

errorEncountered would usually be a Boolean data member that you specify and set according to the needs of the program. Typically, the **TMyView** constructor would initialize *error-Encountered* to *False*, and your application-dependent error checks would set it to *True*.

Reporting errors

Before a **valid** member function returns *False*, it should let the user know about whatever error occurred, since the view is not going to show up on the screen. This is what the **reportError** call in the previous example does. Typically this involves popping up a message dialog box. Each individual view, then, is responsible for reporting any errors, so the program itself does not have to know how to check each and every possible condition.

This is an important advance in programming technique, because it lets you program as if things were going right, instead of always looking for things going wrong. Group objects, including applications, don't have to worry about checking for errors at all, except to see if any of the views they own were invalid, in which case the group simply disposes of itself and its subviews and indicates to *its* owner that it was invalid. *The group can assume that its invalid subview has already notified the user of the problem.*

Using **valid** allows the construction of windows and dialog boxes to be treated as atomic operations. Each subview that makes up the window can be constructed without checking for failure; if the constructor fails, its **valid** simply returns *False*. The window then goes through its entire construction, at which point the entire window can be passed to **validView**. If any of the subviews of the window are invalid, the entire window returns *False* from the valid check. **validView** will dispose of the window and return 0. All that needs to be done is to check the return result from **validView**.

Major consumers

The **valid** function can also handle *major consumers*, which are views that allocate memory greater than the size of the safety pool, such as reading the entire contents of a file into memory. Major consumers should check **lowMemory** themselves, instead of waiting until they have finished all construction and then allowing **validView** to do so for them.

If a major consumer runs out of memory in the middle of constructing itself, it should set a data member flag to indicate that it encountered an error (such as the *errorEncountered* flag in the earlier example) and stop trying to allocate more memory. The flag would be checked in **valid** and the view would call `MyApplication->outOfMemory` and return *False* from the **valid** call.

Obviously, this is not quite as nice as being able to assume that your constructors work, but it is the only way to manage the construction of views that exceed the size of your safety pool.

The program FILEVIEW.CPP included on the distribution disks demonstrates the use of these techniques to implement a safe file viewer. For example, the **TFileViewer** constructor sets *isValid* to *True*. If a call to **readFile** fails, **messageBox** displays an "Invalid drive or directory" message and sets *isValid* to *False*.

C H A P T E R

7

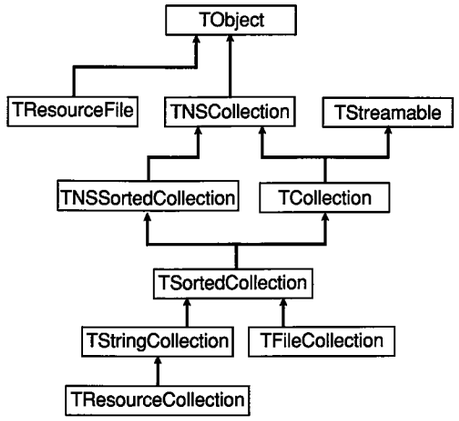
Collections

The traditional arrays of C suffer from two main restrictions. First, the size of an initialized array is fixed at compilation time. Second, the data type of each element must be the same for a given array. The latter restriction can be circumvented by using arrays of generic pointers, but care is needed to avoid type mismatching. Turbo Vision offers help in both problems by providing several *collection* classes. Collections can be considered as generalized arrays of arbitrary objects that can be dynamically resized. You'll see later that collections can also be made *streamable*, meaning that they can be written to output streams and read from input streams in a relatively type-safe manner. Streamable classes implement *persistent* objects with many important applications. For example, you can save all the subviews in any group (your entire application, perhaps) and restore them with just a few lines of code.

From the basic **TNSCollection** (non-streamable collection) and **TCollection** (the streamable version) classes, many specialized classes can be derived: sorted collections, collections of files, and so on. Further specialized classes of streamable collections, known as *resources*, let you save and recover arbitrary objects using strings (names) as indexes. The following figure illustrates the class hierarchy for collection and resource classes.

Figure 7.1
Collections and resources

*As is customary in C++, the arrows point **toward** the base class.*



TCollection class

The **TNSCollection** class provides the basic mechanisms for storing and manipulating a collection of objects. Its streamable offspring, **TCollection** is derived multiply from **TNSCollection** and **TStreamable**. So, **TCollection** derives its access and storage functionality from **TNSCollection**, while its "streamability" comes partly from **TStreamable**, and partly from two **private** pure virtual member functions, **TCollection:readItem** and **TCollection::writeItem**. You'll see in Chapter 8 that the latter must be overridden in each derived class to provide basic stream I/O for their particular data types.

In the following discussion, we'll often talk about collection classes and objects, meaning classes and objects derived from either **TCollection** or **TNSCollection**.

TNSCollection, **TCollection**, and **TStreamable** are intended solely as bases for further, useful class derivation. In fact, **TCollection** and **TStreamable** are both abstract classes (each having at least one pure virtual function) so you *cannot* instantiate **TCollection** or **TStreamable** objects directly. **TNSCollection** is not strictly an abstract class, so you can create instances of it, as you'll see in TVGUID17.CPP; however, such objects have certain limitations in real applications.

The **TNSCollection** class (where *NS* stands for non-streamable), derived from **TObject**, provides all the basic members for collections. These include accessing, adding, and removing items from a collection, together with three *iterators*. Iterators are functions that let you apply your own functions to all or selected items in a collection. **TStreamable** is an abstract class used in the naming and registration of streams. The classes **opstream** and **ipstream** are friends of **TStreamable**, meaning that the familiar C++ stream operators **>>** and **<<**, suitably overloaded, are available (more on this in Chapter 8).

TCollection, derived from **TNSCollection** and **TStreamable**, therefore provides streamable collections, although in this chapter we will be concentrating on the non-streamable properties inherited from **TNSCollection**. Even if you are not concerned with streaming collections, there is usually little overhead in using **TCollection** as a base rather than **TNSCollection**. Note that any class derived from a streamable class, such as **TCollection**, will also inherit from **TStreamable** to provide a streamable class.

Dynamic sizing

The number of items stored in a collection object, given by the data member *count*, can vary dynamically during program execution. Growth is controlled as follows: You set an initial size, *limit*, and an increment value, *delta*. If items are added beyond the given limit, the maximum permitted collection size is increased by *delta*. Setting *delta* to zero is therefore equivalent to having a fixed size collection.

Mixing item types in collections

The protected *items* data member of **TNSCollection** is declared as `void **items`. This array of generic pointers allows the efficient collection of an arbitrary number of arbitrary items (either objects or non-objects) of arbitrary size. If you mix data types in a collection object, the onus is on you to avoid type mismatch problems. **TNSCollection** objects "know nothing" about the items they are handed. They just hold on to them and give them back when asked. Derived classes, of course, can recast the item pointers to handle specific objects. In particular, with collections of mixed types, your derived collection classes must override the access methods, such as **atPut** (add an item at a given index) and **at** (return the item at a given index), in order to provide type checking.

Creating a collection

Creating a collection can be as simple as defining the data type you wish to collect. Suppose you're a consultant, and you want to store and retrieve the account number, name, and phone number of each of your clients. First you define the client structure class (**TClient**, derived from **TObject**) for the objects to be stored in the collection. You then provide a constructor for **TClient**. An explicit destructor is not required since **TObject::~TObject** will take care of client disposal.

```
struct TClient : public TObject
{
// all members public by default

// three data members
    const char *account;
    const char *name;
    const char *phone;

    TClient( char *newAccount, char *newName, char *newPhone);
    ~TClient();

};

// constructor
TClient::TClient( char *newAccount, char *newName, char *newPhone) {
    account = newStr( newAccount );
    name = newStr( newName );
    phone = newStr( newPhone );
};
```

You can now instantiate a collection and insert the client records into it. The main body of the program might include the following:

```
TNSCollection clientList( 50, 10 );    // limit is 50, delta is 10

clientList.insert( new TClient("90-167", "Smith, Zelda", "(800)
                               555-1212" ));
clientList.insert( new TClient("90-160", "Johnson, Agatha", "(302)
                               139-8913" ));
clientList.insert( new TClient("90-177", "Smitty, John", "(406)
                               987-4321"));
clientList.insert( new TClient("91-100", "Anders, Smitty", "(406)
                               111-2222"));

printAll( &clientList );
searchArea( &clientList, "(406)" );
return 0;
```

printAll *and* ***searchArea*** *are discussed on page 161.*

Notice how easy it was to build the collection. The first statement creates a new **TNSCollection** object called **clientList** with an initial size of 50 clients (the data member, *limit*). If more than 50 clients are inserted into **clientList**, its size will increase in increments of 10 clients whenever needed (the data member, *delta*). The next statements create and initialize new client objects and insert pointers to them into the collection.

The **insert** member function is virtual and can be overridden for specific purposes (typically to ensure type-safe operation). By default, **insert** adds items at the *end* of the collection and increments the collection's *count* data member. If the *limit* is reached, **insert** also handles the *delta* increase discussed previously. **insert** returns the **index** of the new item. The index is of type *ccIndex*, and occurs as an argument in many **TNSCollection** member functions. Although *ccIndex* is currently typedef'd as **int**, you should always use *ccIndex* in the interests of future portabiliy. The **at** member, for example, declared as

```
void *at( ccIndex index );
```

returns a pointer to the item at position *index*. The converse operation is **indexOf**, the gives you the index of a given item:

```
virtual ccIndex indexOf( void *item );
```

A variant of **insert** is **atInsert**, that inserts an item at a given index position. **remove** and **atRemove** are the equivalent member functions for removing items from a collection and adjusting the indexes accordingly. The **removeAll** member function removes all items and sets *count* to 0. These "remove" member functions remove but do not destroy the stored items; **free** and **freeAll** member functions remove *and* destroy items.

When a collection is destroyed at the end of the program, the entire collection, clients and all, are also usually destroyed. The action is determined by the Boolean data member *shouldDelete*: if *True* (the default), all items are removed and deleted via **freeAll**; if *False*, the collection limit is set to 0, but the items themselves are not destroyed. Remember that collections are arrays of pointers, so there is a vital difference between removing an item pointer and destroying the item itslef.

Iterator member functions

Inserting and deleting items are not the only common collection operations. Often you'll find yourself writing **for** loops to range over *all* the objects in the collection to display the data or perform some calculation. Other times, you'll want to find the first or last item in the collection that satisfies some search criterion. For these purposes, collections have three *iterator* member functions: **forEach**, **firstThat**, and **lastThat**. Each of these takes two arguments: a function pointer and a generic pointer **void** **arg*. The latter allows you pass arbitrary arguments in addition to the main iterator function.

The forEach iterator

forEach takes an *action* argument of type *ccAppFunc* and a generic pointer, *arg*:

```
void TNSCollection::forEach( ccAppFunc action, void *arg)
{
    for( ccIndex i = 0; i < count; i++ ) // count is current
                                         // number of items in
                                         // collection
        action( items[i], arg );
}
```

The action function is of type *ccAppFunc*, defined as:

```
typedef void (*ccAppFunc)( void *, void *);
```

The first argument of the action function is a pointer to an item in the collection; the second argument is the generic *arg* pointer passed to **forEach**. If the iterator does not require any additional data, you use a 0 for the second argument. **forEach** calls the action function once for each item in the collection, in the order that the items appear in the collection. The **printAll** function in TVGUID17 shows an example of a **forEach** iterator.

```
static void print( void *c, void * )
// make it static for safety in view of generic pointers // Note that
we will not be using the second argument {
    TClient *tc = (TClient *)c;
    cout << setiosflags( ios::left )
         << setw(10) << tc->account
         << setw(20) << tc->name
         << setw(16) << tc->phone
         << endl;
}

void printAll( TNSCollection *c )  // print info for all clients {
    cout << endl << "Client List:" << endl;
    c->forEach( &print, 0 ); // call print for each client }
```

Note that *&print*, passed to **forEach**, matches the *ccAppFunc* data type: Pointer to function taking (**void***, **void***) and "returning" **void**. For each item in the **TNSCollection** object, **c*, the function **print** is called to display each client's information.

The firstThat and lastThat iterators

In addition to being able to apply a function to every element in the collection, it is often useful to be able to find a particular element in the collection that matches some criterion. This is achieved with the **firstThat** and **lastThat** iterators. As their names imply, they search the collection in opposite directions until they find an item matching the predicate of a Boolean, *test* function passed as an argument.

firstThat and **lastThat** return a pointer to the first (or last) item that matches the search conditions. Consider the earlier example of the client list. Suppose that you can't remember a client's account number or exactly how her last name is spelled. Luckily, you distinctly recall that this was the first client you acquired in the state of Montana. Thus you want to find the first occurrence of a client in the 406 area code (since your list happens to be in chronological order). Here's a function using the **firstThat** member function that would do the job:

```
Boolean areaMatch( void *c, void *ph )
{
    char *areaToFind = (char *)ph;
    TClient *tc = (TClient *)c;
    //seek match in first 5 chars: "(xxx)"
    if( strncmp( areaToFind, tc->phone, 5 ) == 0 )
        return True;
    else
        return False;
}

void searchArea( TNSCollection *c, char *areaToFind )
{
    TClient *foundClient =
        (TClient *)(c->firstThat( &areaMatch, areaToFind ));
    if( !foundClient )
        cout << "No client met the search requirement" << endl;
```

```
    else
        {
            cout << "Found client:" << endl;
            print( foundClient, 0 );
        }
}
```

The test function pointer you pass to **firstThat** is of type:

```
typedef Boolean (*ccTestFunc)( void *, void *);
```

Apart from the return type, the test function follows the same pattern as the *ccAppFunc* type used in **forEach**. **areaMatch** uses the second *arg* argument to pass the target area code string, and returns *True* only if this string matches the client's area code found in *tc->phone*. If no object in the collection matches the search criteria, a zero pointer is returned, hence the test `if( !foundClient )`.

Remember: **forEach** takes a function pointer of type *ccAppFunc*, while **firstThat** and **lastThat** take a function pointer of type *ccTestFunc*. In all cases, the user-defined action or test function takes two generic pointers: one to an object in the collection, the other to any other data you might need to pass to the iterator. You need to recast these pointers to match the actual items expected in your collection.

In TVG17B.CPP, we show you a safer way of handling collections. We derive from **TNSCollection** a specific client collection class, called **TClientCollection**:

This is TVG17B.CPP.

```
class TClientCollection : public TNSCollection
{

public:

    TClientCollection (ccIndex aLimit, ccIndex aDelta ) :
           TNSCollection( aLimit, aDelta ) {}

    virtual ccIndex insert( TClient *tc )  { return
            TNSCollection::insert( tc ); }

    void printAll();
    void searchArea( char *areaToFind );

};
```

We can now override **TNSCollection::insert**(void **item*) to improve type safety. The new **insert** expects, and insists on, a **TClient** pointer argument, whereas the base version of **insert** will accept a pointer to anything. In a more complete application, you would also override **at**, **indexOf**, **remove**, **free**, and so on, to take **TClient** pointer arguments. Since we have a dedicated client class, we might as well make **printAll** and **searchArea** member functions, as shown in the previous code. With obvious adjustments, TVG17B.CPP follows the same strategy as TVGUID17.CPP.

Sorted collections

Sometimes you need to have your data in a certain order. Turbo Vision provides two special base collection classes that let you to order your data in any manner you want: **TNSSortedCollection** and **TSortedCollection**.

TNSSortedCollection, which is derived from **TNSCollection**, automatically sorts the objects it is given. It also automatically checks the collection when a new member is added and rejects duplicate members (unless you set the Boolean member *duplicates* to *True*). **TSortedCollection** is the streamable version of **TNSSortedCollection**, having both **TNSSortedCollection** and **TStreamable** as bases. The comments we made earlier on **TNSCollection** (non-streamable) and **TCollection** (streamable) apply to the sorted versions in an obvious way. To reduce verbiage, we'll often talk about sorted collections without specifying whether they are streamable or not.

One technical difference is that *both* **TNSSortedCollection** and **TSortedCollection** are abstract classes. To use them, you must first decide what type of data you're going to collect and define two member functions to meet your particular sorting requirements. As before, we'll base our examples on **TNSSortedCollection** to avoid streamability issues. Let's derive a new collection type from **TNSSortedCollection**, called **TSortedClientCollection**.

TSortedClientCollection already knows how to do all the real work of a collection. It can **insert** new client records and **delete** existing ones—it inherited all this basic behavior from **TNSCollection**. All you have to do is teach **TSortedClient-Collection** which data member to use as a sort key, and how to compare two clients in order to determine which one belongs ahead of the other in the collection. You do this by overriding the **keyOf** and **compare** member functions. The **compare** member is a private pure virtual function, by the way, so it must always be redefined. Consider the following extract:

```
class TSortedClientCollection : public TNSSortedCollection
{

public:

   virtual void *keyOf(void *item);

private:

   virtual int compare( void *key1, void *key2);
}

void *TSortedClientCollection::keyOf(void *item)
{
   return ( ( (TClient*)item )->name );
}

int TSortedClientCollection::compare(void *key1, void *key2)
{
   return (strcmp ( (char *)(key1), (char *)(key2) ));
```

*void * pointers must be typecast*

keyOf defines which data member or data members should be used as a sort key. In this case, it's the client's *name* data member. **compare** takes generic pointers to two sort keys and determines which one should come first in the sorted order. **compare** returns –1, 0, or 1, depending on whether *key1* is less than, equal to, or greater than *key2*. This example uses a straight alphabetical sort of the key (*name*) strings.

Note that since the keys returned by **keyOf** and passed to **compare** are *void* * pointers, you need to cast them to *char* * before dereferencing them.

That's all you have to define! Now you can revamp of TVG17B.CPP by using **TNSSortedCollection** as the base to give **TSortedClientCollection** in place of **TClientCollection**. Your clients can now be listed in alphabetical order:

This is TVGUID18.CPP.

```
int main()
{

   TSortedClientCollection clientList( 50, 10);
   ...
// as before

}
```

Notice also how easy it would be if you wanted the client list sorted by account number instead of by name. All you would have to do is change the **keyOf** member function to return the *account* data member instead of *name*.

String collections

Many programs need to keeping track of sorted strings. For this purpose, Turbo Vision provides a special purpose collection, **TStringCollection**. Note that the elements in a **TStringCollection** are are pointers to strings (type *char* **). Since a string collection is derived from **TSortedCollection**, duplicate strings are permitted only if the *duplicates* data member is set to *True* (the default is *False* which does not allow duplicates).

String collections are used just like other sorted collections. In the following example, **TWordCollection** is derived from **TStringCollection**, and given a constructor and **print** member function. The latter will print the whole word collection using an iterator function (to be described later).

This is TVGUID19.CPP.

```
class TWordCollection : public TStringCollection
{

public:

   TWordCollection( short aLimit, short aDelta ) : TStringCollection(
        aLimit, aDelta ) {}
   virtual void print();
};
```

The sorted word collection will be built by reading a text file, line by line, extracting each word from the line, and inserting the words into the **TWordCollection** object in alphabetical (ASCII) sequence. The **insertWord** function parses each line with the rather simplistic notion that a word is any sequence of non-punctuation/non-whitespace characters surrounded by punctuation/whitespace. **insertWord** also performs the insertion of each non-empty word it finds.

Since **TWordCollection** is a sorted collection class (via **TStringCollection** and **TSortedCollection**), it has a *duplicates* data member that controls whether duplicate word strings are admitted or not. In this example, *duplicates* is *False* (the default, so no explicit action is needed), and the collection will accept only unique words. The *duplicates* data member affects the action of the **search**, **indexOf**, and **insert** member functions, inherited from **TNSSortedCollection**. When **insert** is called, the collection is searched for the target item. If the item is not found, it is always inserted in the proper sort-maintaining position. If the item is found, what happens next depends on the value of *duplicates*. If

duplicates is *True*, the item is inserted ahead of its twin(s). Otherwise, no insertion is made.

The word collection is created in **main** as follows:

```
TWordCollection *wordCollection = new TWordCollection( 50, 5 );
```

wordCollection therefore holds 50 string pointers initially, then grows in increments of five. The **fileRead** function is similar to that used in TVGUID06.CPP, but here it reads one line at a time, and calls **insertWord** after each line. A point to be noted is the use of **newStr** in the insert call:

```
sc->insert( newStr( wstr ));
```

Compare this with the more "obvious" (and perfectly valid):

```
sc->insert( wstr );
```

The **newStr** global function makes a copy of *str*, the word string that has just been extracted, and it is the address of this copy that is passed to the collection. The idea here is that when the collection is de-allocated, it is the copy strings and their pointers that will be destroyed, rather than the "originals." In this example, the difference between the two inserts is not important. In other contexts, it may be vital; you'll need to store a copy if the original is going away. By copying it, the collection controls it.

After the word collection is complete, we use **print** to list all the words in the collection:

```
if( wordCollection->getCount() > 0 )
{
   wordCollection->print();
   cout << "Total word count = " << wordCollection->getCount() <<
           endl;
   }
else
   cout << "No words in WordCollection!" << endl;
```

Iterators revisited

The **print** member function uses the **forEach** iterator applied to **printWord** as follows:

```
/* Iterator */
static void printWord( void *w, void * )
{
   char *s = (char *)w;
   cout << s << endl;
}

void TWordCollection::print()
{
   forEach( &printWord, 0 );
}
```

The **forEach** member function (inherited all the way from **TNSCollection**!) traverses the entire collection one item at a time, and passes each item to the function you provide.

Finding an item

Sorted collections (and therefore string collections) have a **search** member function that returns the index of an item with a particular key. But how do you find an item in a collection that may not be sorted? Or when the search criteria do not involve the key itself? One useful answer is to use **firstThat** and **lastThat**. You simply define a Boolean function to test for whatever criteria you want, and call **firstThat**. Note that with sorted collections that allow duplicates, **search** will find the *first* match if there are more than one. Your program may then have to scan ahead to check for duplicates.

Polymorphic collections

You've seen that collections can store any type of data dynamically, and there are plenty of member functions to help you access collection data efficiently. In fact, **TNSCollection** itself inherits or defines over 18 member functions. When you use collections in your programs, you'll be equally impressed by their speed. They're designed to be flexible and implemented to be fast.

But now comes the *real* power of collections: Items can be treated polymorphically. This means you can store many different object types, from anywhere in your class hierarchy.

If you consider the collection examples you've seen so far, you'll realize that all the items on each collection were of the same type. There was a list of strings in which every item was a string. And there was a collection of clients. But collections can store *any* object that is a descendant of **TObject**, and you can mix these objects freely. Naturally, you'll want the classes to have something in common. In fact, you'll often want them to have an abstract base class in common.

As an example, here's a program that puts three different graphical objects into a collection. Then a **forEach** iterator is used to traverse the collection and display each object. This skeleton program is an excellent template for further experiments with collections, graphical objects and polymorphism.

This example uses the Borland graphics library and the BGI drivers, so make sure you include the line `#include <graphics.h>` in your program and link in both TV.LIB and GRAPHICS.LIB. When you run the program, change to the directory that contains the .BGI drivers or modify the call to **initgraph** to specify their location (for example, C:\BORLANDC\BGI).

The abstract base class is defined first.

This is TVGUID20.CPP.

```
class TGraphObject : public TObject
{

public:

   int x, y;
   TGraphObject();
// for this example, the constructor assigns random values to x, y
   virtual void draw() = 0;
// pure virtual function--must be defined in derived classes
};
```

You can see from the following declarations that each derived graphical class can initialize and display itself on the graphics screen (by redefining the pure virtual **draw**). We define a point, a circle, and a rectangle, each derived from **TGraphObject**:

```
class TGraphPoint : public TGraphObject
{

public:

   TGraphPoint();
// for this example, the constructor assigns random values to x, y
   virtual void draw();

};

class TGraphCircle : public TGraphObject
{

public:

   int radius;
   TGraphCircle();
// for this example, the constructor assigns random values to x, y,
// and radius
   virtual void draw();
};

class TGraphRect : public TGraphObject
{

public:

   int width, height;
   TGraphRect();
// for this example, the constructor assigns random values to x, y,
w, // and h
   virtual void draw();
};
```

These three classes all inherit the positional *x* and *y* data members from **TGraphObject**, but they are all different sizes. **TGraphCircle** adds a *radius*, while **TGraphRect** adds a *width* and *height*. Here's the code to make the collection:

```
TNSCollection *list = new TNSCollection(10, 5);   // Create
collection

   for (int i = 1; i < 20; i++)
   {
      switch ( i % 3 )
      {

         case 0:
         {
             TGraphPoint *gp = new TGraphPoint;
             list->insert( gp );
             break;
         }
         case 1:
         {
             TGraphCircle *gc = new TGraphCircle;
             list->insert( gc );
             break;
         }
         case 2:
         {
             TGraphRect *gr = new TGraphRect;
             list->insert( gr );
             break;
         }
      }
   }
...
```

As you can see, the **for** loop inserts 20 graphical objects into the *list* collection. All you know is that each object in *list* is some kind

of derived **TGraphObject**. But once inserted, you'll have no idea whether a given item in the collection is a circle, a point, or a rectangle. Thanks to polymorphism, you don't need to know, since each object contains the data and the code (**draw**) it needs. Just traverse the collection using an iterator member function and have each object display itself:

```
void callDraw( void *p, void * )
{
   ((TGraphObject *)p)->draw();
// Call the appropriate draw member function
}

void drawAll( TNSCollection *c, void * )
{
   c->forEach( &callDraw, 0 );   // Draw each object
}
...
   drawAll( list, 0 );
...
```

This ability of a collection to store different but related objects is one of the powerful cornerstones of object-oriented programming. In Chapter 8, you'll see this same principle of polymorphism applied to streams with equal advantage.

Collections and memory management

A collection object can grow dynamically from the initial size set by its constructor arguments to a maximum size given by *maxCollectionSize*, define in CONFIG.H as follows:

```
const maxCollectionSize = (int)((65536uL - 16)/sizeof( void * ));
```

In PC implementations, this provides a maximum of 16,380 items in any collection, since each element pointer takes four bytes of memory.

No library of dynamic data structures would be complete unless it provided some provision for error detection. If there is not enough memory to initialize a collection, a null pointer is returned.

If memory is not available when adding an element to a collection object, the static member function **TNSCollection::error** is called

and a run-time heap memory error occurs. **error** is implemented as follows:

```
void TNSCollection::error( ccIndex code, ccIndex )
{
   exit(212 - code);
}
```

By default, therefore, **error** returns a run-time error `(212 - code)`. For example, if you try to remove an item at an index beyond the current maximum, **removeAt** will call **error**(1, 0), and exit with error code 211. You may want to override **TNSCollection::error** to provide your own error-reporting or recovery mechanism. The second, currently unused argument, can be used to pass additional data to your own error routines.

Heap availability

You need to pay special attention to heap availability, because the user has much more control of a Turbo Vision program than a traditional program. If the user is the one who controls the adding of objects to a collection (for example, by opening new windows on the desk top), the possibility of a heap error might not be so easy to predict. You may need to take steps to protect the user from a fatal run-time error, with either memory checks of your own when a collection is being used, or a run-time error handler that lets the program recover gracefully.

C H A P T E R

8

Streamable objects

This chapter describes how the Turbo Vision stream manager provides *persistence* for Turbo Vision objects. The various objects that you create during a Turbo Vision 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 inter-process 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 to their former glory by the same application or by other applications. Efficient and safe streamability is available to all Turbo Vision objects. All the obvious candidates (groups, views, **TCollection** derivatives, and resources) are designed as streamable classes, so streaming them is a 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 additional virtual functions, **read**, **streamableName**, and **write**, inherited from a class called **TStreamable**. All streamable classes must be derived, directly or indirectly, from **TStreamable**. **TView** already has **TStreamable** as

one of its bases, so every class derived from **TView** is streamable. Likewise, all **TGroup** derivatives are streamable, including your application objects!

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 C++ streams, you have little more to master. For those new to streams, this chapter presents a brief introduction and establishes some essential terminology before getting into more detail.

The ever-rolling stream

Neither C nor C++ have keywords or predefined operators to handle I/O operations. Both rely on functions built into standard libraries: **stdio** in C and **iostream** in C++. By exploiting the power of object-oriented programming, the C++ 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 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 object-oriented technology, any source (or producer) of data can be provided with extraction member functions and treated as an input stream.

Similarly most sinks (or consumers) of data can be given insertion member functions and treated as output streams.

Streams associated with disk files will typically provide both input and output. 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. Turbo Vision builds on the **iostream** library to provide stream I/O for the more complex objects used in Turbo Vision.

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;
```

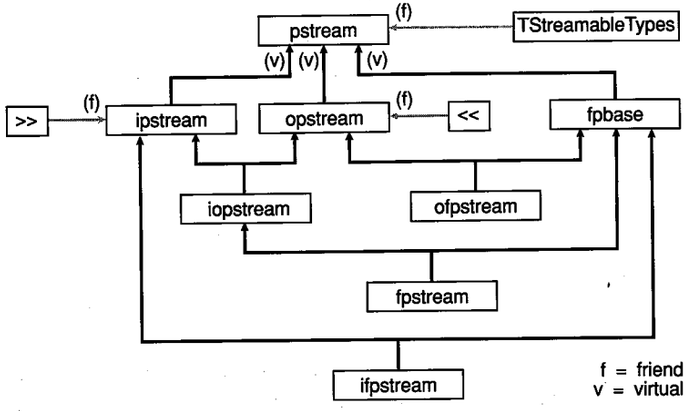
Provided that **<<** and **>>** are suitably overloaded for a given data type and stream class, objects of that type can be written to and read from the appropriate stream objects. Such overloading is "built into" C++ for the standard types, such as **char**, **short**, **int**, **long**, **char ***, **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 Turbo Vision. You can therefore write and read objects with statements such as:

```
os << aTVObject << anotherTVObject;
is >> yetAnotherTVObject;
```

without worrying unduly about the inner details. There are a few straightforward and preliminary chores you need to do when creating and registering your own streamable classes; these are covered later in this chapter. In the previous example, *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 **fpstream**. These correspond to the standard C++ **ifstream**, **ofstream**, and **fstream** classes and have similar data members and member functions. Figure 8.0 illustrates the **pstream** class hierarchy. As is customary in C++, the arrows point *toward* the base class.

Figure 8.1
The **pstream** hierarchy



Overloaded **<<** operators, those used for writing **TView** objects and object pointers to an **opstream**, are defined in VIEWS.H as follows:

```
inline opstream& operator << ( opstream& os, TView& cl );
inline opstream& operator << ( opstream& os, TView* cl );
```

In the first variant, the **TView** 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 **TView** 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, **TView** has a base class **TStreamable**.

Similarly overloaded **<<** and **>>** operators are provided for all the standard streamable classes in Turbo Vision. 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. This may seem circular terminology! What is a streamable class? Read on.

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 Turbo Vision stream manager. In addition to having suitable I/O operators (usually overloaded **<<** and **>>**, but you are free to define your own), a streamable class must have **TStreamable** as a base class somewhere in its hierarchy. A glance at the Turbo Vision class tree (on page 74) reveals that **TView** is multiply derived from **TObject** and **TStreamable**. All classes derived from **TView** therefore inherit the magic of **TStreamable**. All the standard view classes also have overloaded **<<** and **>>** operators, and are therefore streamable.

What of the non-view classes? As you saw in the previous chapter, **TCollection** has **TStreamable** as a base (the other base being **TNSCollection**). In addition, overloaded **<<** and **>>** operators are defined for **TCollection** objects, hence all collection classes derived from **TCollection** are streamable.

Meet the stream manager

A certain amount of background housekeeping is needed when complex objects are saved and retrieved on streams. This is the job of the stream manager. When simple 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 member functions table).

To cope with this and other imponderables, the stream manager maintains a database of streamable classes using **TStreamableTypes**, **TStreamable**, and **TStreamableClass**. The overhead is close to 16 bytes per streamable class (not per instance). 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.

*For more information, see the entries for **TPWrittenObj** and **TPReadObj** in Chapter 13.*

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 using 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 **nptr1* and **nptr2*, only only object will be created and *nptr1* and *nptr2* 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 void *read( ipstream& ) = 0;
    virtual void write( opstream& ) = 0;
```

The job of the writer is to write all the necessary data members 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 are some extracts from **TView::write** and **TGroup::write**:

```
void TView::write( opstream& os )
{
    ushort saveState =
        state & ~( sfActive | sfSelected | sfFocused | sfExposed );

    os << origin << size << cursor
       << growMode << dragMode << helpCtx
       << saveState << options << eventMask;
}
...
void TGroup::write( opstream& os )
{
    ushort index;
    TView::write( os );
    TGroup *ownerSave = owner;
    owner = this;
    int count = indexOf( last );
    os << count;
    forEach( doPut, &os );
    if (current == 0)
       index = 0;
    else
       index = indexOf(current);
    os << index;
    owner = ownerSave;
}
```

The **read** function 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 data members.

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 allocates raw memory and sets vtable pointers. 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 Turbo Vision streamable classes. If you want to create your own streamable classes, you need to provide them.

Streamable class constructors

As explained above, a streamable class whose auto or static objects may be the target of a stream read operation needs a special constructor. This constructor must take a single argument *streamableInit*. You'll find that all the standard streamable classes have such a constructor, which is usually **protected**. The pattern is as follows:

```
class TMyStreamable : public virtual TBase, public TStreamable
{
...
protected:
```

```
    TMyStreamable( StreamableInit s);
...
};
```

The data type **StreamableInit** is an **enum** with the single member *streamableInit*. When you create an object using this constructor,

```
TMyStreamable 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 come to read an object from a stream to the object *str*, the reader for the object does not 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 enumeration 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:

```
TStreamable *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 virtual TBase, public TStreamable
{
...
private:

    virtual const char *streamableName() const;
        { return "TMyStreamable"; }
```

Again, this chore has already been done for all the standard Turbo Vision 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 to track the various class databases it maintains.

Using the stream manager

There are three steps to using the stream manager:

1. Link in the stream manager code
2. Create a suitable stream object
3. Use the stream object

Let's examine each step in detail.

Linking in the stream manager code

Every streamable class must define three functions: **read**, **write** and **build**, designed to handle that class's objects. 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 a single call using the macro **__link**. You need to use **R*ClassName*** with this macro, where ***ClassName*** is the streamable class name, excluding the initial **T**. For example, if you want to

stream the class **TChDirDialog**, your program must contain the statement:

```
_ _link(RChDirDialog);
```

Creating and using a stream object

For the various stream constructors and their arguments, consult Chapter 13.

Creating a **ipstream** or **opstream** stream object just 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:

```
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( const char *filename, int mode = ios::out, int prot =
          filebuf::openprot)
```

has been invoked with the defaults indicated.

A typical read operation might be:

```
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

In Chapter 7, "Collections," you saw how a collection could hold different, but related, objects. The same polymorphic ability applies to streams as well, and they can be used to store an entire collection on disk for retrieval at another time or even by another program. Go back and look at TVGUID20.CPP (on page 166). What more must you do to make that program put the collection on a stream?

The answer is remarkably simple. First, start at the base object, **TGraphObject**, and "teach" it how to store its data (x and y) on a stream. That's what the **write** member function is for. Then, similarly define a new **write** member function for each descendant of **TGraphObject** that adds additional data members (**TGraphCircle** adds a *radius*; **TGraphRec** adds *width* and *height*).

You'll also need to provide **readItem** and **writeItem**. **readItem** reads and return an item from the **ipstream** in a type-safe manner; **writeItem** writes an item to the **opstream**.

And remember to register each class that will be stored. And that's it. The rest is just like normal file I/O: declare a stream variable; create a new stream; put the entire collection on the stream with one simple statement; and close the stream.

Adding write functions

Here are the **write** member functions. Notice that **TGraphPoint** doesn't need one, since it doesn't add any data members to those it inherits from **TGraphObject**.

```
class TGraphObject : public TObject
{
public:
    ...
    virtual void write( opstream& os);
    ...
};

class TGraphCircle : public TGraphObject
{
public:
    int radius;
    ...
    virtual void write( opstream& os);
};

class TGraphRect : public TGraphObject
{
public:
    ...
    int width, height;
    ...
    virtual void write( opstream& os);
};
```

Implementing the **write** is quite straightforward. Each object calls its inherited **write** member function, which stores all the inherited

data. Then it calls the stream's **write** member function to write the additional data.

This is TVGUID21.CPP.

```
void TGraphObject::write( opstream& os)
{
  os << x << y;
}

void TGraphCircle::write( opstream& os)
{
  TGraphObject::write( os );
  os << radius;
}

void TGraphRect::write( opstream& os)
{
  TGraphObject::write( os );
  os << width << height;
```

TVGUID21.CPP will repay careful study. It shows you how to develop your own **read**, **write**, and **build** functions, and how to register your streamable classes.

Saving and restoring the desk top

If the object you save to a stream is the desk top, the desk top will in turn save everything it owns: the entire desk top environment, including all current views.

If you intend to let the user save the desk top, you need to ensure that all possible views have proper **write** and **read** member functions, and that all views are registered, since what the desk top contains at any moment will most likely be up to the user.

You can even go a step further and save and restore whole applications. A **TApplication** object can save and restore itself.

C H A P T E R

9

Resources

Cynics have defined a resource as anything that can be stored in a resource file. To clarify matters, they define a resource file as a place for storing resources! Certainly, in the computer lexicon, the word *resource* has become vague to the point of meaning almost anything. Generally speaking, resources in this book refer to such things as menus, captions, dialog boxes, and so on.

In Turbo Vision, a resource file is a Turbo Vision object that will save objects handed to it, and can then retrieve them by name. Your application can therefore selectively load the objects it uses from a resource rather than invoking constructors. Instead of making your application create and initialize objects, you can have a separate program create all the objects and save them to a resource. The resource can be part of the application .EXE code or it can reside in a separate file.

Turbo Vision has classes called **TResourceCollection** and **TResourceFile**. These give you the capability of storing and retrieving items (usually objects of other classes) to and from streams in a more flexible manner than that offered by the conventional streams discussed in the Chapter 8. The added flexibility stems from the fact that items in a **TResourceFile** object are indexed by name. The name key (an ordinary character string) is supplied when you save the item; you can use it to read back the item by the same or a different program. You don't have to keep track of where the item is in the stream, or how large it is. So a Turbo Vision resource file is very much like an indexed random file.

An added bonus is that a resource file can be appended to an .EXE file. This means that an application can pull in ready-made objects, thereby simplifying your program. Whether the resource is part of the .EXE or supplied as an external file, there are increased opportunities for making your applications more versatile. Many features, such as the language used for captions and help screens, can be separated from the program code. Resource editors can further increase this flexibility.

Resource items are stored in the **fpstream** object owned by each **TResourceFile** object, but for increased efficiency the string keys are maintained in sorted sequence by a **TResourceCollection** object. The latter provides position and size fields mapping to the resources on the stream.

TResourceCollection is a simple derivative of **TStringCollection**, which is a derivative of **TSortedCollection**. **TResourceCollection** adds members to **TStringCollection** to handle the resource indexing. For most applications, you will not be directly involved in the **TResourceCollection** object associated with a resource file. The **TResourceFile** constructor creates a resource collection and **TResourceFile** member functions interact with and maintain the collection. You do have to create a suitable file stream, using one of the **fpstream** constructors with suitable arguments, such as file name, access mode (*ios::in*, *ios::out*, and so on), and protection mode.

The mechanism is fairly simple: a resource file works like a random-access stream, with objects accessed by *keys*, which are simply unique strings identifying the resources. Note that **TNSSortedCollection** has a *duplicates* data member inherited from **TSortedCollection**. This is set *False* by default, meaning that duplicate keys are not allowed. For normal resource applications it makes sense to keep this field *False* and avoid duplicate keys. The default strategy is that if you try to save a resource using an existing key, the new resource will overwrite the previous one.

Unlike other portions of Turbo Vision, you probably won't need or want to change the resource mechanism. As provided, resources are robust and flexible.

Why use resources?

There are a number of advantages to using a resource file.

1. Using resources allows you to customize your application without changing the code. For example, the text of dialog boxes, the labels of menu items, and the colors of views can all be altered within a resource, allowing the appearance of your application to change without anyone having to get inside of it.
2. You can often reduce code size by using resources. Separate programs can be devoted to creating complex objects and saving them in resource files ready for instant retrieval by your main application routines.
3. Using a resource also simplifies maintaining language-specific versions of an application. Your application loads the objects by name, but the language that they display is up to them.
4. If you want to provide versions of an application with differing capabilities, you can, for example, design two sets of menus, one that provides access to all capabilities and another that provides access to only a limited set of functions. That way you don't have to rewrite your code, and you don't have to worry about accidentally stripping out the wrong part of the code. And you can upgrade the program to full functionality by simply providing a new resource, instead of replacing the whole program.

In short, a resource isolates the representation of the objects in your program, and makes it easier to change.

What's in a resource?

Before digging into the details of resources, you might want to make sure you're comfortable with streams and collections, because the resource mechanism uses both of them. You can *use* resources without needing to know just how they work, but the following detail will be useful if you plan to tweak the system.

A **TResourceFile** objects owns both a sorted string collection (a **TResourceCollection** object) and an input/output file stream (an

fpstream object). Two data members of **TResourceFile** establish this object ownership:

```
TResourceCollection *index;
// points to owned collection

fpstream *stream;
// points to owned stream
```

The strings in the collection are keys to objects in the stream. Recall that collections can grow dynamically according to their *limit* and *delta* fields. Each item in **TResourceCollection** is an object of the structure **TResourceItem**:

```
struct TResourceItem
{
   long pos;
   long size;
   char *key;

};
```

The *key* member gives the string key to the object in the associated file stream at the base position *pos* (ignoring certain header information). The *size* member gives the length of the indexed object.

The string collection is created by the **TResourceFile** constructor, and is accessed and maintained by **TResourceFile** members such as **get** and **put**. The file stream, however, must be opened (created and initialized) *before* the resource file is created. In fact, you need to call a **TResourceFile** constructor with an existing **fpstream** pointer as its argument:

```
TResourceFile::TResourceFile( fpstream *aStream );
// create a resource file with given sStream
```

Creating a resource

Creating and using a resource file is a four step process. You need to open a stream, initialize a resource file on that stream, store one or more objects with their keys, and, when finished, close the file stream.

See the GENFORM demonstration source code for a complete, practical example of Turbo Vision resources in action.

The following snippet shows the creation of a simple resource file called MY.REZ used to save a single resource, a status line with the key "Waldo". **TResourceFile::put** takes two arguments: a `TStreamable*` pointer to the streamable item to be saved, and a `const char*` key. Note the `#defines` must precede the `#include<tv.h>` line to ensure proper .H file inclusion and registration of streamable classes.

```
/* in real app, the defines and includes would be in separate header
   file(s) */

#define Uses_TRect
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
/* last two #defines are redundant but harmless--see note below on
   Uses_TResourceCollection */

#define Uses_TResourceFile
#define Uses_TResourceCollection
/* you don't really need #define Uses_TResourceCollection but no harm
   done: TV.H already knows that RESOURCE.H will be included because
   of the Uses_TResourceFile definition
*/

#define Uses_fpstream
...
// add further #defines for each class used
...
// now register the streamable classes to be used:
_ _link(RStatusLine);
_ _link(RResourceCollection);
/* note that TResourceFile is _not_ streamable! Only the resource
   collection streamed
*/
...
// add further #defines to register other streamable classes
...
#include <tv.h>
// this pulls in just the .H files needed

#include <iostream.h>
// more standard includes as needed

...

const char rezFileName[] = "MY.REZ";

cout << "Creating " << rezFileName << endl;

fpstream *ifps;
```

```
ifps = new fpstream( rezFileName, ios::out|ios::binary );
if ( !ifps->good() )
{
    cerr << rezFileName << ": init failed..." << endl;
    exit(1);
}
// check stream OK before calling TResource constructor
// opens fpstream for output/binary mode; default for third prot
// argument is filebuf::openprot
// The error would be handled with a message box in a fuller example
// see LISTDLG.CPP in the demo source code

TResourceFile *myRez;
myRez = new TResourceFile( ifps );
if ( !myRez )
{
    cerr << "Resource file init failed...";
    exit(1);
}

// note: myRez.stream now set to ifps and ready to store...

// Now make a status line for future applications

TRect r( 0, 24, 80, 25 );
TStatusLine *sl;
sl = new TStatusLine( r
       *new TStatusDef( 0, 0xFFFF ) +
         *new TStatusItem( "~Alt-X~ Exit", kbAltX, cmQuit ) +
         *new TStatusItem( "~Alt-F3~ Close", kbAltF3, cmClose )
         );
if (!sl)
{
    cerr << "StatusLine init failed...";
    exit(1);
}

myRez->put( sl, "Waldo");
/* Saves status line in resource with key "Waldo"
   A more complete program might need to check first if the key
exists
   already in myRez and warn user of danger of overwriting an
existing
   resource.
*/

destroy sl;
// assume it's no longer needed!

destroy myRez;
// finished with myRez object now that resources are saved in MY.REZ
...
```


Reading a resource

Retrieving a resource from a resource file is as easy as storing it. You call **TResource::get** with the key of the target resource item as argument. **get** returns a generic `void *` pointer to the item, so type casting is needed before you process the returned object.

The status line resource created in the previous example can be retrieved and used by an application as shown in the following extract. The general idea is that your normal **initStatusLine**, as called via the **TProgInit** constructor, will be replaced by code that reads the "ready-made" status line from the file MY.REZ.

*We've ommitted the **#defines** and **#includes** to avoid clutter.*

```
const char rezFileName[] = "MY.REZ";

class TMyApp: public TApplication
{

public:

   TMyApp();
   static TStatusLine *initStatusLine( TRect );
   ...
};
// omit menu bar and desk top for this example

TMyApp::TMyApp() : TProgInit( &TMyApp::initStatusLine )
{
}

TStatusLine *TMyApp::initStatusLine( TRect )
{

   fpstream *ofps;
   ofps = new fpstream( rezFileName, ios::in|ios::binary );
   if ( !ofps->good() )
       messageBox( "Stream open error", mfError|mfOKButton );
// check ofps non-0 before calling TResource constructor
// opens fpstream for input/binary mode; default for third prot
// argument is filebuf::openprot

   TResourceFile *myRez;
   myRez = new TResourceFile( ofps );
   if ( !myRez )
      messageBox( "Resource file error", mfError | mfOKButton);
   else
      return (TStatusLine *) myRez->get( "Waldo" );
}
...
```

```
int main()
{
   TMyApp waldoApp;
   waldoApp.run;
   return 0;
}
```

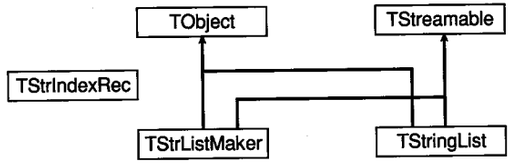
When you read an object off a resource, you need to be aware of the possibility of receiving a 0 pointer. If your index name is invalid (that is, if there is no resource with that key in the file), **get** returns 0. After your resource code is debugged, however, this should no longer be a problem.

You can read an object repeatedly from a resource. It's unlikely that you would want to do so with our example of a status line, but a dialog box, for example, might typically be retrieved many times by a user during the course of an application session. A resource will provide its objects as often as requested.

String lists

In addition to the standard resource mechanism, Turbo Vision provides a pair of specialized objects that handle string lists. Figure 9.1 shows how they fit in the overall scheme of things (as is customary in C++, the arrows point *toward* the base classes).

Figure 9.1
String lists hierarchy



String lists are less flexible than other resources, but they have the merit of being fast and convenient when used as designed.

A *string list* is a simplified resource that allows your program to access resourced strings by number (an **unsigned short**) instead of a key string. This allows a program to store strings out on a resource file for easy customization and internationalization. Do not confuse **TStringList** with **TStringCollection**. The string list is a resource with an associated **ifpstream**, whereas a string collection is simply a sorted collection of strings with no attached stream.

For example, the Borland Programmer's Platform (IDE) uses string lists for all its error messages. This means the program can simply call for an error message by number, and different versions in different countries will find different strings in their resources. You can see the difference between string lists and normal resources by comparing the classes **TStrIndexRec** and **TResourceItem**, used to index the resourced items:

```
struct TResourceItem
{
   long pos;
   long size;
   char *key;
};
class TStrIndexRec
{
public:
   ushort key;
   ushort count;
   ushort offset;
};
```

*For an explanation of the build constructor, see Chapter 8 and the entry for **TStreamable** in Chapter 13.*

TStringList has only the build constructor, the one taking *streamableInit* as argument. This is because string lists exist only on resource files: they can be accessed from a resource file but not directly created elsewhere. String lists are essentially read-only resources, so they have a **get** member function but no **put**.

Making string lists

__TStringList__ is a streamable class used to access the strings. __TStrListMaker__ creates the string list.

In addition to **TStringList**, you'll need to use **TStrListMaker**. **TStrListMaker** creates a string list on a resource file for use with **TStringList**. In contrast to the string list, which is read-only, the string list maker is write-only. **TStrListMaker**, therefore has a **put** method but no **get**. All you can do with a string list maker is initialize a string list, write strings onto it, and store the resulting list on a stream. The following extract illustrates the creation and use of a string list:

```
#define Uses_TStringList
#define Uses_TStrListMaker
...
#include <tv.h>
__link(RStringList);
...
TStrListMaker myStrListMaker( strSize, indexSize );
// creates an array of strings each of size strSize.
// creates an array indexSize objects of type TStrIndexRec,
// each of which holds data members: key, count, and offset.
```

```
// The index data member is set to point at this array of
TStrIndexRec // objects.

myStrListMaker.put( 1, "Pass, Do not GO!" );
myStrListMaker.put( 2, "Collect $500!" );
...
opfstream ops( "STR.LST" );
ops << myStrListMaker;
...
TStringList *myStringList;
// sets indexSize, index and basePos to 0

char legend[maxLeg];

ipfstream ifs( "STR.LST" );
ifs >> myStringList;

myStringList->get( legend, 1);

cout << "Legend 1: " << legend << endl;
// displays "Pass, Do not GO!"
```


C H A P T E R

10

Hints and tips

This chapter contains a few additional suggestions on how to use Turbo Vision more effectively. Because object-oriented programming and event-driven programming are fairly new concepts to even experienced programmers, we want to try to provide some guidance in using these new models.

Debugging Turbo Vision applications

If you have tried stepping or tracing through any of the example programs provided in this cookbook, you have probably noticed that you don't get very far. Because Turbo Vision programs are event-driven, much (or even most) of the program's time is spent running through a rather tight loop in **TGroup::execute**, waiting for an event to occur. As a result, stepping and tracing is not very helpful at that point.

The key to debugging Turbo Vision applications is breakpoints, breakpoints, and breakpoints.

Let's look at how well-placed breakpoints can help you find problems in Turbo Vision programs.

It doesn't get there

One problem in debugging your application might be that some portion of your code is not being executed. For example, you might click on a status line item or select a menu option that you *know* is supposed to bring up a window...but it doesn't.

Your normal instinct might tell you to step through your program until you get to that command, and then figure out where execution *does* go instead of where you expected. But if you try it, it doesn't help. You step, and you end up right back where you were.

The best approach in this situation is to set a breakpoint in the **handleEvent** method that should be calling the code that isn't getting executed. Set the breakpoint at the beginning of the **handleEvent** method and when it breaks, inspect the event record that's being processed to make sure it's the event you expected. At this point you can also start stepping through your code, because the **handleEvent** and any code responding to your own commands will be code you have written, and therefore code you can trace through.

Hiding behind a mask

Keep in mind, however, that there are a couple of reasons why a view may never get to see the event you intend it to handle. The first and simplest mistake is leaving an event type out of your object's event mask. If you haven't told an object that it is allowed to handle a certain kind of event, it won't even look at those events!

Stolen events

A second possibility you need to consider is that some other object is "stealing" the event. That is, the event is being handled and cleared by some object other than the one you intended to give it to.

There are a couple of things which could cause this. The first is duplicate command declarations: if two commands have been assigned the same constant value, they could be handled interchangeably. This is why it is *crucial* to keep track of which constants you have assigned which values, particularly in a situation when you are reusing code modules.

Another possible is duplicate command labels, particularly in reused code. Thus, if you assign a command *cmJump*, and there is

a **handleEvent** method in some other object that already responds to a command *cmJump* that you have forgotten about and never deleted, you could have conflicts. Always check to see if some other object is handling the events that seem to get "lost."

Blame your parents

Finally, check whether the event is being handled in a call to the object's ancestor. Often, the **handleEvent** method of a derived type will rely on the event handler of its ancestor to deal with most events, and it may be handling one that you didn't expect. Try trapping the event before the call to the ancestor's **handleEvent**.

It doesn't do what I expect

Perhaps your window does show up, but it displays garbage, or something other than what you expected. This indicates that the event is being handled properly, but the code that responds to the event is either incorrect or perhaps overridden. In this case, it is best to set a breakpoint in the function that gets called when the event occurs. Once execution breaks, you can step or trace through your code normally.

It hangs

Hang bugs are among the most difficult to track down, but they *can* be found. First you might try some combination of the breakpointing methods suggested previously to narrow down just where the hang occurs. The second thing to look for is pointers being disposed of twice. This can happen when an object is disposed of by its owner, and then you try to dispose of it directly.

Hangs can also be caused by such things as reading stream data into the wrong type of object and incorrectly typecasting data from collections.

Porting applications to Turbo Vision

If you want to port an existing application to Turbo Vision, your first inclination might be to try to port the Turbo Vision interface into the application, or to put a Turbo Vision layer on top of your application. This will be an exercise in frustration. Turbo Vision

applications are event-driven, and most existing applications will not shift easily, if at all, to that paradigm.

Scavenge your old code

There is an easier way. By now, you know that the essence of programming a specific application in Turbo Vision is concentrated in the application's **draw** and **handleEvent** methods. The better approach to porting an existing application is first to write a Turbo Vision interface that parallels your existing one, and then scavenge your old code into your new application. Most of the scavenged code will end up in new view's **draw** and **handleEvent** methods.

You need to spend some time thinking about the essence of your application, so you can divide your interface code from the code that carries out the work of your application. This can be difficult, because you have to think differently about your application.

The job of porting will involve some rewriting to teach the new objects how to represent themselves, but it will also involve a lot of throwing away of old interface code.

When you port an application, you may be surprised to discover how much of your code is dedicated to handling the user interface. When you let Turbo Vision work for you, a lot of the user interface work you did before will simply disappear.

We discovered how rewarding this can be when we ported Borland's integrated environment to Turbo Vision. We scavenged the compiler, the editor, the debugger—all the various engines—from the old user interface, and brought them into a user interface written in Turbo Vision.

Rethink your organization

Programming in this new paradigm takes some getting used to. In traditional programming, we tend to think of the program from the perspective of the code. We are the code, and the data is "out there," something on which we operate. At first glance, we might be tempted to organize a program such as an program development environment around an editor object. After all, that's what you're doing most of the time in the environment, editing. The editor would edit, and at intervals, it would call the compiler.

But we need to make some shifts in perspective to use the true power of object-oriented programming. It makes more sense in the integrated environment to make the application itself the organizing object. When it's time to edit, the application calls up an editor. When it's time to compile, the application brings up the compiler, initializes it, and tells it what files to compile.

If the compiler hits an error, how is the user returned to the point of error in the source code? The application calls the compiler, and it gets a result back from it. If the compiler returns an error result, it also returns a file name and a line number. The application looks to see if it already has an editor open for that file, and if not, it opens it. It passes the error information, including the line number, to the editor and constructs an error message string for the editor.

There's no reason for the editor to know anything about a compiler, or the compiler to know about an editor. The center of it all is the application itself. It's the application that needs the editor and the application that needs the compiler. After all, what is an application but something that binds things together? If we had continued to look on the application as just a lump of data that should be "out there" somewhere, and we might have been tempted to put the center of the application elsewhere. We would then have had to carry a burden of excessive and strained communications among parts of the program.

All in all, the job of writing the integrated environment in Turbo Vision took a fraction of the time that writing the environment from scratch would have taken. We look forward to you discovering the same strengths when you write your next application.

Using bitmapped fields

C veterans can skip this section

Turbo Vision uses many *bitmapped* data members. That is, they use the individual bits of a byte or word to indicate different properties. The individual bits are usually called *flags*, since by being set (equal to 1) or cleared (equal to 0), they indicate whether the designated property is activated.

For example, each view has a bitmapped field called *options*. Each of the individual bits in the word has a different meaning to

Turbo Vision. Definitions of the bits in the *options* word are given in Chapter 13.

Flag values

Bit positions are assigned mnemonics. For example, the fourth bit is called *ofFramed*. If the *ofFramed* bit is set to 1, it means the view has a visible frame around it. If the bit is a 0, the view has no frame.

You don't have to worry about the actual flag bit values unless you plan to define your own, and even then, your only concern is that your definitions be unique. For instance, the six highest bits in the *options* word are presently undefined by Turbo Vision. You may define any of them to mean anything to the views you derive.

Bit masks

A *mask* is simply a shorthand way of dealing with a group of bit flags together. For example, Turbo Vision defines masks for different kinds of events. The *evMouse* mask simply contains all four bits that designate different kinds of mouse events, so if a view needs to check for mouse events, it can compare the event type to see if it's in the mask, rather than having to check for each of the individual kinds of mouse events.

Bitwise operations

The bitwise operators of C and C++ are used in various combinations for flag manipulation and testing. Here are a few examples.

Setting a bit

To set a bit, use the | operator. For example, the following line sets the *ofPostProcess* bit in the *options* field of a button called *myButton*:

```
myButton.options |= ofPostProcess;
// same as myButton.options = (myButton.options | ofPostProcess);
```

Never use **+** to set bits. For example,

***Don't** do this!*

```
myButton.options += ofPostProcess;
```

only works if the *ofPostProcess* bit was 0. If the bit was set, the binary add carries over into the next bit (*ofBuffered*), setting or clearing it, depending on whether it was clear or set to start with. And so on, up the flag!

You can set several bits with one statement. The following code sets two grow mode flags in a scrolling view called *myScroller*:

```
myScroller.growMode |= (gfGrowHiX | gfGrowHiY);
```

Clearing a bit

You clear a bit wih the **&** (bitwise AND) and ~ (bitwise negate) operators. The following line clears the *dmLimitLoX* bit in the *dragMode* field of a label called *aLabel*:

```
aLabel.dragMode &= ~dmLimitLoX;
```

As with setting bits, multiple bits can be cleared in a single operation.

Toggling a bit

Bits can be toggled (1 to 0, 0 to 1) using the **^** (bitwise XOR) operator:

```
myButton.options ^= ofPreProcess;
// toggle the ofPreProcess flag
myView ^= (growModeHiX | growModeHiY);
// toggle both flags
```

Checking bits

A view often needs to check if a certain flag bit is set. The **&** operation is ideal for this. For example, to see if the window *aWindow* may be tiled by the desk top, you check the *ofTileable* option flag like this:

```
if (aWindow.options & ofTileable) { ... }
```

Using masks

You can also use **&** to check if more than one masked bit is set. For example, to test an event record for any type of mouse event, you could write

```
if (event.what & evMouse) { ... }
```

Summary

The following list summarizes the bitmap operations:

Setting a bit:

```
field |= flag;
```

Clearing a bit:

```
field &= ~flag;
```

Toggling a bit:

```
field ^= flag;
```

Checking if a flag is set:

```
if (field & flag) { ... }
```

Checking if a flag is in a mask:

```
if (flag & mask) { ... }
```


PART 3

Turbo Vision Reference

C H A P T E R

11

How to use the reference chapters

The Turbo Vision reference describes all the *standard* classes and member functions in the Turbo Vision hierarchy together with the mnemonic identifiers and miscellaneous constants and records needed to develop Turbo Vision applications. It is not intended as a tutorial.

By their nature, complex libraries of classes like those in Turbo Vision have a multitude of components. In order to avoid endless repetition of material, we have put as much complete information into the alphabetical lookup chapters (Chapters 13 through 16) as possible, along with other, less detailed material that allows you to see Turbo Vision's components in their hierarchical and physical relationships, with references to the more detailed information.

How to find what you want

Chapter 12, "Header file cross-reference" describes the various header files that comprise Turbo Vision. It includes lists of all the types, constants, variables, and functions declared in each header file.

Chapter 13, "Class reference," is a description of all the Turbo Vision standard class types, including all their data members and member functions. The classes are arranged in alphabetical order,

and within each class, the data members and member functions are also listed in alphabetical order.

Chapter 14, "The editor classes," and Chapter 15, "Implementing standard dialog boxes," describe some extensions to the standard Turbo Vision classes that provide a simple editor and some standard dialog box capabilities.

Chapter 16, "Global reference," lists (in alphabetical order) and describes all the global constants, variables, and functions in Turbo Vision. In general, if it's not a class or a part of a class, you'll find it listed here.

Keep in mind that each class description in Chapters 13 through 15 only covers the aspects of each class that are particular to it. Most of the classes will have data members and member functions inherited from other classes. Thus, if you want to find a member function for a particular class, check that class first. If you don't find the member function listed under the appropriate heading in the description of that class, check its immediate base class or the index. There is a diagram at the beginning of the entry for each class that depicts its relationships to its base classes and immediate derived classes.

Objects in general

Remember that each object (apart from the base object *TObject*, and the two special objects *TPoint* and *TRect*) inherits the data members and member functions of its parent object. New objects that you derive will also inherit their base class's member functions and data members. Many of the standard objects have *abstract* member functions which *must* be overridden by your derived objects. Other member functions are marked *virtual*, meaning that you will normally want to override them. There are other member functions that provide useful default actions in the absence of overrides.

Naming conventions

All the standard Turbo Vision object types have a set of names using a mnemonic set of prefixes. The first letter of the identifier tells you whether you are dealing with the object type, its stream registration structure, or its color palette.

- Object types start with *T*: **TObject**
- Stream registration records start with *R*: *RObject*
- Color palettes start with *cp*: *cpObject*
- Member functions and data members have initial lowercase letters with subsequent uppercase letters for each whole word within the member name: **handleEvent**, *hScrollBar*

All Turbo Vision constants have two-letter mnemonic prefixes that indicate their usage.

Table 11.1
Turbo Vision constant prefixes

Prefix	Meaning	Example
ap	Application palette	*apColor*
bf	Button flag	*bfNormal*
cm	Command	*cmQuit*
co	Collection code	*coOverFlow*
dm	Drag mode	*dmDragGrow*
ev	Event constant	*evMouseDown*
gf	Grow mode flag	*gfGrowLoX*
hc	Help context	*hcNoContent*
kb	Keyboard constant	*kbAltX*
mb	Mouse button	*mbLeftButton*
mf	Message box	*mfWarningch*
of	Option flag	*ofTopSelect*
sb	Scroll bar	*sbLeftArrow*
sf	State flag	*sfVisible*
wf	Window flag	*wfMove*
wn	Window numbers	*wnNoNumber*
wp	Window palette	*wpBlueWindow*

C H A P T E R

12

Header file cross-reference

Turbo Vision offers a variety of header files covering related groups of standard Turbo Vision classes as well as some sets of useful derived classes offered as instructional examples. Many of the latter offer specialized classes that you can incorporate into your own applications with little change.

This chapter gives a quick cross-reference to the classes declared in each of the header files provided with Turbo Vision. It also provides some class hierarchy diagrams that can help place the classes in Chapters 13 through 15 in perspective. For a full explanation of what each class does, and how, please consult Chapters 13 through 15. These chapters list all classes alphabetically, with data members and member functions listed alphabetically within each class.

Although the header names are mnemonic, it may not always be obvious where a particular class is defined. Tables 12.1 through 12.3 summarize the various Turbo Vision headers. Following these tables, you'll find the contents of each header file in Table 12.1 listed in more detail.

Table 12.1
Standard header files

Header file	Contents
APP	The major application classes **TProgram** and **TApplication**
BUFFERS	Video memory handling
CONFIG	Miscellaneous system-wide parameters
DIALOGS	Dialog box, control, and history classes
DRAWBUF	Draw buffer classes
MENUS	Menu and status line classes

Table 12.1: Standard header files (continued)

MSGBOX	Globals for message and input boxes
OBJECTS	Basic non-view classes: point, rectangle and collections
RESOURCE	Resource and related classes
SYSTEM	Classes for mouse and keyboard event handling
TEXTVIEW	Specialized classes for text devices
TKEYS	Keyboard constants
TOBJSTRM	Stream classes
TTYPES	Basic typedefs and constants
TV	Master header for #include control
TVOBJS	**TObject** and non-streamable collections
UTIL	Miscellaneous globals
VIEWS	Classes for views, groups, windows, frames, scroll bars

These are the special extensions to the standard classes mentioned previously.

Table 12.2
Special header files

Header file	Contents
COLORSEL	Palette selection classes; defined in Chapter 13
EDITORS	Specialized classes for editors; defined in Chapter 14
STDDLG	Specialized dialog and input lines; defined in Chapter 15

These classes are included to demonstrate the use of the standard classes to accomplish certain special-purpose tasks.

Table 12.3
Demonstration header files

Header file	Contents
ASCII	ASCII chart demonstration
CALC	Calculator demonstration
CALENDAR	Calendar demonstration
GADGETS	Clock viewer and heap viewer demonstrations
MOUSEDLG	A mouse dialog class
FILEVIEW	File viewer classes
PUZZLE	Puzzle demonstration
TVBGI	BGI demonstrations

The APP header file

The APP header file defines the classes **TApplication**, **TBackground**, **TDeskInit**, **TDeskTop**, **TProgInit**, and **TProgram**.

The BUFFERS header file

The BUFFERS header file defines the classes **TBufListEntry**, **TVideoBuf**, and **TVMemMgr**. It also defines the global constant DEFAULT_SAFETY_POOL_SIZE and sets its initial value to 4,096.

The CONFIG header file

The CONFIG header file is for internal use only. It defines the following constants:

Table 12.4
CONFIG.H constant definitions

Constant	Definition
eventQSize	16
maxCollectionSize	(int)((65536uL – 16)/sizeof(void *))
maxFindStrLen	80
maxReplaceStrLen	80
maxViewWidth	132

The DIALOGS header file

The DIALOGS header file defines the constant *cmRecordHistory*, several button type constants, and the following classes related to dialog boxes:

TButton	**THistory**	**TListBox**
TCheckBoxes	**THistoryViewer**	**TParamText**
TCluster	**THistoryWindow**	**TRadioButtons**
TDialog	**TInputLine**	**TSItem**
THistInit	**TLabel**	**TStaticText**

Table 12.5
DIALOGS.H constant definitions

Constant	Definition
bfNormal	0x00
bfDefault	0x01
bfLeftJust	0x02
bfBroadcast	0x04
cmRecordHistory	60

The DRAWBUF header file

The DRAWBUF header file defines the class **TDrawBuffer** and the macros **loByte** and **hiByte**, useful for selecting the character and attribute bytes from a word.

The MENUS header file

The MENUS header file offers overloaded + operators for **TSubMenu**, **TMenuItem**, **TStatusDef**, and **TStatusItem**. It also defines the following classes:

TMenuBar	**TMenuItem**	**TStatusDef**	**TStatusLine**
TMenuBox	**TMenuView**	**TStatusItem**	**TSubMenu**

The MSGBOX header file

The MSGBOX header file defines the following:

Table 12.6
MSGBOX.H definitions

Item	Value	Meaning
Global functions		
messageBox		
messageBoxRect		
inputBox		
inputBoxRect		
Message box constants		
mfWarning	0x0000	Display a Warning box
mfError	0x0001	Display an Error box
mfInformation	0x0002	Display an Information Box
mfConfirmation	0x0003	Display a Confirmation Box
Message box button flags		*Puts into dialog box:*
mfYesButton	0x0100	Yes button
mfNoButton	0x0200	No button
mfOKButton	0x0400	OK button
mfCancelButton	0x0800	Cancel button
mfYesNoCancel	*mfYesButton*	
	| *mfNoButton*	
	| *mfCancelButton*	Standard Yes, No, Cancel dialog
mfOKCancel	*mfOKButton*	
	| *mfCancelButton*	Standard OK, Cancel dialog

The OBJECTS header file

The OBJECTS header file defines the classes **TCollection**, **TPoint**, **TRect**, and **TSortedCollection**.

The RESOURCE header file

The RESOURCE header file defines the classes **TResource-Collection**, **TResourceFile**, **TStrIndexRec**, **TStringCollection**, **TStringList**, **TStrListMaker**, and the structure **TResourceItem**.

The SYSTEM header file

The SYSTEM header file defines the following classes:

Int11trap	**TMouse**
TDisplay	**TScreen**
TEventQueue	**TSystemError**
THWMouse	

It also defines the structures **CharScanType**, **MessageEvent**, **MouseEventType**, **and TEvent**.

Finally, it defines the following event codes and external variables:

Table 12.7
SYSTEM.H variable values

Item	Value	Item	Value
Event code (constant)			
evMouseDown	0x0001;	*evKeyDown*	0x0010;
evMouseUp	0x0002;	*evCommand*	0x0100;
evMouseMove	0x0004;	*evBroadcast*	0x0200;
evMouseAuto	0x0008;		
Event mask (constant)			
evNothing	0x0000;	*evKeyboard*	0x0010;
evMouse	0x000f;	*evMessage*	0xFF00;
External variables (extern ushort)			
biosSeg			
colrSeg			
monoSeg			
Mouse button state mask (constant)			
mbLeftButton	0x01;		
mbRightButton	0x02;		

The TEXTVIEW header file

The TEXTVIEW header file defines the classes **TTextDevice** and **TTerminal.**

The TKEYS header file

The TKEYS header file defines all of the following:

Table 12.8
Keyboard state and shift masks

Masks	Value	Masks	Value
kbAltShift	0x0008	kbLeftShift	0x0002
kbCapsState	0x0040	kbNumState	0x0020
kbCtrlShift	0x0004	kbRightShift	0x0001
kbInsState	0x0080	kbScrollState	0x0010

Table 12.9: Key codes

Key code	Value	Key code	Value	Key code	Value
kbAlt0	0x8100	kbAltX	0x2d00	kbCtrlLeft	0x7300
kbAlt1	0x7800	kbAltY	0x1500	kbCtrlPgDn	0x7600
kbAlt2	0x7900	kbAltZ	0x2c00	kbCtrlPgUp	0x8400
kbAlt3	0x7a00	kbBack	0x0e08	kbCtrlPrtSc	0x7200
kbAlt4	0x7b00	kbCtrlA	0x0001	kbCtrlRight	0x7400
kbAlt5	0x7c00	kbCtrlB	0x0002	kbDel	0x5300
kbAlt6	0x7d00	kbCtrlC	0x0003	kbDown	0x5000
kbAlt7	0x7e00	kbCtrlD	0x0004	kbEnd	0x4f00
kbAlt8	0x7f00	kbCtrlE	0x0005	kbEnter	0x1c0d
kbAlt9	0x8000	kbCtrlF	0x0006	kbEsc	0x011b
kbAltA	0x1e00	kbCtrlG	0x0007	kbF1	0x3b00
kbAltB	0x3000	kbCtrlH	0x0008	kbF2	0x3c00
kbAltC	0x2e00	kbCtrlI	0x0009	kbF3	0x3d00
kbAltD	0x2000	kbCtrlJ	0x000a	kbF4	0x3e00
kbAltE	0x1200	kbCtrlK	0x000b	kbF5	0x3f00
kbAltEqual	0x8300	kbCtrlL	0x000c	kbF6	0x4000
kbAltF	0x2100	kbCtrlM	0x000d	kbF7	0x4100
kbAltF1	0x6800	kbCtrlN	0x000e	kbF8	0x4200
kbAltF10	0x7100	kbCtrlO	0x000f	kbF9	0x4300
kbAltF2	0x6900	kbCtrlP	0x0010	kbF10	0x4400
kbAltF3	0x6a00	kbCtrlQ	0x0011	kbGrayMinus	0x4a2d
kbAltF4	0x6b00	kbCtrlR	0x0012	kbGrayPlus	0x4e2b
kbAltF5	0x6c00	kbCtrlS	0x0013	kbHome	0x4700
kbAltF6	0x6d00	kbCtrlT	0x0014	kbIns	0x5200
kbAltF7	0x6e00	kbCtrlU	0x0015	kbLeft	0x4b00
kbAltF8	0x6f00	kbCtrlV	0x0016	kbNoKey	0x0000
kbAltF9	0x7000	kbCtrlW	0x0017	kbRight	0x4d00
kbAltG	0x2200	kbCtrlX	0x0018	kbPgDn	0x5100
kbAltH	0x2300	kbCtrlY	0x0019	kbPgUp	0x4900
kbAltI	0x1700	kbCtrlZ	0x001a	kbRight	0x4d00
kbAltJ	0x2400	kbCtrlBack	0x0e7f	kbShiftDel	0x0700
kbAltK	0x2500	kbCtrlDel	0x0600	kbShiftF1	0x5400
kbAltL	0x2600	kbCtrlEnd	0x7500	kbShiftF2	0x5500
kbAltM	0x3200	kbCtrlEnter	0x1c0a	kbShiftF3	0x5600
kbAltMinus	0x8200	kbCtrlF1	0x5e00	kbShiftF4	0x5700
kbAltN	0x3100	kbCtrlF2	0x5f00	kbShiftF5	0x5800
kbAltO	0x1800	kbCtrlF3	0x6000	kbShiftF6	0x5900
kbAltP	0x1900	kbCtrlF4	0x6100	kbShiftF7	0x5a00
kbAltQ	0x1000	kbCtrlF5	0x6200	kbShiftF8	0x5b00
kbAltR	0x1300	kbCtrlF6	0x6300	kbShiftF9	0x5c00
kbAltS	0x1f00	kbCtrlF7	0x6400	kbShiftF10	0x5d00
kbAltSpace	0x0200	kbCtrlF8	0x6500	kbShiftIns	0x0500
kbAltT	0x1400	kbCtrlF9	0x6600	kbShiftTab	0x0f00
kbAltU	0x1600	kbCtrlF10	0x6700	kbTab	0x0f09
kbAltV	0x2f00	kbCtrlHome	0x7700	kbUp	0x4800
kbAltW	0x1100	kbCtrlIns	0x0400		

The TOBJSTRM header file

The TOBJSTRM header file defines the following classes:

fpbase	**pstream**
fpstream	**TPReadObjects**
ifpstream	**TPWObj**
iopstream	**TPWrittenObjects**
ipstream	**TStreamable**
ofpstream	**TStreamableClass**
opstream	**TStreamableTypes**

This header file also defines BUILDER and the macros **__link** and **__DELTA**.

The TTYPES header file

The TTYPES header file defines and sets the following:

- const ccNotFound = –1;
- const char EOS = '\0';
- enum Boolean { False, True };
- enum StreamableInit { streamableInit };
- extern const uchar specialChars[];
- typedef Boolean (*ccTestFunc)(void *, void *);
- typedef int ccIndex;
- typedef unsigned char uchar;
- typedef unsigned short ushort;
- typedef void (*ccAppFunc)(void *, void *);

The TV header file

TV.H is known as the "include control" header file.

The TV header file ensures that all the necessary header files are **#included** in your application code effortlessly and without duplication. TV.H always **#includes** the essential Turbo Vision headers such as CONFIG.H and UTIL.H. Other header files are included conditionally by means of the **#if defined** directive.

TV.H tests whether the identifier *Uses_**ClassName*** (where ***ClassName*** is a class name or a class registration name) has been previously defined. Depending on the result, the header files for class ***ClassName*** and its base classes are included automatically. If your program uses class ***ClassName***, you simply include the code `#define Uses_ClassName` in your program or header, followed by an `#include <tv.h>` statement. This automates the inclusion process for that class and its base classes. For example, suppose your program starts with

```
#define Uses_TApplication
#include <tv.h>
```

TV.H tests as follows:

```
#if defined( Uses_TApplication )
#define Uses_TProgram
#define _ _INC_APP_H
#endif
```

Note that **TProgram** is the base class for **TApplication**. Later on, TV.H tests _ _INC_APP_H as follows:

```
#if defined( _ _INC_APP_H )
#include <App.h>
```

The end result is that APP.H, containing the class declarations for **TProgram**, **TProgInit**, and **TApplication**, is included in your program. No harm is done if you **#define** both base and derived classes:

```
#define Uses_TProgram
#define Uses_TApplication
#include <tv.h>
```

The stream registration of streamable classes is accomplished by the use of the **__link** macro. For example, to register **TEditWindow**, your program should use the statement

```
#include <tv.h>
_ _link(REditWindow);
```

The **__link** macro creates the **extern** declaration:

```
extern TStreamableClass REditWindow;
```

which ensures the proper linkage of the stream read and write functions. (See also the description for **TStreamableClass** in Chapter 13.)

Important! When you create your own classes and include files, you can develop your own include control headers based on the TV.H strategy. However, we advise strongly against modifying TV.H,

since it encapsulates the standard Turbo Vision hierarchy: changes to TV.H may have subtly disconcerting, or plain catastrophic, effects.

The TVOBJS header file

The TVOBJS header file defines these three core classes: **TObject**, **TNSCollection**, and **TNSSortedCollection**.

The VIEWS header file

The VIEWS header file defines these classes and one structure:

TCommandSet
TFrame
TGroup
TListViewer
TPalette
TScrollBar
TScroller
TView
TWindow
TWindowInit
write_args (structure)

It also defines these standard command codes to be equal to the given values.

Table 12.10
VIEWS.H code values

Codes	Value	Codes	Value
cmValid	0	*cmZoom*	5
cmQuit	1	*cmResize*	6
cmError	2	*cmNext*	7
cmMenu	3	*cmPrev*	8
cmClose	4	*cmHelp*	9

It also defines these various masks, commands, codes, and options.

Table 12.11
VIEWS.H values

Variable	Value	Variable	Value
TDialog standard commands			
cmOk	10	*cmNo*	13
cmCancel	11	*cmDefault*	14
cmYes	12		
TView *state* masks:			
sfVisible	0x001	*sfFocused*	0x040
sfCursorVis	0x002	*sfDragging*	0x080
sfCursorIns	0x004	*sfDisabled*	0x100
sfShadow	0x008	*sfModal*	0x200
sfActive	0x010	*sfDefault*	0x400
sfSelected	0x020	*sfExposed*	0x800
TView *options* masks:			
ofSelectable	0x001	*ofBuffered*	0x040
ofTopSelect	0x002	*ofTileable*	0x080
ofFirstClick	0x004	*ofCenterX*	0x100
ofFramed	0x008	*ofCenterY*	0x200
ofPreProcess	0x010	*ofCentered*	0x300
ofPostProcess	0x020		
TView *growMode* masks			
gfGrowLoX	0x01	*gfGrowHiY*	0x08
gfGrowLoY	0x02	*gfGrowAll*	0x0f
gfGrowHiX	0x04	*gfGrowRel*	0x10
TView *dragMode* masks:			
dmDragMove	0x01	*dmLimitHiX*	0x40
dmDragGrow	0x02	*dmLimitHiY*	0x80
dmLimitLoX	0x10	*dmLimitAll*	dmLimitLoX | dmLimitLoY | dmLimitHiX | dmLimitHiY
dmLimitLoY	0x20		
TView help context codes:			
hcNoContext	0		
hcDragging	1		
TScrollBar part codes:			
sbLeftArrow	0	*sbDownArrow*	5
sbRightArrow	1	*sbPageUp*	6
sbPageLeft	2	*sbPageDown*	7
sbPageRight	3	*sbIndicator*	8
sbUpArrow	4		
TScrollBar options for **TWindow::StandardScrollBar**:			
sbHorizontal	0x000		
sbVertical	0x001		
sbHandleKeyboard	0x002		
TWindow *flags* masks:			
wfMove	0x01		
wfGrow	0x02		
wfClose	0x04		
wfZoom	0x08		
TWindow number constants:			
wnNoNumber	0		

Table 12.11: VIEWS.H values (continued)

TWindow palette entries:

wpBlueWindow	0
wpCyanWindow	1
wpGrayWindow	2

Application command codes:

cmCut	20	*cmClear*	24
cmCopy	21	*cmTile*	25
cmPaste	22	*cmCascade*	26
cmUndo	23		

Standard messages:

cmReceivedFocus	50
cmReleasedFocus	51
cmCommandSetChanged	52

TScrollBar messages:

cmScrollBarChanged	53
cmScrollBarClicked	54

TWindow *select* messages:

cmSelectWindowNum	55

TListViewer messages:

cmListItemSelected	56

Event masks:

positionalEvents	evMouse
focusedEvents	evKeyboard | evCommand

And a lone typedef:

```
typedef char TScrollChars[5];
```

Class hierarchy diagrams

This section presents those class hierarchy diagrams that haven't already been presented in other chapters. You can find the major overview diagrams on pages 74 and 75. Classes that are represented in these diagrams but aren't shown connected to anything are related but not part of the hierarchy but are related to the classes they are near. As is customary in C++, the arrows point *toward* the base class.

Figure 12.1
Viewers and dialog boxes

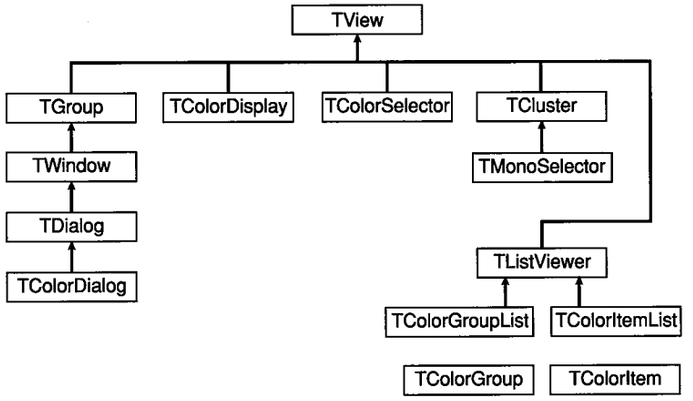


Figure 12.2
Streamable classes

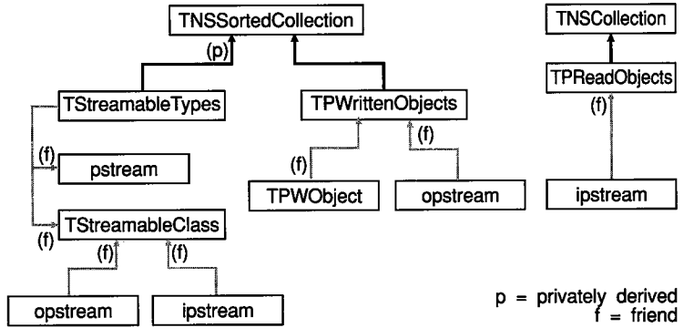
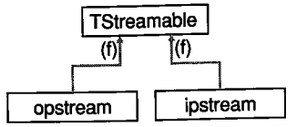


Figure 12.3
TStreamable friends



C H A P T E R

13

Class reference

This chapter contains an alphabetical listing of all the standard Turbo Vision C++ classes, with explanations of their general purposes and usage and their data members, member functions, and color palettes. You'll also find a description of the useful structures defined in Turbo Vision. Chapter 12 provides a cross-reference to the classes in this chapter and in Chapters 14 and 15.

Using this chapter

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 duplicate all that information endlessly, this chapter only documents data members and member functions that are *new* or *changed* for a particular class.

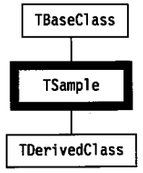
To save you some hunting, all members are listed in the index.

For example, if you want to know about the *owner* data member of a **TLabel** class, you could either check the index for *owner*, or you might look under **TLabel**'s data members. Since *owner* is inherited, you won't find it listed under **TLabel**. You would then check **TLabel**'s immediate ancestor (base class) in the hierarchy, **TStaticText**. Again, *owner* will not be listed. You would next check **TStaticText**'s immediate ancestor, **TView**. There you will find complete information about *owner*, which is inherited unchanged by **TLabel**.

Each class's entry in this chapter has a graphical representation of the class's base classes and immediate derived classes, so it should be easy for you to find the classes from which members are inherited.

Each class's entry is laid out in the following format:

TSample class — header.h



This section gives a brief synopsis of the class. In some cases this is the only information needed.

Data members

This section lists all data members for each class, alphabetically, showing the data member's declaration and an explanation of its use.

Data members and member functions can be public, private, or protected. If they are protected, we have indicated this near the code. If there is no such indication, you can assume the member is public. (Private members are not documented.)

aDatamember

```
SomeType aDatamember;
```

aDatamember is a data member that holds some information about this sample class. This text explains how it functions, what it means, and how you use it.

See also: Related *data members*, **member functions, classes, global functions**, and so on.

anotherData member

```
ushort anotherDatamember;
```

protected

anotherDatamember has similar information to that for *aDatamember*.

Member functions

This section lists all member functions which are either newly defined for this class or which override inherited member functions. For virtual member functions, if you *always* need to override the member function, we've indicated that explicitly. Otherwise, you can override it as appropriate.

constructor

```
ClassName(SomeType aParameter);
```

The **ClassName** constructor creates an object of class **ClassName**, setting the *aDatamember* data member to *aParameter*.

zilch

```
virtual void zilch();
```

zilch causes the sample class to perform some action.

See also: **TSomethingElse::zilch**

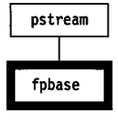
CharScanType — SYSTEM.H

The **CharScanType** structure holds the data that characterizes a keystroke event: the character code and the scan code.

```
struct CharScanType
{
   uchar charCode;
   uchar scanCode;
};
```

See also: **KeyDownEvent**, **TEvent**

fpbase — TOBJSTRM.H



fpbase provides the basic operations common to all object file stream I/O.

Member functions

constructor

```
fpbase();
fpbase( const char *name, int omode, int prot = filebuf::openprot );
fpbase( int f);
fpbase( int f, char *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 via 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( const char *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.

See also: Chapter 8, "Streamable objects"

rdbuf

```
filebuf * rdbuf();
```

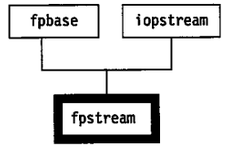
Returns a pointer to the current file buffer.

setbuf

```
void setbuf( char *buf, int len);
```

Allocates a buffer of size *len*.

fpstream TOBJSTRM.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. It is analogous to **fstream**, defined in fstream.h for the Borland C++ stream library.

Member functions

constructor

```
fpstream();
fpstream( const char name*, int mode, int prot = filebuf::openprot );
fpstream( int f );
fpstream( int f, char *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( const char *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 attatched to this stream.

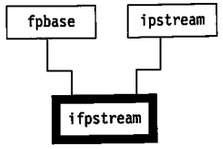
See also: Chapter 8, "Streamable objects"

rdbuf

```
filebuf * rdbuf();
```

Returns the data member *bp*.

ifpstream TOBJSTRM.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( const char *name, int mode = ios::in, int prot =
filebuf::openprot);
ifpstream( int f);
ifpstream( int f, char *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( const char *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.

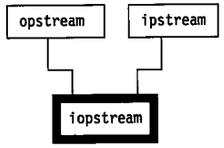
See also: Chapter 8, "Streamable objects"

rdbuf

```
filebuf * rdbuf();
```

Returns a pointer to the current file buffer.

iopstream TOBJSTRM.H



iopstream is a simple "mix" of its bases, **opstream** and **ipstream**. It provides the base class for simultaneous writing and reading streamable objects.

Member functions

constructor

```
iopstream( streambuf * buf);
```

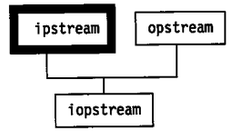
Creates a buffered **iopstream** with the given buffer and sets the *bp* data member to *buf*. The state is set to 0.

destructor

```
~iopstream();
```

Destroys the **iopstream** object.

ipstream TOBJSTRM.H



ipstream, a specialized input stream derivative of **pstream**, is the base class for reading (extracting) streamable objects. It is analogous to **istream**, defined in iostream.h for the Borland C++ stream library. **ipstream** is a friend class of **TPReadObjects**.

Member functions

constructor
```
ipstream( streambuf * );
```
Creates a buffered **ipstream** with the given buffer and sets the *bp* data member to *buf*. The state is set to 0.

destructor
```
virtual ~ipstream();
```
Destroys the **ipstream** object.

find **protected**
```
const void * find( P_id_type id );
```
Returns a pointer to the object corresponding to *id*.

readByte
```
uchar readByte();
```
Returns the character at the current stream position.

readBytes
```
void readBytes( void *data, size_t sz);
```
Reads *sz* bytes from current stream position, and writes them to *data*.

readData **protected**
```
void * readData( const TStreamableClass *c,
void *mem );
```
Invokes the appropriate **read** function to read from the stream to the object **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 **protected**
```
const TStreamableClass * readPrefix();
```
Returns the **TStreamableClass** object corresponding to the class **name** stored at the current position.

readString
```
char * readString();
char * readString( char *buf, unsigned maxLen);
```
Returns a string read from the current stream position.

readSuffix **protected**
```
void readSuffix();
```
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 **protected**
```
void registerObject( const void *adr );
```
Registers the class of the object **adr*.

seekg
```
ipstream& seekg( streampos pos);
ipstream& 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 ipstream& operator >> ( ipstream& ps, signed char& ch );
friend ipstream& operator >> ( ipstream& ps, unsigned char& ch );
friend ipstream& operator >> ( ipstream& ps, signed short& sh);
friend ipstream& operator >> ( ipstream& ps, unsigned short& sh);
friend ipstream& operator >> ( ipstream& ps, signed int& i);
friend ipstream& operator >> ( ipstream& ps, unsigned int& i);
friend ipstream& operator >> ( ipstream& ps, signed long& l);
friend ipstream& operator >> ( ipstream& ps, unsigned long& l);
friend ipstream& operator >> ( ipstream& ps, float& f);
friend ipstream& operator >> ( ipstream& ps, double& d);
friend ipstream& operator >> ( ipstream& ps, long double& d);
friend ipstream& operator >> ( ipstream& ps, TStreamable& t);
friend ipstream& operator >> ( ipstream& ps, void *& t);
```
Extracts (reads) from the **ipstream** *ps*, to the given argument. A reference to the stream is returned, allowing you to 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**.

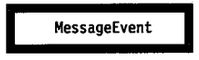
KeyDownEvent SYSTEM.H

The **KeyDownEvent** structure is a union of *keyCode* (a **ushort**) and *charScan* (of type **struct CharScanType**). These two members represent two ways of viewing the same data: either as a scan code or as an **unsigned**. Scan codes are what your program receives from the keyboard. **unsigned** is needed in a **switch** statement.

```
struct KeyDownEvent
{
   union
   {
      ushort keyCode;
      CharScanType charScan;
   };
};
```

See also: **TEvent**, **TEvent::getKeyEvent**

MessageEvent SYSTEM.H



The **MessageEvent** structure is defined as follows:

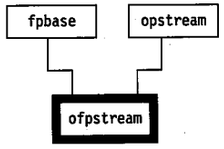
```
struct MessageEvent
{
   ushort command;
   union
   {
      void *infoPtr;
      long infoLong;
      ushort infoWord;
      short infoInt;
      uchar infoByte;
      char infoChar;
   };
};
```

A message event is a command, specified by *command*, together with one of several additional pieces of information, ranging from a single byte of data to a generic pointer. This arrangement allows for great flexibility

when Turbo Vision objects need to transmit and receive messages to and from other Turbo Vision objects.

See also: **TEvent**

ofpstream TOBJSTRM.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( const char *name, int mode = ios::out, int prot =
          filebuf::openprot);
ofpstream( int f );
ofpstream( int f, char *b, int len);
```

Creates a buffered **ofpstream** object. You can set the size and initial contents of the buffer using 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 with the file descriptor, *f*.

destructor

```
~ofpstream();
```

Destroys the **ofpstream** object.

open

```
void open( const char *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.

See also: Chapter 8, "Streamable objects" in this manual.

rdbuf

```
filebuf * rdbuf();
```

Returns the current file buffer.

Operators

For a look at the streamable classes and their hierarchies, see page 174 in Chapter 8 and page 218 in Chapter 12.

Streamable classes all declare four operators: two each of **operator >>** and **operator <<**. These operators are frequently unnecessary, but will ensure no ambiguities arise in cases of multiple inheritance.

Note that the two **operator >>** functions differ in that the first takes *a reference* to a **TClassName** object and the second takes *a pointer to a reference* 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 >>

```
ipstream& operator >> ( ipstream& is, TClassName& cl );
ipstream& operator >> ( ipstream& is, TClassName*& 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**

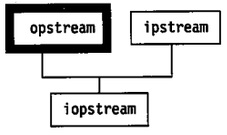
operator <<

```
opstream& operator << ( opstream& os, TClassName& cl );
opstream& operator << ( opstream& os, TClassName* 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**

opstream TOBJSTRM.H



opstream, a specialized derivative of **pstream**, is the base class for writing (inserting) streamable objects. **opstream** is a friend class of **TPWrittenObjects**.

Member functions

constructor

```
opstream( streambuf *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.

destructor

```
~opstream();
```

Destroys the **opstream** object.

find

```
P_id_type find( const void *adr );
```

protected

Returns the type ID for the object **adr*.

flush

```
ostream& flush();
```

Flushes the stream.

registerObject

```
void registerObject( const void *adr );
```

protected

Registers the class of the object **adr*.

seekp

```
opstream& seekp( streampos pos);
opstream& 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( const void *data, size_t sz );
```

Writes *sz* bytes from *data* buffer to the stream.

writeData

```
void writeData( TStreamable& );
```

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( const TStreamable& );` **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( const char *str );`

Writes *str* to the stream (together with a leading length byte).

writeSuffix `void writeSuffix( const TStreamable& );` **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

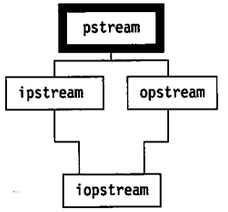
operator <<

```
friend opstream& operator << ( opstream& ps, signed char ch);
friend opstream& operator << ( opstream& ps, unsigned char ch);
friend opstream& operator << ( opstream& ps, signed short sh);
friend opstream& operator << ( opstream& ps, unsigned short sh);
friend opstream& operator << ( opstream& ps, signed int i);
friend opstream& operator << ( opstream& ps, unsigned int i);
friend opstream& operator << ( opstream& ps, signed long l);
friend opstream& operator << ( opstream& ps, unsigned long l);
friend opstream& operator << ( opstream& ps, float f);
friend opstream& operator << ( opstream& ps, double d);
friend opstream& operator << ( opstream& ps, long double d);
friend opstream& operator << ( opstream& ps, TStreamable& t);
friend opstream& operator << ( opstream& ps, TStreamable * 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 TOBJSTRM.H

Refer also to Chapter 8, "Streamable objects," for additional information on streams.



pstream is the base class for handling streamable objects. It is analogous to **ios**, defined in iostream.h for the Borland C++ stream library. Note that the following object streams work with the same **streambuf** classes as used with the standard Borland C++ **iostream** classes. This means that if you have developed an **iostream** variant that reads and writes to a modem, then you could also transmit objects via that modem.

Data members

bp `streambuf *bp;` **protected**

Pointer to the stream buffer.

state `int state;` **protected**

The format state flags, as enumerated in **ios**. Use **rdstate** to access the current state.

See also: **ios**, **pstream::rdstate**

types `static TStreamableTypes *types;` **protected**

Pointer to the **TStreamableTypes** data base of all registered types in this application.

See also: **TStreamableTypes**, **pstream::initTypes**

Member functions

constructor
```
pstream( streambuf *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. The second form allocates a default buffer.

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, const TStreamable& );
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( streambuf *sbp );
```
protected

Initializes the stream: sets *state* to 0 and *bp* to *sbp*.

initTypes
```
static void initTypes();
```
Creates the associated **TStreamableTypes** object, **types*. Called by the **TStreamableClass** constructor.

See also: **TStreamableTypes**, **TStreamableClass**

operator void *0
```
operator void *() 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
```
streambuf * 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.

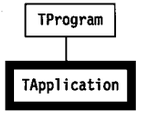
setstate
```
void setstate( int b );
```
protected

Updates the *state* data member with `state |= (b&0xFF)`.

Friends

The class **TStreamableTypes** is a friend of **pstream**.

TApplication APP.H



TApplication is a simple "wrapper" around **TProgram**, and only differs from **TProgram** in its constructors and destructors. Turbo Vision's subsystems (the memory, video, event, system error, and history list managers) are all static objects, so they are constructed before entry into **main**, and are all destroyed on exit from **main**.

Normally, you will want to derive your own applications from **TApplication**. Should you require a different sequence of subsystem initialization and shut down, however, you can derive your application

from **TProgram**, and manually initialize and shut down the Turbo Vision subsystems along with your own.

Member functions

constructor `TApplication();` **protected**

The implementation of **TApplication::TApplication** is as follows:

```
TApplication::TApplication() :
    TProgInit( &TApplication::initStatusLine,
               &TApplication::initMenuBar,
               &TApplication::initDeskTop)
{
   initHistory();
}
```

This creates a default **TApplication** object by passing the three *init* function pointers to the **TProgInit** constructor. The net result is that **TApplication** objects get a full-screen view, **initScreen** is called to set up various screen-mode-dependent variables, and a screen buffer is allocated. **initDeskTop**, **initStatusLine**, and **initMenuBar** are then called to create the three basic Turbo Vision views for your application. The desk top, status line, and menu bar objects are inserted in the application group. The state data member is set to (*sfVisible* | *sfSelected* | *sfFocused* | *sfModal* | *sfExposed*). The options data member is set to zero. Finally, the *application* pointer is set (to this object) and **initHistory** is called to initialize an associated **THistory** object.

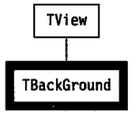
See also: **TProgInit::TProgInit, initScreen, initHistory**

destructor `virtual ~TApplication();` **protected**

Destroys the application object and, via the base destructors, destroys all its associated objects and frees all memory allocations. The implementation is as follows:

```
TApplication::~TApplication
{
   doneHistory();
}
```


TBackground APP.H



TBackground is a simple view consisting of a uniformly patterned rectangle. It is usually owned by a **TDeskTop** object.

Data member

pattern `char pattern;` **protected**

The character giving the view's background.

Member functions

constructor `TBackground(const TRect& bounds, char aPattern);`

Creates a **TBackground** class with the given *bounds* by calling the **TView** constructor. *growMode* is set to *gfGrowHiX* | *gfGrowHiY*, and the *pattern* data member is set to *aPattern*.

constructor `TBackground( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**, *TBackground::pattern*

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData, TStreamable**

draw `virtual void draw();`

Fills the background view rectangle with the current *pattern* in the default color.

getPalette `virtual TPalette& getPalette() const;`

Returns the default background palette string, *cpBackground*, "\x01".

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamable, ipstream**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

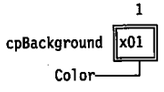
See also: **TStreamableClass, TStreamable, opstream**

Related functions

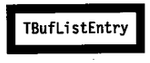
Certain operator functions are related to **TBackground** but are not member functions; see page 232 for more information.

TBackgound palette

Background objects use the default palette *cpBackground* to map onto the first entry in the application palette.



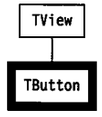
TBufListEntry — BUFFERS.H



TBufListEntry, in conjunction with **TVMemMgr**, is used internally by Turbo Vision to create and manage the video cache buffers for group drawing operations. All its members are private and will seldom if ever be referenced explicitly in normal applications. **TVMemMgr** is a friend class and the global operator **new** is a friend function.

See also: **TGroup::draw, TGroup::buffer**, operator **new**

TButton — DIALOGS.H



A **TButton** object is a box with a title and a shadow that generates a command when pressed. A button can be selected by typing the highlighted letter, by tabbing to the button and pressing *Spacebar*, by pressing *Enter* when the button is the default (indicated by highlighting), or by clicking on the button with a mouse.

With color and black-and-white palettes, a button has a three-dimensional look that moves when selected. On monochrome systems, a button is bordered by brackets, and other ASCII characters are used to indicate whether the button is default, selected, and so on.

Like the other controls defined in **TDialogs**, **TButton** is a "terminal" class. It can be inserted into any group and is intended for use without having to override any of its member functions.

You initialize a button by passing it a **TRect**, a title string, the command to generate when the button is pressed, and *aflags*, an unsigned short integer. To define a hot key for the button, the title string may contain tildes (~) around one of its characters, which then becomes the hot key. The hot-key *aFlags* parameter indicates whether the title should be centered or left justified, and whether the button should be the default (and therefore selectable by *Enter*).

There can only be one default button in a window or dialog at any given time. Buttons that are peers in a group grab and release the default state via *evBroadcast* messages. Buttons can be enabled or disabled using **setState** and the **enableCommand** and **disableCommand** member functions.

Data members

amDefault `Boolean amDefault;`

If *True*, the button is the default (and therefore selected when *Enter* is pressed). Otherwise, the button is "normal."

See also: *bfXXXX* button flag constants

command `ushort command;`

The command word of the event generated when this button is pressed.

See also: **TButton**'s constructor

flags `uchar flags;`

flags is a bitmapped data member used to indicate whether button text is left-justified or centered. The individual flags are described in Chapter 16 under "bf*XXXX* button flag constants."

See also: **TButton::draw**, *bfXXXX* button flag constants

title `const char *title;`

A string giving the label text for the button.

Member functions

constructor
```
TButton(const TRect& bounds, const char *aTitle, ushort aCommand,
        ushort aFlags);
```

Creates a **TButton** class with the given size by calling the constructor **TView**(*bounds*). The *aTitle* string is assigned to *title*. If the *bfDefault* bit is set in *aFlags*, this button will be highlighted as the default button. The button title will be centered if the *bfLeftJust* bit is clear, or left-justified if *bfLeftJust* is set.

The *options* data member is set to (*ofSelectable* | *ofFirstClick* | *ofPreProcess* | *ofPostProcess*) so that by default **TButton** responds to these events. *eventMask* is set to *evBroadcast*. The value of *aCommand* sets the *state* data member: If the given *aCommand* is not enabled, *sfDisabled* is set in the *state* data member.

constructor `TButton( StreamableInit streamableInit );` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView** constructor, *bfXXXX* button flag constants

destructor `~TButton();`

Frees the memory assigned to the button's *title*, then destroys the view with **~TView**.

See also: **~TView**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

draw `virtual void draw();`

Draws the button by calling **TButton::drawState**(*False*).

See also: **TButton::drawState**

drawState `void drawState(Boolean down);`

Draws the button in the "down" state (no shadow) if *down* is *True*; otherwise, it draws the button in the "up" state if *down* is *False*. The appropriate palettes are used to reflect the current state (normal, default, disabled). The button label is positioned according to the *bfLeftJust* bit in the *flags* data member.

See also: **TButton::draw**, **TButton::drawTitle**, *bfXXXX* flags

getPalette `virtual TPalette& getPalette() const;`

Returns the default button palette string, *cpButton*, "\x0A\x0B\x0C\x0D\x0E\x0E\x0E\x0F".

handleEvent `virtual void handleEvent(TEvent& event);`

Responds to being pressed in any of three ways: mouse clicks on the button, its hot key being pressed, or being the default button when a *cmDefault* broadcast arrives. When the button is pressed, a command event is generated with **TView::putEvent**, with the *TButton::command* data member assigned to *event::command* and *event::infoPtr* set to **this**.

Buttons also recognize the broadcast commands *cmGrabDefault* and *cmReleaseDefault*, to become or "unbecome" the default button, as appropriate, and *cmCommandSetChanged*, which causes them to check whether their commands have been enabled or disabled.

See also: **TView::handleEvent**

makeDefault `void makeDefault(Boolean enable);`

Used to make this button the default with *enable* set *True*, or to release the default with *enable* set *False*. These changes are usually the result of tabbing within a dialog box. The status is changed without actually

operating the button. The default button can be subsequently "pressed" by using the *Enter* key. **makeDefault** does nothing if the button is already the default button. Otherwise, the button's owner is told of the change in the button's default status. If *enable* is *True* the *cmGrabDefault* command is broadcast; otherwise, the *cmReleaseDefault* is broadcast. The button is redrawn if necessary to show the new status.

See also: **TButton::amDefault, TButton::press,** *bfDefault*

press `void press();`

press broadcasts a message that a button press event has occurred. Used internally by **handleEvent** when a mouse click "press" is detected when the default button is "pressed" with the *Enter* key.

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

setState `void setState(ushort aState, Boolean enable);`

Calls **TView::setState**(*aState, enable*), then **drawView**s the button if the button has been made *sfSelected* or *sfActive*. If focus is received (that is, if *aState* is *sfFocused*), the button grabs or releases default from the default button by calling **makeDefault**(*enable*).

See also: **TView::setState, TButton::makeDefault**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

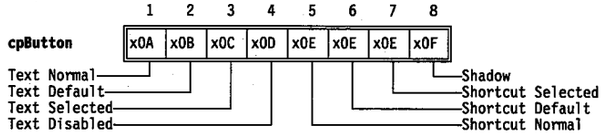
See also: **TStreamableClass, TStreamable, opstream**

Related functions

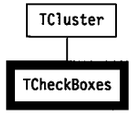
Certain operator functions are related to **TButton** but are not member functions; see page 232 for more information.

TButton palette

Button objects use the default palette *cpButton* to map onto *cpDialog* palette entries 10 through 15.



TCheckBoxes DIALOGS.H



TCheckBoxes is a specialized cluster of one to sixteen controls. Unlike radio buttons, any number of check boxes can be marked independently, so there is no default check box in the group. Marking can be done with mouse clicks, cursor movements, and *Alt-letter* shortcuts. Each check box can be highlighted and toggled on/off (with the *Spacebar*). An *X* appears in the box when it is selected. Other parts of your application typically examine the state of the check boxes to determine which options have been chosen by the user. Check box clusters are often associated with **TLabel** objects.

Data members

Apart from *value*, *sel*, and certain strings, which are all inherited from **TCluster**, there are no other public data members. The **ushort** *value* is interpreted as a set of 16 bits (0 through 15), with a 1 in the *item*'th bit position, meaning that the *item*'th check box is marked.

Member functions

constructor `TCheckBoxes(const TRect& bounds, TSItem *aStrings);`

Invokes the constructors **TCluster**(*bounds, aStrings*) and **TView**(*bounds*) to create a **TCheckBox** object with the given *bounds*. The **TStringCollection** **strings* data member is set from the *aStrings* argument, a linked list of **TSItem** objects. The *sel* and *value* data members are set to zero. *options* is set to (*ofSelectable* | *ofFirstClick* | *ofPreProcess* | *ofPostProcess*).

constructor `TCheckBoxes(StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TCluster::TCluster**, **TSItem**, *ofXXXX* flags

destructor **TCheckBoxes** uses ~**TCluster**, the base class destructor, to delete strings. The **TView** destructor then disposes of the view.

See also: **TCluster::~TCluster**

build
```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

draw
```
virtual void draw();
```

Draws the **TCheckBoxes** object by calling the inherited **TCluster::drawBox** member function. The default check box is `" [ ] "` when unselected and `" [X] "` when selected.

Note that if the boundaries of the view are sufficiently wide, check boxes can be displayed in multiple columns.

See also: **TCluster::drawBox**, **TCluster::column**

mark
```
virtual Boolean mark(int item);
```

Returns *True* if the *item*'th bit of *value* is set; that is, if the *item*'th check box is marked. These bits have no instrinsic meaning. You are free to override **mark**, **press**, and other check box member functions to give *value* your own interpretation. By default, the items are numbered 0 through 15 and each bit of *value* represents the state (on or off) of a check box.

See also: **TCheckBoxes::press**, **TCluster::value**

press
```
virtual void press(int item);
```

Toggles the *item*'th bit of *value*. These bits have no instrinsic meaning. You are free to override **mark**, **press**, and other check box member functions to give *value* your own interpretation. By default, the items are numbered 0 through 15.

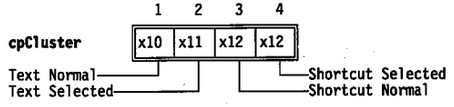
See also: **TCheckBoxes::mark**, **TCluster::value**

Related functions

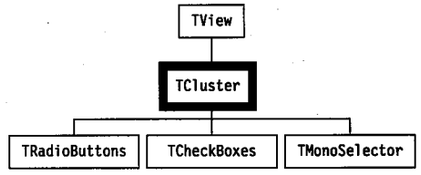
Certain operator functions are related to **TCheckBoxes** but are not member functions; see page 232 for more information.

Palette

By default, check box objects use *cpCluster*, the default palette for all cluster objects.



TCluster DIALOGS.H



A cluster is a group of controls that all respond in the same way. **TCluster** is an abstract class from which the useful group controls such as **TRadioButtons**, **TCheckBoxes**, and **TMonoSelector** are derived. Cluster controls are often associated with **TLabel** objects, letting you select the control by selecting on the adjacent explanatory label.

While buttons are used to generate commands and input lines are used to edit strings, clusters are used to toggle bit values in the *value* data member, which is of type **ushort**. The two standard descendants of **TCluster** use different algorithms when changing *value*: **TCheckBoxes** simply toggles a bit, while **TRadioButtons** toggles the enabled one and clears the previously selected bit. Both inherit most of their behavior from **TCluster**.

Data members

sel
```
int sel;
```

The currently selected item of the cluster.

strings `TStringCollection *strings;`

The list of items in the cluster.

value `ushort value;`

Current value of the control. The actual meaning of this data member is determined by the member functions developed in the classes derived from **TCluster**. For example, **TCheckBoxes** interprets each of the 16 bits of *value* as the state (on or off) of 16 distinct check boxes. In **TRadioButtons**, on the other hand, *value* can represent the state of a cluster of up to 65,536 buttons, since only one radio button can be "on" at any one time. (Note that **ushort** is currently typedefed as a 16-bit unsigned short giving a range of 0 to 65,535.)

Member functions

constructor `TCluster(const TRect& bounds, TSItem *aStrings);`

Calls **TView**(*bounds*) to create a **TCluster** object with the given *bounds*. Clears the *value* and *sel* data members. The **TStringCollection** **strings* data member is set from the *aStrings* argument, a linked list of **TSItem** objects.

constructor `TCluster( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TSItem**, **TView::TView**

destructor `~TCluster();`

Deletes the cluster's string collection, then destroys the view with **~TView**.

See also: **TView::~TView**

build `virtual Tstreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

dataSize `virtual ushort dataSize();`

Returns the size of *value*. Must be overridden in derived classes that change *value* or add other data data members, in order to work with **getData** and **setData**.

See also: **TCluster::getData**, **TCluster::setData**

drawBox `void drawBox(const char *icon, char marker);`

Called by the **draw** member functions of derived classes to draw the box in front of the string for each item in the cluster. *icon* is a five-character string (`" [ ] "` for check boxes, `" ( ) "` for radio buttons). *marker* is the character to use to indicate the box has been marked (`"X"` for check boxes, `"•"` for radio buttons).

See also: **TCheckBoxes::draw**, **TRadioButtons::draw**

getData `virtual void getData(void *rec);`

Writes the *value* data member to the given record and calls **drawView**. Must be overridden in derived classes that change the *value* data member in order to work with **dataSize** and **setData**.

See also: **TCluster::dataSize**, **TCluster::setData**, **TView::drawView**

getHelpCtx `ushort getHelpCtx();`

Returns (*sel* + *helpCtx*). This enables you to have separate help contexts for each item in the cluster. Use it to reserve a range of help contexts equal to *helpCtx* plus the number of cluster items minus one.

getPalette `virtual TPalette& getPalette() const;`

Returns a pointer to the default palette, *cpCluster*, "\x10\x11\x12\x12\".

handleEvent `virtual void handleEvent(TEvent& event) const;`

Calls **TView::handleEvent**, then handles all mouse and keyboard events appropriate to this cluster. Controls are selected by mouse click or cursor movement keys (including *Spacebar*). The cluster is redrawn to show the selected controls.

See also: **TView::handleEvent**

mark `virtual Boolean mark(int item);`

Called by **draw** to determine which items are marked. The default **TCluster::mark** returns *False*. **mark** should be overridden to return *True* if the *item*'th control in the cluster is marked; otherwise, it should return *False*.

See also: **TCheckBoxes::mark**, **TRadioButtons::mark**

movedTo `virtual void movedTo(int item);`

Called by **handleEvent** to move the selection bar to the *item*'th control of the cluster.

press `virtual void press(int item);`

Called by **handleEvent** when the *item*'th control in the cluster is pressed either by mouse click or keyboard event. This abstract member function must be overridden.

See also: **TCheckBoxes::press, TRadioButtons::press**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

setData `virtual void setData(void *rec);`

Reads the *value* data member from the given record and calls **drawView**. Must be overridden in derived cluster types that require other data members to work with **dataSize** and **getData**.

See also: **TCluster::dataSize, TCluster::getData, TView::drawView**

setState `virtual void setState(ushort aState, Boolean enable);`

Calls **TView::setState**, then calls **drawView** if *aState* is *sfSelected*.

See also: **TView::setState, TView::drawView**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

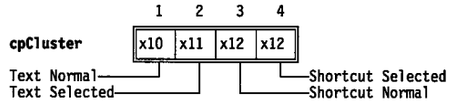
See also: **TStreamableClass, TStreamable, opstream**

Related functions

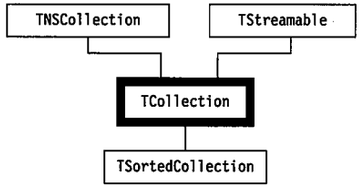
Certain operator functions are related to **TCluster** but are not member functions; see page 232 for more information.

Palette

TCluster objects use *cpCluster*, the default palette for all cluster objects, to map onto entries 16 through 18 of the standard dialog box palette.



TCollection OBJECTS.H



TCollection implements a streamable collection of arbitrary items, including other objects. Its main purpose is to provide a base class for more useful streamable collection classes. **TNSCollection** (the nonstreamable collection class) is a virtual base class for **TCollection**, providing the functions for adding, accessing, and removing items from a collection. **TStreamable** provides **TCollection** with the ability to create and name streams. Several friend functions and the overloaded operators, **>>** and **<<**, provide the ability to write and read collections to and from streams.

A collection is a more general concept than the traditional array, set, or list. **TCollection** objects size themselves dynamically at run time and offer a base for more specialized derived classes such as **TSortedCollection**, **TStringCollection**, and **TResourceCollection**. **TCollection** inherits from **TNSCollection** the member functions for adding and deleting items, as well as several *iterator* routines that call a function for each item in the collection.

Data members

`static const char *const name;`

The name of the collection class, "TCollection". Used internally by the stream manager.

See also: **TSortedCollection::TSortedCollectionName**

Member functions

constructor `TCollection( ccIndex aLimit, ccIndex aDelta );`

Creates a collection with *limit* set to *aLimit* and *delta* set to *adelta*. The initial number of items will be limited to *aLimit*, but the collection is allowed to grow in increments of *aDelta* until memory runs out or the number of items reaches *maxCollectionSize*.

constructor `TCollection( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TNSCollection::TNSCollection, TNSCollection::limit, TNSCollection::delta**

read `void read( ipstream& is );` **protected**

Reads a collection from the input stream *is* to the associated **TCollection** object.

See also: **ipstream**

readItem `virtual void *readItem( ipstream& is ) = 0;` **private**

You must define this pure virtual function in your derived class to read and return an item from the **ipstream** in a type-safe manner. This is usually done with a sequence of **>>** operations for each data member in your derived class.

write `void write( opstream& os );` **protected**

Writes the associated collection to the output stream *os*.

See also: **opstream**

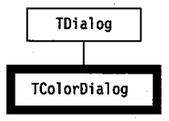
writeItem `virtual void writeItem( void *item, opstream& os ) = 0;` **private**

You must define this pure virtual function in your derived class to write an item to the **opstream**. This is usually done with a sequence of **<<** operations for each data member in your derived class.

Related functions

Certain operator functions are related to **TCollection** but are not member functions; see page 232 for more information.

TColorDialog COLORSEL.H



The interrelated classes **TColorItem, TColorGroup, TColorSelector, TMonoSelector, TColorDisplay, TColorGroupList, TColorItemList,** and **TColorDialog** provide viewers and dialog boxes from which the user can select and change the color assignments from available palettes with immediate effect on the screen.

TColorDialog is a specialized scrollable dialog box called "Colors" from which various palette selections can be examined before selection is made. **TColorDialog** uses many of the classes listed in the previous paragraph. We recommend that you read the descriptions for each of those classes.

Data members

bakLabel `TLabel *bakLabel;`

The background color label.

See also: **TLabel**

bakSel `TColorSelector *bakSel;`

The background color selector.

See also: **TColorSelector**

display `TColorDisplay *display;`

The color display object for this dialog box.

See also: **TColorDisplay**

forLabel `TLabel *forLabel;`

The foreground color label.

See also: **TLabel**

forSel `TColorSelector *forSel;`

The foreground color selector.

See also: **TColorSelector**

groups `TColorGroupList *groups;`

The color group for this dialog box.

See also: **TColorGroupList**

monoLabel `TLabel *monoLabel;`

The monochrome label.

See also: **TLabel**

monoSel `TMonoSelector *monoSel;`

The selector for monochrome attributes.

See also: **TMonoSelector**

pal `uchar *pal;`

The current palette selection.

See also: **TPalette**

Member functions

constructor

```
TColorDialog( uchar *aPalette, TColorGroup *aGroups );
TColorDialog( StreamableInit );
```

protected

The first format calls the **TDialog** and **TScrollBar** constructors to create a fixed, framed window titled "Colors" with two scroll bars. The *pal* data member is set to *aPalette*. The given *aGroups* argument creates and inserts a **TColorGroup** object with an associated label: "~G~roup". The items in *aGroups* initialize a **TColorItemsList** object, which is also inserted in the dialog, labeled as "~I~tem".

Foreground and background color selectors are created and inserted, labeled as "~F~oreground" and "~B~ackground". The string "Text " is displayed in the current text color. A **TMonoSelector** object is created, inserted, and labeled "~C~olor". All the items are displayed in their correct colors and attributes. Finally, "O~K~" and "Cancel" buttons are inserted.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

dataSize `virtual ushort dataSize();`

By default, **dataSize** returns the size of the current palette.

See also: **TPalette** class

getData `virtual void getData( void *rec );`

Copies **dataSize** bytes from *pal* to *rec*.

handleEvent `virtual void handleEvent( TEvent& event );`

Calls **TDialog::handleEvent** and redisplays if the broadcast event generated is *cmNewColorIndex*.

See also: **TDialog::handleEvent**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream**

setData `virtual void setData( void *rec);`

The reverse of **getData**: copies from *rec* to *pal*, restoring the saved color selections.

write `virtual void write( opstream& os);`

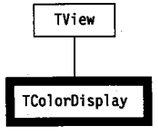
Writes to the output stream *os*.

See also: **TStreamableClass**, **TStreamable**, **opstream**

Related functions

Certain operator functions are related to **TColorDialog** but are not member functions; see page 232 for more information.

TColorDisplay COLORSEL.H



The interrelated classes **TColorItem, TColorGroup, TColorSelector, TMonoSelector, TColorDisplay, TColorGroupList, TColorItemList,** and **TColorDialog** provide viewers and dialog boxes from which the user can select and change the screen attributes and color assignments from available palettes with immediate effect on the screen.

TColorDisplay is a view for displaying text so that the user can select a suitable palette.

Data members

color `uchar *color;`

The current color for this display.

text `const char *text;`

The text string to be displayed.

Member functions

constructor `TColorDisplay( const TRect& bounds, const char *aText );`

Creates a view of the given size with **TView**(*bounds*), then sets *text* to the *aText* argument.

destructor `virtual ~TColorDisplay();`

Destroys both the view and the *text* string.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws the view with given text and current color.

handleEvent `virtual void handleEvent( TEvent& event );`

Calls **TView::handleEvent** and redraws the view as appropriate in response to the *cmColorBackgroundChanged* and *cmColorForegroundChanged* broadcast events.

See also: **TView::handleEvent**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

setColor `virtual void setColor( uchar *aColor );`

Sets *color* to *aColor*, broadcasts the change to the owning group, then calls **drawView**.

See also: **TView::drawView**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TColorDisplay** but are not member functions; see page 232 for more information.

TColorGroup COLORSEL.H

The interrelated classes **TColorItem, TColorGroup, TColorSelector, TMonoSelector, TColorDisplay, TColorGroupList, TColorItemList,** and **TColorDialog** provide viewers and dialog boxes from which the user can select and change the color assignments from available palettes with immediate effect on the screen.

The **TColorGroup** class defines a group of linked lists of **TColorItem** objects. Each member of a color group consists of a set of color names and their associated color codes.

Data members

items `TColorItem *items;`

Pointer to the associated list of color items associated with this color group.

name `const char *name;`

The name of the color group.

next `TColorGroup *next;`

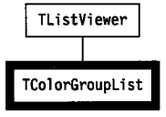
Pointer to next color group, or 0 if no next.

Member functions

constructor `TColorGroup( const char *nm, TColorItem *itm, TColorGroup *nxt = 0 );`

Creates a color group with the given argument values.

TColorGroupList COLORSEL.H



The interrelated classes **TColorItem**, **TColorGroup**, **TColorSelector**, **TMonoSelector**, **TColorDisplay**, **TColorGroupList**, **TColorItemList**, and **TColorDialog** provide viewers and dialog boxes from which the user can select and change the color assignments from available palettes with immediate effect on the screen.

TColorGroupList is a specialized derivative of **TListViewer** providing a scrollable list of named color groups. Groups can be selected in any of the usual ways (by mouse or keyboard). **TColorGroupList** uses the **TListViewer** event handler without modification.

Data members

groups `TColorGroup *groups;`

The color group for this list viewer.

Member functions

constructor `TColorGroupList( const TRect& bounds, TScrollBar *aScrollBar, TColorGroup *aGroups);`

Calls `TListViewer( bounds, 1, 0, aScrollBar)` to create a single-column list viewer a single vertical scroll bar. Then, sets *groups* to *aGroups*.

destructor `virtual ~TColorGroupList();`

Destroys the list viewer and all associated groups and their items.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

focusItem `virtual void focusItem( int item );`

Selects the given *item* by calling **TListViewer::focusItem**(*item*) and then broadcasts a *cmNewColorItem* event.

See also: **TListViewer::focusItem**

getText `virtual void getText( char *dest, int item, int maxLen );`

Copies the group name corresponding to *item* to the *dest* string.

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream**

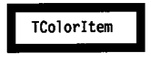
write `virtual void write( opstream& os);`

Writes to the output stream *os*.
See also: **TStreamableClass**, **TStreamable**, **opstream**

Related functions

Certain operator functions are related to **TColorGroupList** but are not member functions; see page 232 for more information.

TColorItem COLORSEL.H



The interrelated classes **TColorItem**, **TColorGroup**, **TColorSelector**, **TMonoSelector**, **TColorDisplay**, **TColorGroupList**, **TColorItemList**, and **TColorDialog** provide viewers and dialog boxes from which the user can select and change the color assignments from available palettes with immediate effect on the screen.

TColorItem defines a linked list of color names and indexes.

Data members

index `uchar index;`

The color index of the item.

name `const char *name;`

The name of the color item.

next `TColorItem *next;`

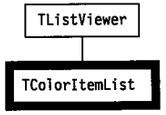
Link to the next color item, or 0 if there is no next item.

Member functions

constructor `TColorItem( const char *nm, uchar idx, TColorItem *nxt = 0 );`

Creates a color item object with *name* set to *nm*; *index* set to *idx*; and, by default, *next* set to 0.

TColorItemList COLORSEL.H



The interrelated classes **TColorItem**, **TColorGroup**, **TColorSelector**, **TMonoSelector**, **TColorDisplay**, **TColorGroupList**, **TColorItemList**, and **TColorDialog** provide viewers and dialog boxes from which the user can select and change the color assignments from available palettes with immediate effect on the screen.

TColorItemList is a simpler variant of **TColorGroupList** for viewing and selecting single color items rather than groups of colors. Like **TColorGroupList**, **TColorItemList** is specialized derivative of **TListViewer**. Color items can be selected in any of the usual ways (by mouse or keyboard). Unlike **TColorGroupList**, **TColorItemList** overrides the **TListViewer** event handler.

Data members

items `TColorItem *items;`

The color item list for this viewer.

Member functions

constructor `TColorItemList( const TRect& bounds, TScrollBar *aScrollBar, TColorItem *aItems );`

Calls **TListViewer**(*bounds*, 1, 0, *aScrollBar*) to create a single-column list viewer with a single vertical scroll bar. Then, the constructor sets *items* to *aItems* and *range* to the number of items.

constructor `TColorItemList( StreamableInit );` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by

calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData, TStreamable**

focusItem `virtual void focusItem( int item );`

Selects the given *item* by calling **TListViewer::focusItem**(*item*), then broadcasts a *cmNewColorIndex* event.

See also: **TListViewer::focusItem**

getText `virtual void getText( char *dest, int item, int maxLen );`

Copies the item *name* corresponding to *item* to the *dest* string.

handleEvent `virtual void handleEvent( TEvent& event );`

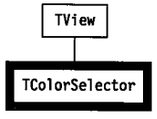
Calls **TListViewer::handleEvent**. If the event is *cmNewColorItem*, the appropriate item is focused and the viewer is redrawn.

See also: **TListViewer::handleEvent**

Related functions

Certain operator functions are related to **TColorItemList** but are not member functions; see page 232 for more information.

TColorSelector COLORSEL.H



The interrelated classes **TColorItem, TColorGroup, TColorSelector, TMonoSelector, TColorDisplay, TColorGroupList, TColorItemList,** and **TColorDialog** provide viewers and dialog boxes from which the user can select and change the color assignments from available palettes with immediate effect on the screen.

TColorSelector is a view for displaying the color selections available.

Data members

color `uchar color;`

Holds the currently selected color.

selType `ColorSel selType;`

Gives attribute (foreground or background) of the currently selected color. *ColorSel* is an **enum** defined as follows:

```
enum ColorSel { csBackground, csForeground };
```

Member functions

constructor `TColorSelector( const TRect& bounds, ColorSel sSelType );`

Calls **TView**(*bounds*) to create a view with the given *bounds*. Sets options *ofSelectable, ofFirstClick,* and *ofFramed*. Sets *eventMask* to *evBroadcast*, *selType* to *aSelType*, and *color* to 0.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws the color selector.

handleEvent `virtual void handleEvent( TEvent& event );`

Handles mouse and key events: You can click on a given color indicator to select that color, or you select colors by positioning the cursor with the arrow keys. Changes invoke **drawView** when appropriate.

read `virtual void *read(ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

write `virtual void write(opstream& os);`

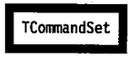
Writes to the output stream *os*.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TColorSelector** but are not member functions; see page 232 for more information.

TCommandSet VIEWS.H



TCommandSet is a non-view class for handling command sets. Member functions are provided for enabling and disabling commands and for testing for the presence of a given command. Several operators are overloaded to allow natural testing for equality and so on. Commands can be considered as integers in the range of 0 to 255.

Member functions

constructor

```
TCommandSet();
TCommandSet(const TCommandSet& commands);
```

The first constructor form creates and clears a command set. The second form creates a command set and initializes it from the *commands* argument.

disableCmd

```
void disableCmd(int cmd);
void disableCmd(const TCommandSet& tc);
```

Removes *cmd* (or the commands in *tc*) from the calling command set.

enableCmd

```
void enableCmd(int cmd);
void enableCmd(const TCommandSet& tc);
```

Adds *cmd* (or the commands in *tc*) to the calling command set.

has

```
Boolean has(int cmd);
```

Returns *True* if *cmd* is in the calling command set.

isEmpty

```
Boolean isEmpty();
```

Returns *True* if the calling command set is empty.

operator +=

```
void operator += (int cmd);
void operator += (const TCommandSet& tc);
```


A synonym for *enableCmd*: adds *cmd* (or the commands in *tc*) to the calling command set.

operator -=

```
void operator -= (int cmd);
void operator -= (const TCommandSet& tc);
```

A synonym for *disableCmd*: removes *cmd* (or the commands in *tc*) from the calling command set.

operator &=

```
TCommandSet& operator &= (const TCommandSet& tc);
```

Returns the intersection of *tc* and sets the calling command set to its intersection with *tc*, then returns the result.

operator |=

```
TCommandSet& operator |= (const TCommandSet& tc);
```

Returns the union of *tc* and sets the calling command set.

Friends

operator &

```
friend TCommandSet& operator & (const TCommandSet& tc1, const
                                TCommandSet& tc1);
```

Returns the intersection of *tc1* and *tc2* (that is, those commands common to both sets).

operator ==

```
friend int operator == (const TCommandSet& tc1, const TCommandSet& tc2);
```

Returns *True* if the sets *tc1* and *tc2* are not equal.

operator !=

```
friend int operator != (const TCommandSet& tc1, const TCommandSet& tc2);
```

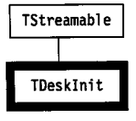
Returns *True* if the sets *tc1* and *tc2* are unequal.

operator |

```
friend TCommandSet& operator | (const TCommandSet& tc1,
                                const TCommandSet& tc2);
```

Returns the union of *tc1* and *tc2* (that is, those commands belonging to either or both sets).

TDeskInit APP.H



TDeskInit is used as a virtual base class for a number of classes, providing a constructor and **createBackground** member function used in creating

and inserting a background object. **TDeskInit**'s base class is **TStreamable**; together with some friend functions and operators, this allows desk tops to be written to and read from streams.

Member functions

constructor `TDeskInit( TBackground *(*cBackground) ( TRect bounds ) );`

This constructor takes a function address argument, usually *&TDeskTop::initBackground*. The **TDeskTop** constructor invokes **TGroup**(*bounds*) and **TDeskInit**(*&TDesk::initBackground*) to create a desk top object of size *bounds* and associated background. The latter is inserted in the desk top group object.

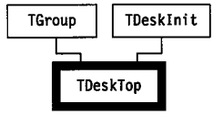
See also: **TDeskTop::TDeskTop**, **TDeskTop::initBackground**

createBackground `TBackground *(*createBackground) ( TRect bounds );` **protected**

Called by the **TDeskInit** constructor to create a background object of size *bounds* for the desk top.

See also: **TDeskTop::TDeskTop**, **TDeskTopInit::TDeskTopInit**

TDeskTop APP.H



TDeskTop inherits multiply from **TGroup** and the virtual base class **TDeskInit. TDeskInit** provides a constructor and **createBackground** member function used in creating and inserting a background object. **TDeskTop** is a simple group that owns the **TBackground** view upon which the application's windows and other views appear. **TDeskTop** represents the desk top area of the screen between the top menu bar and bottom status line (but only when the bar and line exist). **TDeskTop** objects can be written to and read from streams using the overloaded **>>** and **<<** operators.

Data members

background `TBackground *background;` **protected**

A pointer to the **TBackground** object associated with this desk top.

See also: **TDeskTop::initBackground**

Member functions

constructor `TDeskTop(const TRect& bounds);`

Creates a **TDeskTop** group with size *bounds* by calling its base constructors: **TGroup**(*bounds*) and **TDeskInit**(*&TDeskTop::initBackground*). The resulting **TBackgound** object is then inserted into the desk top. *growMode* is set to *gfGrowHiX* | *gfGrowHiY*.

constructor `TDeskTop( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TDeskTop::initBackground**, **TGroup::TGroup**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

cascade `void cascade(TRect& R);`

Redisplays all tileable windows owned by the desk top in cascaded format. The first tileable window in Z-order (the window "in back") is zoomed to fill the desk top, and each succeeding window fills a region beginning one line lower and one space further to the right than the one before. The active window appears "on top" as the smallest window.

See also: *ofTileable*, **TDeskTop::tile**

handleEvent `virtual void handleEvent(TEvent& event);`

Calls **TGroup::handleEvent** and takes care of the commands **cmNext** (usually the hot key *F6*) and **cmPrevious** by cycling through the windows (starting with the currently selected view) owned by the desk top.

See also: **TGroup::handleEvent**, *cmXXXX* command constants

initBackground

```
static TBackGround *initBackground( TRect );
```

The address of this member function is passed as an argument to the **TDeskInit** constructor. The latter invokes **initBackground** to create a new **TBackground** object with the same *bounds* as the calling **TDeskTop** object. The **TDeskTop::background** data member is set to point at the new **TBackground** object.

See also: **TDeskTop::TDeskTop**

shutDown

```
virtual void shutDown();
```

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

tile

```
void tile(TRect& r);
```

Redisplays all *ofTileable* views owned by the desk top in tiled format.

See also: **TDeskTop::cascade**, *ofTileable*

tileError

```
virtual void tileError();
```

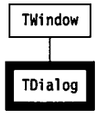
tileError is called if an error occurs during **TDeskTop::tile** or **TDeskTop::cascade**. By default, it does nothing. You may wish to override it to notify the user that the application is unable to rearrange the windows.

See also: **TDeskTop::tile**, **TDeskTop::cascade**

Related functions

Certain operator functions are related to **TDeskTop** but are not member functions; see page 232 for more information.

TDialog — DIALOGS.H



TDialog is a simple child of **TWindow** with the following properties:

- *growMode* is zero; that is, dialog boxes are not growable.
- The *flags* data member is set for *wfMove* and *wfClose*; that is, dialog boxes are moveable and closable (a close icon is provided).
- The **TDialog** event handler calls **TWindow::handleEvent** but additionally handles the special cases of *Esc* and *Enter* key responses. The *Esc* key generates a *cmCancel* command, while *Enter* generates the *cmDefault* command.
- The **TDialog::valid** member function returns *True* on *cmCancel*; otherwise, it calls its **TGroup::valid**.

Member functions

constructor

```
TDialog(const TRect& bounds, const char *aTitle);
```

Creates a dialog box with the given size and title by calling

```
TWindow::TWindow(bounds, aTitle, wnNoNumber);
TWindowInit( &TDialog::initFrame);
```

growMode is set to 0, and *flags* is set to *wfMove* | *wfClose*. This means that, by default, dialog boxes can move and close (via the close icon) but cannot grow (resize).

constructor **protected**

```
TDialog( StreamableInit streamableInit);
```

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

Note that **TDialog** does not define its own destructor, but uses **close** and the destructors inherited from **TWindow**, **TGroup**, and **TView**.

See also: **TWindow::TWindow**

build

```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

getPalette

```
virtual TPalette& getPalette() const;
```

Returns the default palette string, *cpDialog*:

"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2A\x2B\x2C\x2D\x2E\x2F
\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3A\x3B\x3C\x3D\x3E\x3F"

handleEvent

```
virtual void handleEvent(TEvent& event);
```

Calls **TWindow::handleEvent**(*event*), then handles *Enter* and *Esc* key events specially. In particular, *Esc* generates a *cmCancel* command, and *Enter* broadcasts a *cmDefault* command. This member function also handles *cmOk*, *cmCancel*, *cmYes*, and *cmNo* command events by ending the modal state of the dialog box. For each of the above events handled successfully, this member function calls **clearEvent**.

See also: **TWindow::handleEvent**

valid

```
virtual Boolean valid(ushort command);
```

Returns *True* if the command argument is *cmCancel*. This is the command generated by **handleEvent** when the *Esc* key is detected. If the command argument is not *cmCancel*, **valid** calls **TGroup::valid**(*command*) and returns the result of this call. **TGroup**'s **valid** calls the **valid** member functions of each of its subviews. The net result is that **valid** returns *True* only if the group controls all return *True*; otherwise, it returns *False*. A modal state cannot terminate until all subviews return *True* when polled with **valid**.

See also: **TGroup::valid**

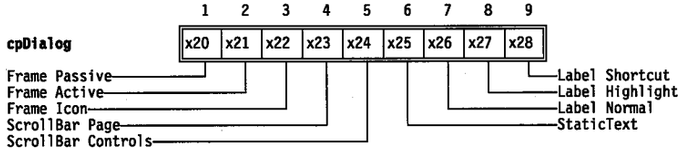
Related functions

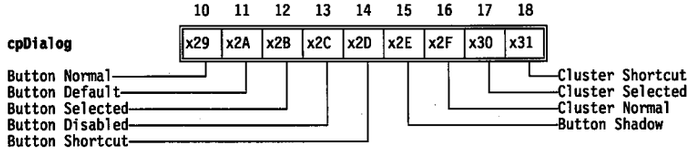
Certain operator functions are related to **TDialog** but are not member functions; see page 232 for more information.

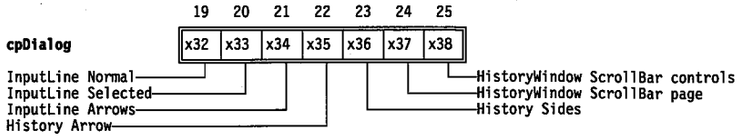
Palette

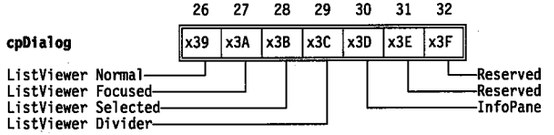
Dialog box objects use the default palette *cpDialog* to map onto the thirty-second through sixty-third entries in the application palette.

See also the ***getPalette*** *member function for each class.*

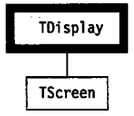








TDisplay SYSTEM.H



TDisplay provides low-level video functions for its derived class **TScreen**. These, and the other systems classes in SYSTEM.H, are listed briefly for guidance only: they are used internally by Turbo Vision and you would not need to use them explicitly for normal applications. **TView** is a friend class of **TDisplay**.

constructor

```
TDisplay();                                    protected
TDisplay( const TDisplay& );                   protected
```

Creates a **TDisplay** object. Called automatically via the **TApplication** constructor.

destructor

```
~TDisplay();                                   protected
```

Destroys the **TDisplay** object.

clearScreen

```
static void clearScreen( uchar w, uchar h );
```

Clears the screen of width *w* and height *h*.

getCols

```
static ushort getCols();
```

Returns the number of screen columns.

getCrtMode

```
static ushort getCrtMode();
```

Returns the current video mode.

See also: **TDisplay::setCrtMode, TDisplay::videoModes**

getCursorType

```
static ushort getCursorType();
```

Returns the cursor type.

See also: **TDisplay::setCursorType**

getRows

```
static ushort getRows();
```

Returns the number of screen rows.

setCrtMode

```
static void setCrtMode( ushort vmode );
```

Sets the video mode to *vmode* if possible. The available hardware is examined and the video mode is set to the "best possible" mode available.

See also: **TDisplay::getCrtMode, TDisplay::videoModes**

setCursorType

```
static void setCursorType( ushort ct );
```

Sets the cursor type to *ct*.

See also: **TDisplay::getCursorType**

videoModes

```
enum videoModes
{
   smBW80      = 0x0002,
   smCO80      = 0x0003,
   smMono      = 0x0007,
   smFont8x8   = 0x0100
};
```

Mnemonics for the video modes used by **TDisplay**.

See also: **TDisplay::setCrtMode, TDisplay::getCrtMode**

TDrawBuffer DRAWBUF.H

TDrawBuffer implements a simple, non-view buffer class with member functions for moving characters, attributes, and strings to and from a draw buffer. The contents of a draw buffer are typically used with **TView::writeBuf** or **TView::writeLine** to display text.

Member functions

data **protected**

```
ushort data[maxViewWidth];
```

Defines the array for this draw buffer.

moveBuf

```
void moveBuf(ushort indent, const void far *source, ushort attr,
             ushort count);
```

Moves text to the calling draw buffer. *count* bytes are copied from *source*. Copying starts at the offset given by *indent*. The high bytes of the words copied are set to *attr*, or remain unchanged if *attr* is zero.

moveChar

```
void moveChar(ushort indent, char c, ushort attr, ushort count);
```

Moves *count* copies of the character *c* and attribute *attr* into the calling draw buffer. Copying starts at the offset given by *indent*. The low byte of each affected word in the buffer receives *c*, while the high (attribute) bytes are set to *attr* provided *attr* is nonzero. If *attr* is zero, the high bytes are unchanged.

See also: **TDrawBuffer::putChar**

moveCStr

```
void moveCStr(ushort indent, const char far *str, ushort attrs);
```

Moves the two-color string *str* to the calling draw buffer. Copying starts at the offset given by *indent*. The characters in *str* are set in the low bytes of each buffer word. The high bytes of the buffer words are set to `lo(attrs)` or `hi(attrs)`. Tilde characters (~) in the string are used to toggle between the two attribute bytes passed via *attrs*.

See also: **TDrawBuffer::moveStr**

moveStr

```
void moveStr(ushort indent, const char far *str, ushort attrs);
```

Moves the string *str* to the calling draw buffer. Copying starts at the offset given by *indent*. The characters in *str* are set in the low bytes of each buffer word. The high bytes of the buffer words are set to *attrs*, or remain unchanged if *attrs* is zero.

See also: **TDrawBuffer::moveCStr**

putAttribute

```
void putAttribute(ushort indent, ushort attr);
```

Inserts *attr* into the upper byte of the calling buffer. The insertion position is offset by the value of *indent*.

See also: **TDrawBuffer::putChar**

putChar

```
void putChar(ushort indent, ushort c);
```

Inserts *c* into the lower byte of the calling buffer. The insertion position is offset by the value of *indent*.

See also: **TDrawBuffer::moveChar**

Friends

The classes **TSystemError** and **TView** and the function **genRefs** are friends of **TDrawBuf**.

TEvent SYSTEM.H

TEvent

The **TEvent** structure is defined as follows:

```
struct TEvent
{
   ushort what;
   union
   {
      MouseEventType mouse;
      KeyDownEvent keyDown;
      MessageEvent message;
   };
   void getMouseEvent();
   void getKeyEvent();
};
```

TEvent holds a union of objects of type **MouseEventType**, **KeyDownEvent**, and **MessageEvent** types, keyed by the **what** field. The **handleEvent** member functions for **TView** and its derived classes take a **TEvent** object as argument and respond with the appropriate action. The **getMouseEvent** member function of **TEvent** extracts the appropriate fields from the event queue by calling **TEventQueue::getMouseEvent**. **TEvent::getKeyEvent** grabs key events directly (using int 16h calls).

See also: **MouseEventType**, **KeyDownEvent**, **MessageEvent**, **TView::handleEvent**, **TEventQueue**

TEventQueue SYSTEM.H

TEventQueue

TEventQueue implements a FIFO queue of mouse events. The inner details will seldom be of interest in normal applications. It will usually be sufficient to know how the **TEvent** structure interacts with **TView::handleEvent** and its derivatives.

Member functions

constructor `TEventQueue();`

Creates a **TEventQueue** object and calls **resume**.

See also: **TEventQueue::resume**

destructor `~TEventQueue();`

Destroys the **TEventQueue** object and calls **suspend**.

See also: **TEventQueue::suspend**

getMouseEvent `static void getMouseEvent( TEvent& e );`

Extracts a mouse event (if one) from the FIFO queue and sets the **TEvent** data members accordingly.

See also: **TEvent::getMouseEvent**

resume `static void resume();`

Does nothing if a mouse is not currently present. Otherwise grabs the next mouse event (if any, by calling **TMouse::getEvent**), and (re-)registers the mouse handler.

See also: **TMouse::present**, **TMouse::registerHandler**

suspend `static void suspend();`

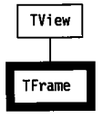
Calls **TMouse::suspend** for the current **TMouse** object.

See also: **TMouse::suspend**

Friends

The class **TView** and the function **genRefs** are friends of **TEventQueue**.

TFrame VIEWS.H



TFrame provides the distinctive frames around windows and dialog boxes. Users will probably never need to deal with frame objects directly, as they are added to window objects by default.

Member functions

constructor `TFrame(const TRect& bounds);`

Calls **TView**(*bounds*), then sets *growMode* to *gfGrowHiX* | *gfGrowHiY* and sets *eventMask* to *eventMask* | *evBroadcast*, so **TFrame** objects default to handling broadcast events.

constructor `TFrame( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**, **TStreamable**

draw `virtual void draw();`

Draws the frame with color attributes and icons appropriate to the current *state* flags: active, inactive, being dragged. Adds zoom, close and resize icons depending on the owner window's *flags*. Adds the title, if any, from the owning window's *title* data member. Active windows are drawn with

a double-lined frame and any icons; inactive windows are drawn with a single-lined frame and no icons.

See also: *sfXXXX* state flag constants, *wfXXXX* window flag constants

getPalette `virtual TPalette& getPalette() const;`

Returns the default frame palette string, *cpFrame*, "\x01\x01\x02\x02\x03".

handleEvent `virtual void handleEvent(TEvent& event);`

Calls **TView::handleEvent**, then handles mouse events. If the mouse is clicked on the close icon, **TFrame::handleEvent** generates a *cmClose* event. Clicking on the zoom icon or double-clicking on the top line of the frame generates a *cmZoom* event. Dragging the top line of the frame moves the window, and dragging the resize icon moves the lower right corner of the view and therefore changes its size.

See also: **TView::handleEvent**

setState `virtual void setState(ushort aState, Boolean enable);`

Calls **TView::setState**(*aState*, *enable*). If the new state is *sfActive* or *sfDragging*, calls **drawView** to redraw the view.

See also: **TView::setState**, **TView::drawView**

Friends

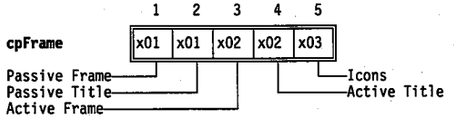
TDisplay is a friend of **TFrame**'s.

Related functions

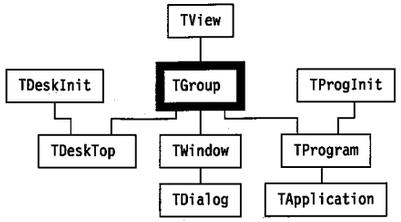
Certain operator functions are related to **TFrame** but are not member functions; see page 232 for more information.

Palette

Frame objects use the default palette, *cpFrame*, to map onto the first three entries in the standard window palette.



TGroup VIEWS.H



TGroup objects and their derivatives (called groups for short) provide the central driving power to Turbo Vision. A group is a special breed of view. In addition to all the members derived from **TView** and **TStreamable**, a group has additional members and many overrides that allow it to control a dynamically linked list of views (including other groups) as though they were a single object. We often talk about the subviews of a group even when these subviews are often groups in their own right.

Although a group has a rectangular boundary from its **TView** ancestry, a group is only visible through the displays of its subviews. A group conceptually draws itself via the **draw** member function of its subviews. A group owns its subviews, and together they must be capable of drawing (filling) the group's entire rectangular *bounds*. During the life of an application, subviews and subgroups are created, inserted into groups, and displayed as a result of user activity and events generated by the application itself. The subviews can just as easily be hidden, deleted from the group, or disposed of by user actions (such as closing a window or quitting a dialog box).

The three subclasses of **TGroup**, namely **TWindow**, **TDeskTop**, and **TApplication** (via **TProgram**), illustrate the group and subgroup concept. A **TApplication** will typically own a **TDeskTop** object, a **TStatusLine** object, and a **TMenuView** object. **TDeskTop** is a **TGroup** subclass, so it, in turn, can own **TWindow** objects, which in turn own **TFrame** objects, **TScrollBar** objects, and so on.

TGroup objects delegate both drawing and event handling to their subviews, as explained in Chapter 4, "Views" and Chapter 5, "Event-driven programming".

TGroup objects are not usually instantiated; rather you would instantiate one or more of **TGroup**'s subclasses: **TApplication**, **TDeskTop**, and **TWindow**.

All **TGroup** objects are streamable, inheriting from **TStreamable** by way of **TView**. This means that **TGroup** objects (including your entire application group) can be written to and read from streams in a type-safe manner using the familiar C++ **iostream** operators.

Data members

buffer `uchar far *buffer;`

Points to a buffer used to cache redraw operations, or is 0 if the group has no cache buffer. Cache buffers are created and destroyed automatically, unless the *ofBuffered* flag is cleared in the group's *options* member.

See also: **TGroup::draw**, **TGroup::lock**, **TGroup::unlock**

clip `TRect clip;`

Holds the clip extent of the group, as returned by **getExtent** or **getClipRect**. The clip extent is defined as the minimum area that needs redrawing when **draw** is called.

See also: **TView::getClipRect**, **TView::getExtent**

current `TView *current;`

Points to the subview that is currently selected, or is 0 if no subview is selected.

See also: *sfSelected*, **TView::select**

endState `ushort endState;`

Holds the state of the group after a call to **endModal**.

See also: **TGroup::endModal**

last `TView *last;`

Points to the last subview in the group (the one furthest from the top in Z-order).

lockFlag `uchar lockFlag;`

Acts as a semaphore to control buffered group draw operations. **lockFlag** keeps a count of the number of locks set during a set of nested **draw** calls. **lock** and **unlock** increment and decrement this value. When it reaches zero, the whole group will draw itself from its buffer. Intensive draw operations should be sandwiched between calls to **lock** and **unlock** to prevent excessive screen flicker.

See also: **TGroup::draw**, **TGroup::lock**, **TGroup::unlock**

phase `phaseType phase;`

The current phase of processing for a focused event. Subviews that have the *ofPreProcess* or *ofPostProcess* flags set can examine *owner->phase* to determine whether a call to their **handleEvent** is happening in the *phPreProcess*, *phFocused*, or *phPostProcess* phase.

phaseType is an enumeration defined as follows in **TView**:

```
enum phaseType {phFocussed, phPreProcess, phPostProcess};
```

See also: *ofPreProcess*, *ofPostProcess*, **TGroup::handleEvent**

Member functions

constructor `TGroup(TRect& bounds);`

Calls **TView::TView**(*bounds*), sets *ofSelectable* and *ofBuffered* in *options*, and sets *eventMask* to 0xFFFF. The members *last*, *current*, *buffer*, *lockFlag*, and *endState* are all set to zero.

constructor `TGroup( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**

destructor `~TGroup();`

Hides the group using **hide**, then disposes of each subview in the group using **delete** *p*. Finally, the buffer is freed (if one).

See also: **TView::hide**

at `TView *at(short index);`

Returns a pointer to the subview at index position in Z-order.

See also: **TGroup:indexOf**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

changeBounds `virtual void changeBounds (TRect& bounds);`

Overrides **TView::changeBounds**. Changes the group's bounds to *bounds* and then calls **calcBounds** followed by **changeBounds** for each subview in the group.

See also: **TView::calcBounds**, **TView::changeBounds**

dataSize `virtual ushort dataSize();`

Overrides **TView::dataSize**. Returns total size of group by calling and accumulating **dataSize** for each subview.

See also: **TView::dataSize**

draw `virtual void draw();`

Overrides **TView::draw**. If a cache buffer exists (see **TGroup::buffer** data member), then the buffer is written to the screen using **TView::writeBuf**. Otherwise, each subview is told to draw itself using a call to **TGroup::redraw**.

See also: **TGroup::buffer**, **TGroup::redraw**, **TView::draw**

drawSubViews `void drawSubViews(TView *p, TView *bottom);`

Calls **drawView** for each subview starting at **p*, until the subview **bottom* is reached.

endModal `virtual void endModal(ushort command);`

If this group is the current modal view, **endModal** terminates its modal state. *command* is passed to **execView** (which made this view modal in the first place), which returns *command* as its result. If this group is *not* the current modal view, it calls **TView::endModal**.

See also: **TGroup::execView**, **TGroup::execute**, **TView::endModal**, *sfModal*

eventError `virtual void eventError(TEvent& event);`

eventError is called whenever the modal **TGroup::execute** event-handling loop encounters an event that cannot be handled. The default action is: If the group's *owner* is nonzero, **eventError** calls its owner's **eventError**. Normally this chains back to **TApplication**'s **eventError**. You can override **eventError** to trigger appropriate action.

See also: **TGroup::execute**, **TGroup::execView**, *sfModal*

execute `virtual ushort execute();`

Overrides **TView::execute**. **execute** is a group's main event loop: It repeatedly gets events using **getEvent** and handles them using **handleEvent**. The event loop is terminated by the group or some subview through a call to **endModal**. Before returning, however, **execute** calls **valid** to verify that the modal state can indeed be terminated.

See also: **TGroup::getEvent**, **TGroup::handleEvent**, **TGroup::endModal**, **TGroup::valid**, **TGroup::endState**

execView `ushort execView(TView *p);`

execView is the "modal" counterpart of the "modeless" **insert** and **remove** member functions. Unlike **insert**, after inserting a view into the group, **execView** waits for the view to execute, then removes the view, and finally returns the result of the execution. **execView** is used in a number of places throughout Turbo Vision, most notably to implement **TApplication::run** and to execute modal dialog boxes.

execView saves the current context (the selected view, the modal view, and the command set), makes *p* modal by calling *p->setState(sfModal, True)*, inserts *p* into the group (if it isn't already inserted), and calls *p->execute*. When *p->execute* returns, the group is restored to its previous state, and the result of *p->execute* is returned as the result of the **execView** call. If *p* is 0 upon a call to **execView**, a value of *cmCancel* is returned.

See also: **TGroup::insert**, **TGroup::execute**, *sfModal*.

first `TView *first();`

Returns a pointer to the first subview (the one closest to the top in Z-order), or 0 if the group has no subviews.

See also: **TGroup::last**

firstMatch `TView *firstMatch(ushort aState, ushort aOptions)`

Returns a pointer to the first subview that matches its *state* with *aState* and its *options* with *aOptions*.

firstThat `TView *firstThat(Boolean(*func)(TView*, void*), void *args);`

firstThat applies a user-supplied Boolean function ***func**, along with an argument list given by *args* (possibly empty), to each subview in the group (in Z-order) until ***func** returns *True*. The returned result is the subview pointer for which ***func** returns *True*, or 0 if ***func** returns *False* for all items.

The first pointer argument of ***func** scans the subview. The second argument of ***func** is set from the *args* pointer of **firstThat**, as shown in the following implementation:


```
TView *TGroup::firstThat( Boolean (*func)(TView *, void *), void *args )
{
    TView *temp = last;
    if( temp == 0 )
        return 0;

    do  {
        temp = temp->next;
        if( func( temp, args ) == True )
            return temp;
    } while( temp != last );
    return 0;
}
```

See also: **TGroup::forEach**

forEach `void forEach(void (*func)(TView *, void *), void *args);`

forEach applies an action, given by the function ***func**, to each subview in the group in Z-order. The *args* argument lets you pass arbitrary arguments to the action function:

```
void TGroup::forEach( void (*func)(TView*, void *), void *args )
{
   TView *term = last;
   TView *temp = last;
   if( temp == 0 )
      return;

   TView *next = temp->next;
   do  {
      temp = next;
      next = temp->next;
      func( temp, args );
   } while( temp != term );

}
```

See also: **TGroup::firstThat**

freeBuffer `void freeBuffer();`

Frees the group's draw buffer (if one exists) by calling **delete** *buffer* and setting buffer to 0.

See also: *TGroup::buffer*, **TGroup::getBuffer**, **TGroup::draw**

getBuffer `void getBuffer();`

If the group is *sfExposed* and *ofBuffered*, a draw buffer is created. The buffer size will be (*size.x* * *size.y*) and the *buffer* data member is set to point at the new buffer.

See also: *TGroup::buffer*, **TGroup::freeBuffer**, **TGroup::draw**

getData

```
virtual void getData(void *rec);
```

Overrides **TView::getData**. Calls **getData** for each subview in reverse Z-order, incrementing the location given by *rec* by the *dataSize* of each subview.

See also: **TView::getData**, **TGroup::setData**

getHelpCtx

```
virtual ushort getHelpCtx();
```

Returns the help context of the current focused view by calling the selected subviews' **getHelpCtx** member function. If no help context is specified by any subview, **getHelpCtx** returns the value of its own *HelpCtx* member.

handleEvent

```
virtual void handleEvent(TEvent& event);
```

Overrides **TView::handleEvent**. A group basically handles events by passing them to the **handleEvent** member functions of one or more of its subviews. The actual routing, however, depends on the event class.

For focused events (by default, *evKeyDown* and *evCommand*; see *focusedEvents* variable), event handling is done in three phases: First, the group's *phase* member is set to *phPreProcess* and the event is passed to the **handleEvent** of all subviews that have the *ofPreProcess* flag set. Next, *phase* is set to *phFocused* and the event is passed to the **handleEvent** of the currently selected view. Finally, *phase* is set to *phPostProcess* and the event is passed to the **handleEvent** of all subviews that have the *ofPostProcess* flag set.

For positional events (by default, *evMouse*; see *PositionalEvents* variable), the event is passed to the **handleEvent** of the first subview whose bounding rectangle contains the point given by *event.where*.

For broadcast events (events that aren't focused or positional), the event is passed to the **handleEvent** of each subview in the group in Z-order.

If a subview's *eventMask* member masks out an event class, **TGroup::handleEvent** will **never** send events of that class to the subview. For example, the default *eventMask* of **TView** disables *evMouseUp*, *evMouseMove*, and *evMouseAuto*, so **TGroup::handleEvent** will never send such events to a standard **TView**.

See also: **focusedEvents**, **positionalEvents**, *evXXXX* event constants, **TView::eventMask**, **handleEvent** member functions

indexOf

```
short indexOf(TView *p);
```

Returns the Z-order position (index) of the subview **p*.

See also: **TGroup:at**

insert

```
void insert(TView *p);
```

Inserts the view given by *p* in the group's subview list. The new subview is placed on top of all other subviews. If the subview has the *ofCenterX* and/or *ofCenterY* flags set, it is centered accordingly in the group. If the view has the *sfVisible* flag set, it will be shown in the group—otherwise, it remains invisible until specifically shown. If the view has the *ofSelectable* flag set, it becomes the currently selected subview.

See also: **TGroup::remove**, **TGroup::execView**

insertBefore

```
void insertBefore(TView *p, TView *target);
```

Inserts the view given by *p* in front of the view given by *target*. If *target* is 0, the view is placed behind all other subviews in the group.

See also: **TGroup::insert**, **TGroup::remove**

lock

```
void lock();
```

Locks the group, delaying any screen writes by subviews until the group is unlocked. **lock** has no effect unless the group has a cache buffer (see *ofBuffered* and *TGroup::buffer*). **lock** works by incrementing the data member *lockFlag*. This semaphore is likewise decremented by **unlock**. When a call to **unlock** decrements the count to zero, the entire group is written to the screen using the image constructed in the cache buffer.

By "sandwiching" draw-intensive operations between calls to **lock** and **unlock**, unpleasant "screen flicker" can be reduced, if not eliminated. For example, the **TDeskTop::tile** and **TDeskTop::cascade** member functions use **lock** and **unlock** to reduce flicker.

lock and **unlock** calls *must* be balanced; otherwise, a group may end up in a permanently locked state, causing it to not redraw itself properly when so requested.

See also: **TGroup::unlock**, **TGroup::lockFlag**

matches

```
Boolean matches(TView * p);
```

Returns *True* if the *state* and *options* settings of the view **p* are identical to those of the calling view.

read

```
virtual void *read( ipstream& is);
```

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream**

redraw

```
void redraw();
```

Redraws the group's subviews in Z-order. **TGroup::redraw** differs from **TGroup::draw** in that **redraw** will never draw from the cache buffer.

See also: **TGroup::draw**

remove

```
void remove(TView *p);
```

Removes the subview *p* from the group and redraws the other subviews as required. *p*'s *owner* and *next* members are set to 0.

See also: **TGroup::insert, TGroup::removeView**

removeView

```
void removeView(TView *p);
```

Removes the subview *p* from this group. Used internally by **TGroup::remove**.

See also: **TGroup::remove**

resetCurrent

```
void resetCurrent();
```

Selects (makes current) the first subview in the chain that is *sfVisible* and *ofSelectable*. **resetCurrent** works by calling:

```
setCurrent(firstMatch(sfVisible, ofSelectable), normalSelect).
```

The following **enum** type is useful for select mode arguments:

```
enum selectMode { normalSelect, enterSelect, leaveSelect };
```

See also: **TGroup::setCurrent**

selectNext

```
void selectNext(Boolean forwards);
```

If *forwards* is *True*, **selectNext** selects (makes current) the next selectable subview (one with its *ofSelectable* bit set) in the group's Z-order. If *forwards* is *False*, the member function selects the previous selectable subview.

See also: *ofXXXX* option flag constants, **TGroup::selectView**

setCurrent

```
void setCurrent(TView *p, selectMode mode);
```

selectMode is an enumeration defined in **TGroup** as follows:

```
enum selectMode {normalSelect, enterSelect, leaveSelect};
```

If **p* is the current subview, **setCurrent** does nothing. Otherwise, **p* is made current (that is, selected) by a call to **setState**.

See also: **TGroup::resetCurrent**

setData

```
virtual void setData(void *rec);
```


Overrides **TView::setData**. Calls **setData** for each subview in reverse Z-order, incrementing the location given by *rec* by the **dataSize** of each subview.

See also: **TGroup::getData, TView::setData**

setState

```
virtual void setState(ushort aState, Boolean enable);
```

Overrides **TView::setState**. First calls the inherited **TView::setState**, then updates the subviews as follows:

- If *aState* is *sfActive* or *sfDragging*, then each subview's **setState** is called to update the subview correspondingly.
- If *aState* is *sfFocused*, then the currently selected subview is called to focus itself correspondingly. If *aState* is *sfExposed*, **doExpose** is called for each subview. Finally, if *enable* is *False*, **freeBuffer** is called.

See also: **TView::setState, TGroup::doExpose, TGroup::freeBuffer**

shutDown

```
virtual void shutDown();
```

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

unlock

```
void unlock();
```

Unlocks the group by decrementing *lockFlag*. If *lockFlag* becomes zero, then the entire group is written to the screen using the image constructed in the cache buffer.

See also: **TGroup::lock**

valid

```
virtual Boolean valid(ushort command);
```

Overrides **TView::valid**. Returns *True* if all the subview's **valid** calls return *True*. **TGroup::valid** is used at the end of the event-handling loop in **TGroup::execute** to confirm that termination is allowed. A modal state cannot terminate until all **valid** calls return *True*. A subview can return *False* if it wants to retain control.

See also: **TView::valid, TGroup::execute**

write

```
virtual void write( opstream& os);
```

Writes to the output stream *os*.

See also: **TStreamableClass, TStreamable, opstream**

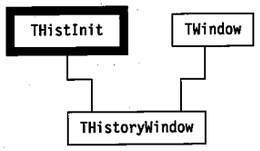
Friends

The function **genRefs** is a friend of **TGroup**.

Related functions

Certain operator functions are related to **TGroup** but are not member functions; see page 232 for more information.

THistInit DIALOGS.H



THistInit provides a constructor and a **createListViewer** member function used in creating and inserting a list viewer into a history window.

Member functions

constructor

```
THistInit( TListViewer *(*cListViewer)( TRect r, TWindow *w,ushort
                        histID );
```

The **THistoryWindow** constructor calls this base constructor, **THistInit::THistInit**, passing the address of **THistory::initViewer**, which is of type **cListViewer**. This creates and inserts a list viewer into the given history window with the given size and history list.

See also: **THistoryWindow** constructor

createListViewer

```
TListViewer *(*createListViewer)(TRect r, TWindow *w, ushort
             histId);
```

protected

Called by the **THistInit** constructor to create a list viewer for the window *w* with size *r* and history list given by *histId*.

See also: **THistory**, **TListViewer**, **THistoryWindow**

THistory DIALOGS.H

A **THistory** object implements a pick list of previous entries, actions, or choices from which the user can select a "rerun". **THistory** objects are linked to a **TInputLine** object and to a history list. History list information is stored in a block of memory on the heap. When the block fills up, the oldest history items are deleted as new ones are added.

THistory itself shows up as an icon (↓) next to an input line. When the user clicks on the history icon, Turbo Vision opens up a history window (see **THistoryWindow**) with a history viewer (see **THistoryViewer**) containing a list of previous entries for that list.

Different input lines can share the same history list by using the same ID number.

Data members

historyID

```
ushort historyId;
```

Each history list has a unique ID number, assigned by the programmer. Different history objects in different windows may share a history list by using the same history ID.

link

```
TInputLine *link;
```

A pointer to the linked **TInputLine** object.

Member functions

constructor

```
THistory(const TRect& bounds, TInputLine *aLink, ushort aHistoryId);
```

Creates a **THistory** object of the given size by calling **TView**(*bounds*), then setting the *link* and *historyId* members with the given argument values. The *options* member is set to *ofPostProcess*. The *evBroadcast* bit is set in

eventMask in addition to the *evMouseDown*, *evKeyDown*, and *evCommand* bits set by **TView**(*bounds*).

constructor `THistory( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

draw `virtual void draw();`

Draws the **THistory** icon in the default palette.

getPalette `virtual TPalette& getPalette() const;`

Returns a pointer to the default palette, *cpHistory*, "\x16\x17".

handleEvent `virtual void handleEvent(TEvent& event);`

Calls **TView::handleEvent**(*event*), then handles relevant mouse and key events to select the linked input line and create a history window.

See also: **THistory::initHistoryWindow**

initHistoryWindow `virtual THistoryWindow *initHistoryWindow(const TRect& bounds)`

Creates a **THistoryWindow** object and returns a pointer to it. The new object has the given *bounds* and the same *historyId* as the calling **THistory** object. The new object gets its *helpCtx* from the calling object's linked **TInputLine**.

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream**

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

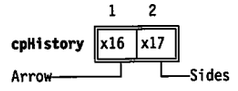
See also: **TStreamableClass**, **TStreamable**, **opstream**

Related functions

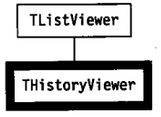
Certain operator functions are related to **THistory** but are not member functions; see page 232 for more information.

Palette

History icons use the default palette, *cpHistory*, to map onto the twenty-second and twenty-third entries in the standard dialog box palette.



THistoryViewer DIALOGS.H



THistoryViewer is a rather straightforward descendant of **TListViewer**. It is used by the history list system, and appears inside the history window set up by clicking on the history icon. For details on how **THistory**, **THistoryWindow**, and **THistoryViewer** cooperate, see the entry for **THistory** on page 289.

Data member

historyId `ushort historyId;`

historyId is the ID number of the history list to be displayed in the view.

Member functions

constructor

```
THistoryViewer(const TRect& bounds, TScrollBar *aHScrollBar,
               TScrollBar *aVScrollBar, ushort aHistoryId);
```

Initializes the viewer list by first calling the **TListViewer** constructor to set up the boundaries, a single column, and the two scroll bar pointers passed in *aHScrollBar* and *aVScrollBar*. The view is then linked to a history list, with the *historyID* data member set to the value passed in *aHistory*. That list is then checked for length, so the range of the list is set to the number of items in the list. The first item in the history list is given the focus, and the horizontal scrolling range is set to accommodate the widest item in the list.

See also: **TListViewer::TListViewer**

getPalette

```
virtual TPalette& getPalette() const;
```

Returns the default palette string, *cpHistoryViewer*, "\x06\x06\x07\x06\x06".

getText

```
virtual void getText(char *dest, short item, short maxLen);
```

Set *dest* to the *item*'th string in the associated history list. **getText** is called by the virtual **draw** member function for each visible item in the list.

See also: **TListViewer::draw**

handleEvent

```
virtual void handleEvent(TEvent& event);
```

The history viewer handles two kinds of events itself; all others are passed to **TListViewer::handleEvent**. Double clicking or pressing the *Enter* key terminates the modal state of the history window with a *cmOk* command. Pressing the *Esc* key, or any *cmCancel* command event, cancels the history list selection.

See also: **TListViewer::handleEvent**

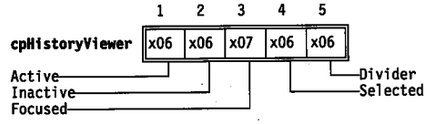
historyWidth

```
int historyWidth();
```

Returns the length of the longest string in the history list associated with *historyID*.

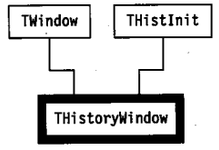
Palette

History viewer objects use the default palette *cpHistoryViewer* to map onto the sixth and seventh entries in the standard dialog box palette.



THistoryWindow

dialogs.h



For details on the use of history lists and their associated objects, see the entry for ***THistory*** *on page 289.*

THistoryWindow is a specialized descendant of **TWindow** and **THistInit** (multiple inheritance) used for holding a history list viewer when the user clicks on the history icon next to an input line. By default, the window has no title and no number. The history window's frame has only a close icon: the window can be closed, but not resized or zoomed.

Data member

viewer

```
TListViewer *viewer;
```

protected

viewer points to the list viewer to be contained in this history window.

Member functions

constructor

```
THistoryWindow(const TRect& bounds, ushort aHistoryId);
```

Calls the **THistInit** constructor with the argument **&THistoryWindow::initViewer**. This creates the list viewer. Next, the **TWindow** constructor is called to set up a window with the given *bounds*, a null title string, and no window number (*wnNoNumber*). Then the **TWindowInit** constructor is called with the argument **&THistoryWindow::initFrame** to create a frame for the history window. Finally, the **TWindow::flags** data member is set to *wfClose* to provide a close icon, and a history viewer object is created and

inserted in the history window to show the items in the history list given by *ahistoryID*.

See also: **TWindow** constructor, **THistoryWindow::initViewer**

getPalette

```
virtual TPalette& getPalette() const;
```

Returns the default palette string, *cpHistoryWindow*, "\x13\x13\x15\x18\x17\x13\x14\".

getSelection

```
virtual void getSelection(char * dest);
```

Returns in *dest* the string value of the focused item in the associated history viewer.

See also: **THistoryViewer::getText**

initViewer

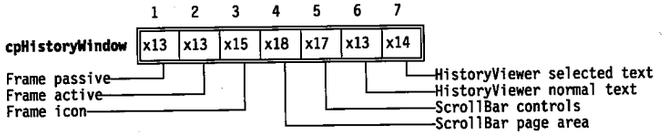
```
static TListViewer *initViewer(TRect bounds, TWindow *w, ushort
aHistoryId);
```

Instantiates and inserts a **THistoryViewer** object inside the boundaries of the history window for the list associated with the ID *aHistoryId*. Standard scroll bars are placed on the frame of the window to scroll the list.

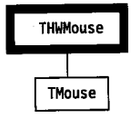
See also: **THistoryViewer** constructor

Palette

History window objects use the default palette *cpHistoryWindow* to map onto the 19th through 25th entries in the standard dialog box palette.



THWMouse SYSTEM.H



THWMouse provides low-level mouse handling functions for its derived class **TMouse**. These, and the other systems classes in SYSTEM.H, are

listed briefly for guidance only: they are used internally by Turbo Vision and you would not need to use them explicitly for normal applications.

Data members

buttonCount

```
static uchar near buttonCount;
```
protected

Holds the number of buttons on the mouse, or 0 if no mouse is detected.

Member functions

constructor

```
THWMouse();
```
protected

```
THWMouse( const THWMouse& m);
```
protected

Calls **THWMouse::resume**.

See also: **THWMouse::resume**

destructor

```
~THWMouse();
```
protected

Calls **THWMouse::suspend**.

See also: **THWMouse::suspend**

getEvent

```
static void getEvent( MouseEventType& me );
```
protected

Gets a mouse event from the event queue and sets the *buttons*, *where.x*, *where.y* and *doubleClick* data members of the **MouseEventType** structure, **me**.

See also: **MouseEventType**

hide

```
static void hide();
```
protected

Hides the mouse cursor.

present

```
static Boolean present();
```
protected

Returns *True* if a mouse is physically present and active; otherwise returns *False*.

registerHandler

```
static void registerHandler( unsigned mask, void (far *func)()
);
```
protected

Registers **func** as the current mouse handler, and sets *handlerInstalled* as *True*.

resume

```
static void resume();
```
protected

Restores the mouse by (re-)registering the handler and (re-)setting *buttonCount*.

setRange `static void setRange( ushort rx, ushort ry );` **protected**

Sets the mouse range to the given *x, y* arguments.

show `static void show();` **protected**

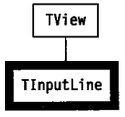
Displays the mouse cursor.

suspend `static void suspend();` **protected**

Does nothing if **present** returns *False*; otherwise, hides the mouse, unregisters the handler, and sets *buttonCount* to zero.

See also: **THWMouse::present**

TInputLine DIALOGS.H



A **TInputLine** object provides a basic input line string editor. It handles keyboard input and mouse clicks and drags for block marking and a variety of line editing functions (see **TInputLine::handleEvent**). The selected text is deleted and then replaced by the first text input. If *maxLen* is greater than the *x* dimension (*size.x*), horizontal scrolling is supported and indicated by left and right arrows.

The **getData** and **setData** member functions are available for writing and reading data strings (referenced via the *data* string data member) into the given record. **TInputLine::setState** simplifies the redrawing of the view with appropriate colors when the state changes from or to *sfActive* and *sfSelected*.

An input line frequently has a **TLabel** and/or a **THistory** object associated with it.

TInputLine can be extended to handle data types other than strings. To do so, you'll generally add additional data members and then override the constructors and the **store, valid, dataSize, getData**, and **setData** member functions. For example, to define a numeric input line, you might want it to contain minimum and maximum allowable values which will be tested by the **valid** function. These minimum and maximum data members

would be loaded (with the load constructor) and **store**d on the stream. **valid** would be modified to make sure the value was numeric and within range. **dataSize** would be modified to include the size of the new range data members (probably **sizeof**(*long*) for each). Oddly enough, in this example it would not be necessary to add a data member to store the numeric value itself. It could be stored as a string value (which is already managed by **TInputLine**) and converted from string to numeric value and back by **getData** and **setData**, respectively.

Data members

curPos `int curPos;`

Index to insertion point (that is, to the current cursor position).

See also: **TInputLine::selectAll**

data `char *data;`

The string containing the edited information.

firstPos `int firstPos;`

Index to the first displayed character.

See also: **TInputLine::selectAll**

maxLen `int maxLen;`

Maximum length allowed for string to grow (excluding the final 0).
See also: **TInputLine::dataSize**

selEnd `int selEnd;`

Index to the end of the selection area (that is, to the last character block marked).

See also: **TInputLine::selectAll**

selStart `int selStart;`

Index to the beginning of the selection area (that is, to the first character block marked).

See also: **TInputLine::selectAll**

Member functions

constructor `TInputLine(const TRect& bounds, int aMaxLen);`

Creates an input box control with the given values by calling **TView**(*bounds*). *state* is then set to *sfCursorVis*, *options* is set to (*ofSelectable* | *ofFirstClick*), and *maxLen* is set to *aMaxLen*. Memory is allocated and cleared for *aMaxlen* + 1 bytes and the *data* data member set to point at this allocation.

constructor `TInputLine( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**, *sfCursorVis*, *ofSelectable*, *ofFirstClick*

destructor `~InputLine();`

Deletes the *data* memory allocation, then calls **~TView** to destroy the **TInputLine** object.

See also: **~TView**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

dataSize `virtual ushort dataSize();`

Returns the size of the record for **TInputLine::getData** and **TInputLine::setData** calls. By default, it returns *maxLen* + 1. Override this member function if you define descendants to handle other data types.

See also: **TInputLine::getData**, **TInputLine::setData**

draw `virtual void draw();`

Draws the input box and its data. The box is drawn with the appropriate colors depending on whether the box is *sfFocused* (that is, whether the box view owns the cursor), and arrows are drawn if the input string exceeds the size of the view (in either or both directions). Any selected (block-marked) characters are drawn with the appropriate palette.

getData `virtual void getData(void *rec);`

Writes the number of bytes (obtained from a call to **dataSize**) from the *data* string to the array given by *rec*. Used with **setData** for a variety of applications; for example, temporary storage, or passing on the input string to other views. Override **getData** if you define **TInputLine** descendants to handle non-string data types. You can also use **getData** to convert from a string to other data types after editing by **TInputLine**.

See also: **TInputLine::dataSize**, **TInputLine::setData**

getPalette `virtual TPalette& getPalette() const;`

Returns the default palette string, *cpInputLine*, "\x13\x13\x14\x15".

handleEvent `void handleEvent(TEvent& event);`

Calls **TView::handleEvent**, then handles all mouse and keyboard events if the input box is selected. This member function implements the standard editing capability of the input box.

Editing features include: block marking with mouse click and drag; block deletion; insert or overwrite control with automatic cursor shape change; automatic and manual scrolling as required (depending on relative sizes of the *data* string and *size.x*); manual horizontal scrolling via mouse clicks on the arrow icons; manual cursor movement by arrow, *Home*, and *End* keys (and their standard control-key equivalents); character and block deletion with *Del* and *Ctrl-G*. The view is redrawn as required and the **TInputLine** data members are adjusted appropriately.

See also: *sfCursorIns*, **TView::handleEvent**, **TInputLine::selectAll**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream**

selectAll `void selectAll(Boolean enable);`

Sets *curPos*, *firstPos*, and *selStart* to 0. If *enable* is set to *True*, *selEnd* is set to the length of the *data* string, thereby selecting the whole input line; if *enable* is set to *False*, *selEnd* is set to 0, thereby deselecting the whole line. Finally, the view is redrawn by calling **drawView**.

See also: **TView::drawView**

setData `virtual void setData(void *rec);`

By default, copies the number of bytes (as returned by **dataSize**) from the *rec* array to the *data* string, and then calls **selectAll**(*True*). This zeros *curPos*, *firstPos*, and *selStart*. Finally, **drawView** is called to redraw the input box.

Override **setData** if you define descendants to handle non-string data types. You also use **setData** to convert other data types to a string for editing by **TInputLine**.

See also: **TInputLine::dataSize**, **TInputLine::getData**, **TInputLine::selectAll**

setState `virtual void setState(ushort aState, Boolean enable);`

Called when the input box needs redrawing (for example, if the palette is changed) following a change of *state*. Calls **TView::setState** to set or clear the view's *state* with the given *aState* bit(s). Then if *aState* is *sfSelected* (or *sfActive* and the input box is *sfSelected*), **selectAll**(*enable*) is called (which, in turn, calls **drawView**).

See also: **TView::setState**, **TInputLine::selectAll**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

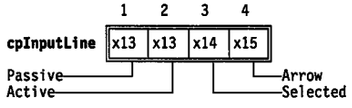
See also: **TStreamableClass**, **TStreamable**, **opstream**

Related functions

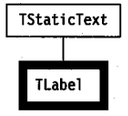
Certain operator functions are related to **TInputLine** but are not member functions; see page 232 for more information.

Palette

Input lines use the default palette, *cpInputLine*, to map onto the 19th through 21st entries in the standard dialog palette.



TLabel DIALOGS.H



A **TLabel** object is a piece of text in a view that can be selected (highlighted) by a mouse click, cursor keys, or *Alt-letter* hot key. The label is usually "attached" via a pointer to **TView** (called *link*) to some other control view such as an input line, cluster, or list viewer to guide the user. Selecting (or "pressing") the label selects the attached control. Conversely, the label is highlighted when the linked control is selected.

Data members

light `Boolean light;`

If *True*, the label and its linked control has been selected and will be highlighted. Otherwise, *light* is set to *False*.

link `TView *link;`

Pointer to the **TView** control associated with this label.

Member functions

constructor `TLabel(const TRect& bounds, const char *aText, TView *aLink);`

Creates a **TLabel** object of the given size and text by calling **TStaticText**(*bounds*, *aText*), then setting the *link* data member to *aLink* for the associated control (make *aLink* 0 if no control is needed). The *options* data member is set to *ofPreProcess* and *ofPostProcess*. The *eventMask* is set to *evBroadcast*. *aText* can designate a hot key letter for the label by surrounding the letter with tildes (~).

constructor `TLabel( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by

calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TStaticText::TStaticText**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws the label with the appropriate colors from the default palette.

getPalette `virtual TPalette& getPalette() const;`

Returns the default palette string, *cpLabel*, "\x07\x08\x09\x09".

handleEvent `virtual void handleEvent(TEvent& event);`

Handles all events by calling **TStaticText::handleEvent**. If an *evMouseDown* or hot key event is received, the appropriate linked control (if any) is selected with `Link->Select`. **handleEvent** also handles *cmReceivedFocus* and *cmReleasedFocus* broadcast events from the linked control in order to adjust the value of the *light* data member and redraw the label as necessary.

See also: **TView::handleEvent**, *cmXXXX* command constants

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

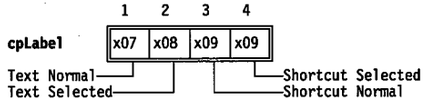
See also: **TStreamableClass, TStreamable, opstream**

Related functions

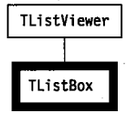
Certain operator functions are related to **TLabel** but are not member functions; see page 232 for more information.

Palette

Labels use the default palette, *cpLabel*, to map onto the seventh, eighth, and ninth entries in the standard dialog palette.



TListBox DIALOGS.H



TListBox is derived from **TListViewer** to help you set up the most commonly used list boxes, namely those displaying collections of strings, such as file names. **TListBox** objects represent displayed lists of such items in one or more columns with an optional vertical scroll bar. The horizontal scroll bars of **TListViewer** are not supported. The inherited **TListViewer** member functions let you select (and highlight) items by mouse and keyboard cursor actions. **TListBox** does not override **TListViewer::handleEvent** or **TListViewer::draw**, so you should refer to the sections describing these before using **TListBox** in your applications.

TListBox has an additional (private) data member called *items* not found in **TListViewer**. *items* points to a **TCollection** object that provides the items to be listed and selected. The public member function **list** returns the *items* pointer. Inserting data into the **TCollection** object is your responsibility, as are the actions to be performed when an item is selected.

TListViewer inherits its destructor from **TView**, so it is also your responsibility to dispose of the contents of *items* when you are finished with it. A call to **newList** disposes of the old list, so calling **newList**(0) and then disposing of the list box will free everything.

Data member

items `TCollection *items;` **private**

items points at the collection of items to scroll through. Typically, this might be a collection of strings representing the item texts. User can access this private member only by calling the function **list**.

See also: **TListBox::list**

Member functions

constructor `TListBox(const TRect& bounds, ushort aNumCols, TScrollBar *aScrollBar);`

Creates a list box control with the given size, number of columns, and a vertical scroll bar referenced by the *aScrollBar* pointer. This constructor calls **TListViewer**(*bounds, aNumCols, 0, aScrollBar*), thereby supressing the horizontal scroll bar.

The *list* data member is initially empty collection, and the inherited *range* data member is set to zero. Your application must provide a suitable **TCollection** holding the strings (or other objects) to be listed. The *list* data member must be set to point to this collection using **newList**.

constructor `TListBox(StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TListViewer** constructor, **TListBox::newList**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

dataSize `virtual ushort dataSize();`

Returns the size of the data read and written to the records passed to **getData** and **setData**. These three member functions are useful for initializing groups. By default, **TListBox::dataSize** returns the size of

&TCollection plus the size of **ushort** (for *items* and the selected item). You may need to override this member function for your own applications.

See also: **TListBox::getData, TListBox::setData**

getData `virtual void getData(void *rec);`

Writes **TListBox** object data to the target record. By default, **getData** writes the current *items* and *focused* data members to *rec*. You may need to override this member function for your own applications.

See also: **TListBox::dataSize, TListBox::setData**

getText `virtual void getText(char *dest, short item, short maxLen);`

Sets a string in *dest* from the calling **TListBox** object. By default, the returned string is obtained from the *item*'th item in the **TCollection** using *(char *)((list())->at(item))*. If **list** returns a collection containing non-string objects, you will need to override this member function. If **list** returns 0, **getText** sets *dest* to `" "`.

See also: **TCollection::at**

list `TCollection *list();`

list returns the private *items* pointer.

See also: **TListBox::items**

newList `virtual void newList(TCollection *aList);`

Creates a new list by deleting the current one and replacing it with the given *aList*.

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

setData `virtual void setData(void *rec);`

Replaces the current list with *items* and *focused* values read from the given *rec* array. **setData** calls **newList** so that the new list is displayed with the correct focused item. As with **getData** and **dataSize**, you may need to override this member function for your own applications.

See also: **TListBox::dataSize, TListBox::getData, TListBox::newList**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

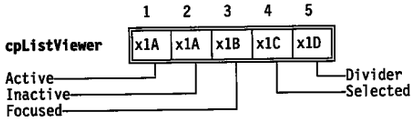
See also: **TStreamableClass, TStreamable, opstream**

Related functions

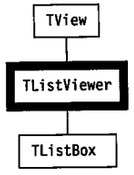
Certain operator functions are related to **TListBox** but are not member functions; see page 232 for more information.

Palette

List boxes use the default palette, *cpListViewer*, to map onto the twenty-sixth through twenty-ninth entries in the standard application palette.



TListViewer VIEWS.H



TListViewer is an abstract class from which you can derive list viewers of various kinds, such as **TListBox**. **TListViewer**'s members offer the following functionality:

- A view for displaying linked lists of items (but no list)
- Control over one or two scroll bars
- Basic scrolling of lists in two dimensions
- Reading and writing the view and its scroll bars from and to a stream
- Ability to use a mouse or the keyboard to select (highlight) items on list
- **draw** member function that copes with resizing and scrolling

TListViewer has a pure virtual **getText** member function, so you need to supply the mechanism for creating and manipulating the text of the items to be displayed. **TListViewer** has no list storage mechanism of its own. Use it to display scrollable lists of arrays, linked lists, or similar data structures. You can also use classes derived from **TListViewer**, such as **TListBox**, which associates a collection with a list viewer.

Data members

focused
```
short focused;
```
The item number of the focused item. Items are numbered from 0 to *range* – 1. Initially set to 0, the first item, *focused*, can be changed by mouse click or *Spacebar* selection.

See also: **TListViewer::range**

hScrollBar
```
TScrollBar *hScrollBar;
```
Pointer to the horizontal scroll bar associated with this view. If 0, the view does not have such a scroll bar.

numCols
```
short numCols;
```
The number of columns in the list control.

range
```
short range;
```
The current total number of items in the list. Items are numbered from 0 to *range* – 1.

See also: **TListViewer::setRange**

topItem
```
short topItem;
```
The item number of the top item to be displayed. This value changes during scrolling. Items are numbered from 0 to *range* – 1. This number depends on the number of columns, the size of the view, and the value of *range*.

See also: **TListViewer::range**

vScrollBar
```
TScrollBar *vScrollBar;
```
Pointer to the vertical scroll bar associated with this view. If 0, the view does not have such a scroll bar.

Member functions

constructor
```
TListViewer(const TRect& bounds, ushort aNumCols, TScrollBar
            *aHScrollBar, TScrollBar *aVScrollBar);
```

Creates and initializes a **TListViewer** object with the given size by first calling **TView**(*bounds*). The *numCols* data member is set to *aNumCols*. *options* is set to (*ofFirstClick* | *ofSelectable*) so that mouse clicks that select this view will be passed first to **TListViewer::handleEvent**. The *eventMask* is set to *evBroadcast*. The initial values of *range* and *focused* are zero. You can supply pointers to vertical and/or horizontal scroll bars by way of the *aVScrollBar* and *aHScrollBar* arguments. Setting either or both to 0 suppresses one or both scroll bars. These two pointer arguments are assigned to the *vScrollBar* and *hScrollBar* data members.

If you provide valid scroll bars, their *pgStep* and *arStep* data members will be adjusted according to the **TListViewer** size and number of columns. For a single-column **TListViewer**, for example, the default vertical *pgStep* is *size.y* – 1, and the default vertical *arStep* is 1.

constructor `TListViewer( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView, TScrollBar::setStep**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

changeBounds `virtual void changeBounds(TRect& bounds);`

Changes the size of the **TListViewer** object by calling **TView::changeBounds**(*bounds*). If a horizontal scroll bar has been assigned, *pgStep* is updated by way of *setStep*.

See also: **TView::changeBounds, TScrollBar::setStep**

draw `virtual void draw();`

Draws the **TListViewer** object with the default palette by repeatedly calling **getText** for each visible item. Takes into account the focused and selected items and whether the view is *sfActive*.

See also: **TListViewer::getText**

focusItem `virtual void focusItem(short item);`

Makes the given item focused by setting the *focused* data member to *item*. Also sets the *value* data member of the vertical scroll bar (if any) to *item* and adjusts *topItem*.

See also: **TListViewer::isSelected, TScrollBar::setValue**

focusItemNum `virtual void focusItemNum(short item);`

Used internally by **focusItem**. Makes the given item focused by setting the *focused* data member to *item*.

See also: **TListViewer::focusItemNum**

getPalette `virtual TPalette& getPalette() const;`

Returns *cpListViewer*, the default **TListViewer** palette string, "\x1A\x1A\x1B\x1C\x1D\".

getText `virtual void getText(char *dest, short item, short maxLen)=0;`

This is a pure virtual function. Derived classes must either redeclare this member as a pure virtual function or override it with a function that returns a string not exceeding *maxLen*, given an item index referenced by *item*. Note that **TListViewer::draw** needs to call **getText**.

See also: **TListViewer::draw**

handleEvent `virtual void handleEvent(TEvent& event);`

Handles events by calling **TView::handleEvent**(*event*). Mouse clicks and "auto" movements over the list will change the focused item. Items can be selected with double mouse clicks. Keyboard events are handled as follows: *Spacebar* selects the currently focused item; the arrow keys, *PgUp*, *PgDn*, *Ctrl-PgDn*, *Ctrl-PgUp*, *Home*, and *End* keys are tracked to set the focused item. Broadcast events from the scroll bars are handled by changing the focused item and redrawing the view as required.

See also: **TView::handleEvent, TListViewer::focusItem**

isSelected `virtual Boolean isSelected(short item);`

Returns *True* if the given *item* is selected (focused), that is, if *item* == *focused*.

See also: **TListViewer.focusItem**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

selectItem `virtual void selectItem(short item);`

Selects the *item*'th element of the list, then broadcasts this fact to the owning group by calling:

```
message(owner, evBroadcast, cmListItemSelected, this);
```

See also: **TListViewer::focusItem, message**

setRange `void setRange(short aRange);`

Sets the *range* data member to *aRange*. If a vertical scroll bar has been assigned, its parameters are adjusted as necessary (and **TScrollBar::drawView** is invoked if redrawing is needed). If the currently focused item falls outside the new *range*, the *focused* data member is set to zero.

See also: **TListViewer::range, TScrollBar::setParams**

setState `virtual void setState(ushort aState, Boolean enable);`

Calls **TView::setState**(*aState, enable*) to change the **TListViewer** object's state if *enable* is *True*. Depending on the *aState* argument, this can result in displaying or hiding the view. Additionally, if *aState* is *sfSelected* and *sfActive*, the scroll bars are redrawn; if *aState* is *sfSelected* but not *sfActive*, the scroll bars are hidden.

See also: **TView::setState, TScrollBar::show, TScrollBar::hide**

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

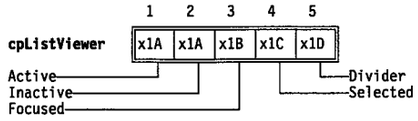
See also: **TStreamableClass, TStreamable, opstream**

Related functions

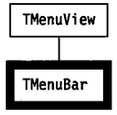
Certain operator functions are related to **TListViewer** but are not member functions; see page 232 for more information.

Palette

List viewers use the default palette, *cpListViewer*, to map onto the twenty-sixth through twenty-ninth entries in the standard application palette.



TMenuBar MENUS.H



TMenuBar objects represent the horizontal menu bars from which menu selections can be made by:

- direct clicking
- *F10* selection and hot keys
- selection (highlighting) and pressing *Enter*
- hot keys

The main menu selections are displayed in the top menu bar. This is represented by an object of type **TMenuBar**, usually owned by your **TApplication** object. Submenus are displayed in objects of type **TMenuBox**. Both **TMenuBar** and **TMenuBox** are derived from **TMenuView** (which is in turn derived from **TView**).

For most Turbo Vision applications, you will not be involved directly with menu objects. By overriding **TApplication::initMenuBar** with a suitable set of nested `new TMenuItem` and `new TMenu` calls, Turbo Vision takes care of all the standard menu mechanisms.

TM
Std class

Member functions

constructor `TMenuBar(const TRect& bounds, TMenu *aMenu);`

Creates a menu bar by calling **TMenuView**(*bounds*). The grow mode is set to *gfGrowHiX*. The *options* data member is set to *ofPreProcess* to allow hot keys to operate. The *menu* data member is set to *aMenu*, providing the menu selections.

constructor `TMenuBar( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TMenuView::TMenuView**, *gfXXXX* grow mode flags, *ofXXXX* option flags, **TMenuView::menu**, **TMenu**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**, **TStreamable**

draw `virtual void draw();`

Draws the menu bar with the default palette. The *name* and *disabled* data members of each **TMenuItem** object in the *menu* linked list are read to give the menu legends in the correct colors. The *current* (selected) item is highlighted.

getItemRect `virtual TRect getItemRect(TMenuItem *item);`

Overrides **TMenuView::getItemRect**. Returns the rectangle occupied by the given menu item. It can be used with **mouseInView** to determine if a mouse click has occurred on a given menu selection.

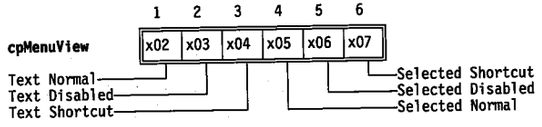
See also: **TMenuView::getItemRect**, **TView::mouseInView**

Related functions

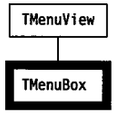
Certain operator functions are related to **TMenuBar** but are not member functions; see page 232 for more information.

Palette

Menu bars, like all menu views, use the default palette *cpMenuView* to map onto the second through seventh entries in the standard application palette.



TMenuBox MENUS.H



TMenuBox objects represent vertical menu boxes. These can contain arbitrary lists of selectable actions, including submenu items. As with menu bars, color coding is used to indicate disabled items. Menu boxes can be instantiated as submenus of the menu bar or other menu boxes, or can be used alone as pop-up menus.

Member functions

constructor
```
TMenuBox(const TRect& bounds, TMenu *aMenu);
TMenuBox(const TRect& bounds, TMenu *aMenu, TMenuView *aParentMenu=0);
```

Creates a **TMenuBox** object by calling **TMenuView**(*bounds*). The *bounds* parameter is then adjusted to accommodate the width and length of the items in *aMenu*.

The *ofPreProcess* bit in the *options* data member is set so that hot keys will operate. *state* is set to include *sfShadow*. The *menu* data member is set to *aMenu*, which provides the menu selections. The second form of the constructor allows an explicit *aParentMenu* argument (defaulting to 0) which is set to *parentMenu*.

constructor `TMenuBox( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TMenuView::TMenuView**, *sfXXXX* state flags, *ofXXXX* option flags, **TMenuView::menu**, **TMenuView::parentMenu**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

TM
Std class

draw `virtual void draw();`

Draws the framed menu box and associated menu items in the default colors.

getItemRect `virtual TRect getItemRect(TMenuItem *item);`

Overrides **TMenuView::getItemRect**. Returns the rectangle occupied by the given menu item. It can be used to determine if a mouse click has occurred on a given menu selection.

See also: **TMenuView::getItemRect, TView::mouseInView**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

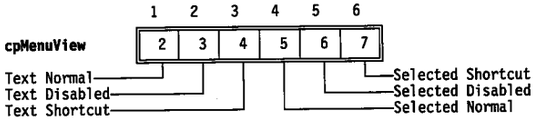
See also: **TStreamableClass, TStreamable, opstream**

Related functions

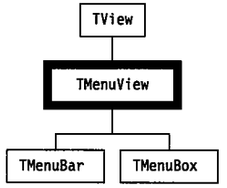
Certain operator functions are related to **TMenuBox** but are not member functions; see page 232 for more information.

Palette

Menu boxes, like all menu views, use the default palette *cpMenuView* to map onto the second through seventh entries in the standard application palette.



TMenuView MENUS.H



TMenuView provides an abstract base from which menu bar and menu box classes (either pull down or pop up) are derived. You cannot instantiate a **TMenuView** itself.

Data members

current `TMenuItem *current;`

A pointer to the currently selected menu item.

menu `TMenu *menu;`

A pointer to the **TMenu** object for this menu, which provides a linked list of menu items. The *menu* pointer allows access to all the data members of the menu items in this menu view.

See also: **TMenuView::findItem, TMenuView::getItemRect, TMenu struct**

parentMenu `TMenuView *parentMenu;`

A pointer to the **TMenuView** object (or any class derived from **TMenuView**) that owns this menu. Note that **TMenuView** is not a group. Ownership here is a much simpler concept than **TGroup** ownership, allowing menu nesting: the selection of submenus and the return back to the "parent" menu. Selections from menu bars, for example, usually result in a submenu being "pulled down." The menu bar in that case is the parent menu of the menu box.

See also: **TMenuBox::TMenuBox**

Member functions

constructor `TMenuView(const TRect& bounds);`

Calls **TView::TView**(*bounds*) to create a **TMenuView** object of size *Bounds*. The *current* **TMenuItem**, *parentMenu* and *menu* pointers are set to 0. The default *eventMask* is set to *evBroadcast*. The **TMenuView** constructors are called by the derived classes, **TMenuBar** and **TMenuBox**, and would rarely, if ever, be invoked directly.

constructor `TMenuView( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**, *evBroadcast*, **TMenuBar::TMenuBar**, **TMenuBox::TMenuBox**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

execute `virtual ushort execute();`

Executes a menu view until the user selects a menu item or cancels the process. Returns the command assigned to the selected menu item, or zero if the menu was canceled. This member function should *never* be called except by **execView**.

See also: **TGroup::execView**

findItem `TMenuItem *findItem(char ch);`

Returns a pointer to the menu item that has **toupper**(*ch*) as its hot key (the highlighted character). Returns 0 if no such menu item is found or if the menu item is disabled. Note that *ch* is case insensitive.

getHelpCtx `virtual ushort getHelpCtx();`

By default, this member function returns the help context of the current menu selection. If this is *hcNoContext*, the parent menu's current context is checked. If there is no parent menu, **getHelpCtx** returns *hcNoContext*.

See also: *hcXXXX* help context constants

getItemRect `virtual TRect getItemRect(TMenuItem *item);`

Classes derived from **TMenuView** *must* override this member function in order to respond to mouse events. Your overriding functions in derived classes must return the rectangle occupied by the given menu item. It is used with **mouseInView** to determine if a mouse click has occurred on a given menu selection, and thence determine the required action.

See also: **TMenuBar::getItemRect**, **TMenuBox::getItemRect**, **TView::mouseInView**

getPalette `virtual TPalette& getPalette() const;`

Returns the default palette string, *cpMenuView*, "\x02\x03\x04\x05\x06\x07".

handleEvent `virtual void handleEvent(TEvent& event);`

Called whenever a menu event needs to be handled. Determines which menu item has been mouse or keyboard selected (including hot keys) and generates the appropriate command event with **putEvent**.

See also: **TView::handleEvent**, **TView::putEvent**, **TMenuView::hotKey**

hotKey `TMenuItem *hotKey(ushort keyCode);`

Returns a pointer to the menu item associated with the hot key given by *keyCode*. Returns 0 if no such menu item exists, or if the item is disabled. Hot keys are usually function keys or *Alt* key combinations, determined by **TProgram::initMenuBar**. **hotKey** is used by **handleEvent** to determine whether a keystroke event selects an item in the menu.

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

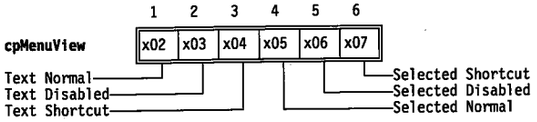
See also: **TStreamableClass**, **TStreamable**, **opstream**

Related functions

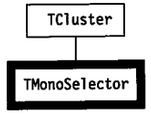
Certain operator functions are related to **TMenuView** but are not member functions; see page 232 for more information.

Palette

All menu views use the default palette *cpMenuView* to map onto the second through seventh entries in the standard application palette.



TMonoSelector COLORSEL.H



The interrelated classes **TColorItem**, **TColorGroup**, **TColorSelector**, **TMonoSelector**, **TColorDisplay**, **TColorGroupList**, **TColorItemList**, and **TColorDialog** are used to provide viewers and dialog boxes from which the user can select and change the color assignments from available palettes with immediate effect on the screen.

TMonoSelector implements a cluster from which you can select normal, highlight, underline, or inverse video attributes on monochrome screens.

Member functions

constructor `TMonoSelector( const TRect& bounds );`

Creates a cluster by calling the **TCluster** constructor with four buttons labeled: "Normal", "Highlight", "Underline", and "Inverse". The *evBroadcast* flag is set in *eventMask*.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**, **TStreamable**

draw `virtual void draw();`

Draws the selector cluster.

handleEvent `virtual void handleEvent( TEvent& event );`

Calls **TCluster::handleEvent** and responds to *cmColorSet* events by changing the data member *value* accordingly. The view is redrawn if necessary. *value* actually holds a video attribute corresponding to the selected attribute.

See also: **TCluster::handleEvent**, **TCluster::value**

mark `virtual Boolean mark( int item );`

Returns *True* if the *item*'th button has been selected; otherwise returns *False*.

movedTo `void movedTo( int item );`

Sets *value* to the *item*'th attribute (same effect as **press**).

See also: **TMonoSelector::press**

newColor `void newColor();`

Informs the owning group if the attribute has changed.

`virtual void press( int item );`

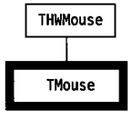
"Presses" the *item*'th button and calls **newColor**.

See also: **TMonoSelector::newColor**

Related functions

Certain operator functions are related to **TMonoSelector** but are not member functions; see page 232 for more information.

TMouse SYSTEM.H



TMouse provides low-level mouse handling functions. These, and the other systems classes in SYSTEM.H, are listed briefly for guidance only: they are used internally by Turbo Vision and you would not need to use them explicitly for normal applications.

Member functions

constructor `TMouse();`

Calls **TMouse::show** to display the mouse cursor.

See also: **TMouse::show**

destructor `~TMouse();`

Calls **THWMouse::hide** to hide the mouse cursor.

See also: **THWMouse::hide**

getEvent `static void getEvent( MouseEventType& me);`

Calls `THWMouse::getEvent( me )` to set the *buttons*, *where.x*, *where.y* and *doubleClick* data members of the MouseEventType structure, **me**.

See also: **THWMouse::getEvent, MouseEventType**

hide `static void hide();`

Calls **THWMouse::hide** to hide the mouse cursor.

See also: **THWMouse::hide**

present `static Boolean present();`

Calls **THWMouse::present**. Returns *True* if a mouse is active (that is, if *buttonCount* is nonzero); otherwise, returns *False*.

See also: **THWMouse::present**

registerHandler `static void registerHandler( unsigned mask, void (far *func)() );`

Calls `THWMouse::registerHandler( mask, func )` to register **func** as the current mouse handler.

See also: **THWMouse::registerHandler**

resume `static void resume();`

Calls **THWMouse::resume**. Restores the mouse by re-registering the handler and re-setting *buttonCount*.

See also: **THWMouse::resume**

setRange `static void setRange( ushort rx, ushort ry );`

Calls `THWMouse::setRange( rx, ry )` to set the mouse range to the given arguments.

See also: **THWMouse::setRange**

show `static void show();`

Calls **THWMouse::show** to display the mouse cursor.

See also: **THWMouse::show**

suspend `static void suspend();`

Calls **THWMouse::suspend**. Does nothing if **present** returns *False*; otherwise, hides the mouse, unregisters the handler, and sets *buttonCount* to zero.

See also: **THWMouse::suspend**

TMouseEventType SYSTEM.H

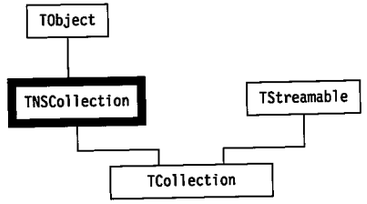
The **TMouseEventType** structure holds the data that characterizes a mouse event: button number, whether double-clicked, and the coordinates of the point where the click was detected.

```
struct MouseEventType
{
   uchar buttons;
   Boolean doubleClick;
   TPoint where;
};
```

See also: **TEvent::getMouseEvent, TView::handleEvent, THWMouse::getEvent**

TM
Std class

TNSCollection TVOBJS.H



TNSCollection implements a nonstreamable collection of items (hence the prefix *NS*,) including other objects. Its main purpose is to provide a base class (together with **TStreamable** via multiple inheritance) for the streamable collection class, **TCollection**. **TNSCollection** provides **TCollection** with the functions for adding, accessing, and removing items from a collection. **TStreamable** provides **TCollection** with the ability to name and create streams to which and from which collections can be written and read.

A collection is a more general concept than the traditional array, set, or list. **TNSCollection** objects size themselves dynamically at run time and offer a base for more specialized derived classes such as **TCollection**, **TNSSortedCollection**, **TSortedCollection**, **TStringCollection**, and **TResourceCollection**. In addition to member functions for adding and deleting items, **TNSCollection** offers several iterator routines that call a function for each item in the collection.

Data members

count `ccIndex count;` **protected**

The current number of items in the collection, up to *maxCollectionSize*.

See also: *maxCollectionSize* variable

delta `ccIndex delta;` **protected**

The number of items by which to increase the *items* list whenever it becomes full. If *delta* is zero, the collection cannot grow beyond the size set by *limit*.

See also: *limit*, **TNSCollection** constructor

items `void **items;` **protected**

A pointer to an array of generic item pointers.

limit `ccIndex limit;` **protected**

The currently allocated size (in elements) of the *items* list.

See also: *delta*, **TNSCollection** constructor

limit `Boolean shouldDelete;` **protected**

If set *True* (the default), the **TNSCollection** destructor will call **freeAll** before setting limit to 0. If set *False*, the destructor simply sets *limit* to 0.

See also: **TNSCollection::~TNSCollection**, **TNSCollection::freeAll**

Member functions

constructor `TNSCollection(ccIndex aLimit, ccIndex aDelta);`

Creates a collection with *limit* set to *aLimit* and *delta* set to *adelta*. *count* and *items* are both set to 0. *shouldDelete* is set *True*. The initial number of items will be limited to *aLimit*, but the collection is allowed to grow in increments of *aDelta* until memory runs out or the number of items reaches *maxCollectionSize*.

constructor `TNSCollection();`

This constructor sets *shouldDelete* to true and all other data members to 0.

See also: **TNSCollection::shouldDelete**, **TNSCollection::count**, **TNSCollection::limit**, **TNSCollection::delta**

destructor `~TNSCollection();`

If *shouldDelete* is *True*, the destructor removes and destroys all items in the collection by calling **TNSCollection::freeAll** and setting *limit* to 0. If *shouldDelete* is *False*, the destructor sets limit to zero but does not destroy the collection.

See also: **TNSCollection::shouldDelete**, **TNSCollection::freeAll**, **TNSCollection::setLimit**

at `void *at(ccIndex index);`

Returns a pointer to the item indexed by *index* in the collection. This member function lets you treat a collection as an indexed array. If *index* is less than zero or greater than or equal to *count*, the **error** member function is called with an argument of *coIndexError*, and 0 is returned.

See also: **TNSCollection::indexOf**

atInsert `void atInsert(ccIndex index, void *item);`

Moves the following items down by one position, then inserts *item* at the *index*'th position. If *index* is less than zero or greater than *count*, the **error** member function is called with an argument of *coIndexError* and the new *item* is not inserted. If *count* is equal to *limit* before the call to **atInsert**, the allocated size of the collection is expanded by *delta* items using a call to **setLimit**. If the **setLimit** call fails to expand the collection, the **error** member function is called with an argument of *coOverflow* and the new *item* is not inserted.

See also: **TNSCollection::at, TNSCollection::atPut**

atPut `void atPut(ccIndex index, void *item);`

Replaces the item at index position *index* with the given *item*. If *index* is less than zero or greater than or equal to *count*, the **error** member function is called with an argument of *coIndexError*.

See also: **TNSCollection::at, TNSCollection::atInsert**

atRemove `void atRemove(ccIndex index);`

Removes the item at the *index*'th position by moving the following items up by one position. The item itself is not destroyed. *count* is decremented by 1, but the memory allocated to the collection (as given by *limit*) is not reduced. If *index* greater than or equal to *count*, **error**(1,0) is called. Contrast the **atRemove** action with **atFree**. The latter does an **atRemove**, then destroys the item with **delete**(*item*).

See also: **TNSCollection::atFree, TNSCollection::remove**

error `static void error(ccIndex code, ccIndex info);`

Called whenever a collection error is encountered. By default, this member function produces a run-time error of (212 – *code*).

firstThat `void *firstThat(ccTestFunc Test, void *arg);`

firstThat applies a Boolean function ***Test,** along with an argument list given by *arg* (possibly empty), to each item in the collection until ***Test** returns *True*. The result is the item pointer for which ***Test** returns *True*, or 0 if ***Test** returns *False* for all items. *ccTestFunc* is typedefed as follows:

```
typedef Boolean (*ccTestFunc) (void *, void *);
```

The first pointer argument of ***Test** scans the collection. The second argument of ***Test** is set from the *arg* pointer of **firstThat**, as shown in the following implementation:


```
void *TNSCollection::firstThat( ccTestFunc Test, void *arg )
{
   for( ccIndex i = 0; i < count; i++ )
      {
      if( Test( items[i], arg ) == True )
         return items[i];
      }
   return 0;
}
```

Recall that the protected data member *items* is of type **void ****, so that `items[i]` is of type **void ***.

See also: **TNSCollection::lastThat, TNSCollection::forEach**

forEach `void forEach(ccAppFunc action, void *arg);`

The **forEach** iterator applies an action, given by the function ***action**, to each item in the collection. The *arg* pointer can be used to pass additional arguments to the action. *ccAppFunc* is typedef'd as follows:

```
typedef void (*ccAppFunc) ( void *, void *);
```

The first pointer argument of ***action** scans the collection. The second argument of ***action** is set from the *arg* pointer of **forEach**, as shown in the following implementation:

```
void TNSCollection::forEach( ccAppFunc action, void *arg )
{
   for( ccIndex i = 0; i < count; i++ )
      action( items[i], arg );
}
```

Recall that the protected data member *items* is of type **void ****, so that `items[i]` is of type **void ***.

See also: **TNSCollection::firstThat, TNSCollection::lastThat**

free `void free(void *item);`

Removes and destroys the given *item*. Equivalent to

```
remove( item );
delete( item );
```

See also: **TNSCollection::remove**

freeAll `void freeAll();`

Removes and destroys all items in the collection and sets *count* to 0.

See also: **TNSCollection::removeAll**

indexOf `virtual ccIndex indexOf(void *item);`

Returns the index of the given *item*; that is, the converse operation to **TNSCollection::at**. If *item* is not in the collection, **indexOf** calls **error**(1,0).

See also: **TNSCollection::at**

insert `virtual void insert(void *item);`

Inserts *item* into the collection, and adjusts other indexes if necessary. By default, insertions are made at the end of the collection by calling **atInsert**(*count*, *item*);

See also: **TNSCollection::atInsert**

lastThat `void *lastThat(ccTestFunc Test, void *arg);`

lastThat applies the Boolean function ***Test**, together with the *arg* argument list (possibly empty), to each item in the collection, starting at the last item, and scanning in reverse order until ***Test** returns *True*. The result is the item pointer for which ***Test** returns *True*, or 0 if ***Test** returns *False* for all items. *ccTestFunc* is typedef'd as follows:

```
typedef Boolean (*ccTestFunc) (void *, void *);
```

The first pointer argument of **Test** scans the collection. The second argument of **Test** is set from the *arg* pointer of **lastThat**, as shown in the following implementation:

```
void *TNSCollection::lastThat( ccTestFunc Test, void *arg )
{
   for( ccIndex i = count; i > 0; i-- )
      {
      if( Test( items[i-1], arg ) == True )
          return items[i-1];
      }
   return 0;
}
```

Recall that the protected data member *items* is of type **void ****, so that `items[i]` is of type **void ***.

See also: **TNSCollection::firstThat, TNSCollection::forEach**

pack `void pack();`

Deletes all null pointers in the collection and moves items up to fill any gaps.

See also: **TNSCollection::remove, TNSCollection::removeAll**

remove `void remove(void *item);`

Removes the item given by *item* from the collection. Equivalent to **atRemove(indexOf**(*item*)).

See also: **TNSCollection::atRemove, TNSCollection::indexOf**

removeAll `void removeAll();`

Removes all items from the collection by setting *count* to zero.

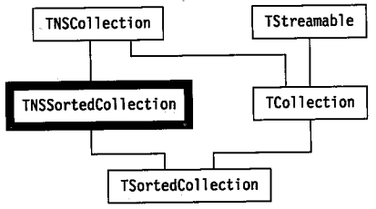
See also: **TNSCollection::remove, TNSCollection::atRemove**

setLimit `virtual void setLimit(ccIndex aLimit);`

Expands or shrinks the collection by changing the allocated size to *aLimit*. If *aLimit* is less than *count*, it is set to *count*, and if *aLimit* is greater than *maxCollectionSize*, it is set to *maxCollectionSize*. Then, if *aLimit* is different from the current *limit*, a new *items* array of *aLimit* elements is allocated, the old *items* array is copied into the new array, and the old array is deleted.

See also: **TNSCollection::limit, TNSCollection::count**, *maxCollectionSize* variable

TNSSortedCollection TVOBJS.H



The abstract class **TNSSortedCollection** is a specialized derivative of **TNSCollection** implementing non-streamable collections sorted by a key (with or without duplicates). No instances of **TNSSortedCollection** are allowed. It exists solely as a base for other standard or user-defined derived classes.

Sorting is implied by the pure virtual (and private) member function **compare**, which you *must* override (or redefine as pure virtual). Eventually, you would override it in your derived classes to provide a particular ordering of the collection. As new items are added they are

automatically inserted in the order given by **compare**. Items can be located using the binary (by default) **search** function (also virtual). The virtual **indexOf** function, which returns a pointer for **compare**, can also be overridden if **compare** needs additional information.

For streamable sorted collections, you must use the class **TSorted-Collection**, which is multiply derived from **TNSSortedCollection** and **TCollection** (which has **TStreamable** as a base). Apart from streamability, the two classes **TSortedCollection** and **TNSSortedCollection** offer the same functionality.

Data member

duplicates `Boolean duplicates;`

Set to *True* if duplicate indexes are allowed; otherwise set to *False*. The default is *False*. If *duplicate* is *True*, the *search*, *insert*, and *indexOf* member functions work differently (see these member function entries).

Member functions

constructor `TNSSortedCollection(ccIndex aLimit, ccIndex aDelta);`

Invokes the **TCollection** constructor to set *count*, *items*, and *limit* to zero; calls **setLimit**(*aLimit*) to set the collection limit to *aLimit*, then sets *delta* to *aDelta*. Note that *ccIndex* is a **typedef**ed **int**. *duplicates* is set to *False*. If you want to allow duplicate keys, you must set *duplicates* to *True*.

See also: **TCollection** constructor, **TCollection** data members

compare `virtual int compare(void *key1, void *key2) = 0;` **private**

compare is a pure virtual function that must be overridden in all derived classes. **compare** should compare the two key values, and return a result as follows:

< 0	if *key1* < *key2*
0	if *key1* = *key2*
> 0	if *key1* > *key2*

key1 and *key2* are generic pointers, as extracted from their corresponding collection items by the **keyOf** member function. The **search** member function implements a binary search through the collection's items using **compare** to compare the items.

insert `virtual void insert(void *item);`

If *duplicates* is *False*, **insert** works as follows: If the target item is not found in the sorted collection, it is inserted at the correct index position. Calls **search** to determine if the item exists, and if not, where to insert it. **insert** is implemented as follows:

```
{
   ccIndex i;

   if (search(keyOf(item), i) == 0)
      atInsert(i,item);
}
```

If *duplicates* is *True*, the item is inserted ahead of any items (if any) with the same key.

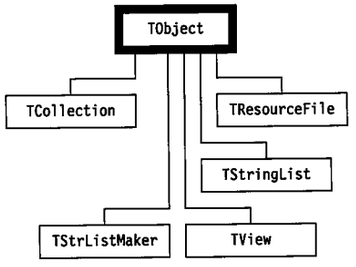
See also: **TNSSortedCollection::search**, **TCollection::atInsert**

search `virtual Boolean search(void *key, ccIndex& index);`

Returns *True* if the item identified by *key* is found in the sorted collection. If the item is found, *index* is set to the found index; otherwise *index* is set to the index where the item would be placed if inserted. Note that if *duplicates* is *True* and the key is duplicated, search will locate the *first* item that matches.

See also: **TNSSortedCollection::compare**, **TNSSortedCollection::insert**

TObject TVOBJS.H



TObject is the starting point for much of Turbo Vision's class hierarchy. It has no parents but many descendants. Apart from **TPoint** and **TRect** (and various initialization and stream management classes and structures), most of Turbo Vision's standard classes are ultimately derived from **TObject**.

Member functions

destructor `virtual ~TObject();`

Performs the necessary cleanup and disposal for dynamic objects.

destroy `static void destroy( TObject *ob);`

Whenever you want to delete an object (*ob*) of a type derived from **TObject** (that is, any object created with operator **new**), call **destroy**. **destroy** terminates the object, correctly freeing the memory that it occupies. Use this function in place of the C++ operator **delete**. For example,

```
TDialog *dlg = new TDialog( ... );
// delete dlg;    // DON'T DO THIS
destroy( dlg );   // DO IT THIS WAY
```

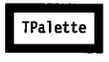
See also: Chapter 6, "Writing safe programs"

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

TPalette VIEWS.H



TPalette is a simple class used to create and manipulate palette arrays without bothering you with their internal structure. Although palettes are arrays of **char**, and are often referred to as strings, they are not the conventional null-terminated strings found in C. In fact, there may well be a 0-byte within a palette. Because of this, normal C string functions cannot be used. The first byte of a palette string holds its length (not counting the first byte itself). Each basic view has a default palette that determines (by indexing into its owner's palette) the usual colors assigned to the various parts of a view, such as scroll bars, frames, buttons, text, and so on. For a detailed discussion, see Chapter 4, "Views."

Member functions

constructor
```
TPalette( const char *d, ushort len );
TPalette( const TPalette& tp );
```

The first form creates a **TPalette** object with string *d* and length *len*. The private member *data* is set with *len* in its first byte, following by the array *d*. The second form creates a new palette by copying the palette *tp*.

destructor `~TPalette();`

Destroys the palette.

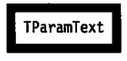
operator = `TPalette& operator = ( const TPalette& tp );`

The code `p = tp;` copies the palette *tp* to the palette *p*.

operator [] `char& operator[] ( int index );`

The subscripting operator returns the character at the *index*'th position.

TParamText DIALOGS.H



TParamText is derived from **TStaticText**. It uses parameterized text strings for formatted output using the **stdio** function **vsprintf**.

Data members

paramCount `short paramCount;`

paramCount gives the number of parameters contained in *paramList*.

See also: **TParamText::paramList**

paramList `void *paramList;`

paramList is a generic pointer, typically pointing to an array (or structure) of pointers or **long** values to be used as formatted parameters for a text string.

Member functions

constructor `TParamText(const TRect& bounds, const char *aText, int aParamCount);`

Creates and initializes a static text object by calling **TStaticText**(*bounds, aText*). *aText* can contain **printf** format specifiers (in the form %[-][nnn]X) that will be replaced by the parameters passed at run time. The parameter count, passed in *aParamCount*, is assigned to the *paramCount* data member.

constructor `TParamText( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TStaticText::TStaticText, vsprintf** (stdio.h)

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

dataSize `virtual ushort dataSize();`

Returns the size of the data required by the object's parameters.

getText `virtual void getText(char *s);`

Produces a formatted text string in *s*, produced by merging the parameters contained in *paramList* into the text string in *text*, using a call to **vsprintf**(*s, text, paramList*). If *text* is empty, **s* is set to *EOS*.

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

setData `virtual void setData(void *rec);`

Sets *paramList* to *&rec*.

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

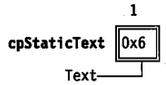
See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TParamText** but are not member functions; see page 232 for more information.

Palette

TParamText objects use the default palette *cpStaticText* to map onto the sixth entry in the standard dialog palette.



TPoint OBJECTS.H

TPoint is a class implementing points on the screen with several overloaded operators for point manipulation.

Data members

x `int x;`

x is the screen column of the point.

y `int y;`

y is the screen row of the point.

Member functions

operator += `TPoint& operator+=(const TPoint& adder);`

Increments *x* by *adder.x*, and *y* by *adder.y*. Returns **this*.

operator -= `TPoint& operator-=(const TPoint& subber);`

Decrements *x* by *adder.x*, and *y* by *adder.y*. Returns **this*.

TP
Std class

Friends

operator – `friend TPoint& operator-(const TPoint& one, const TPoint& two);`

Subtracts coordinate *two* from coordinate *one*. Sets *x* to *(one.x – two.x)* and sets *y* to *(one.y – two.y)*. The arguments *one* and *two* are, of course, unchanged. Returns **this*.

operator + `friend TPoint& operator+(const TPoint& one, const TPoint& two);`

Adds coordinate *two* to coordinate *one*. Sets *x* to *(one.x | two.x)* and sets *y* to *(one.y | two.y)*. The arguments *one* and *two* are, of course, unchanged. Returns **this*.

operator == `friend int operator==(const TPoint& one, const TPoint& two);`

Returns true (nonzero) if the points *one* and *two* are identical (that is, if *one.x == two.x && one.y == two.y)*. Otherwise, false (0) is returned.

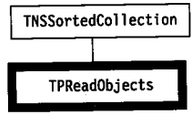
operator != `friend int operator!=(const TPoint& one, const TPoint& two);`

Returns true (nonzero) if the points *one* and *two* are different (that is, if *one.x != two.x || one.y != two.y)*. Otherwise, false (0) is returned.

Related functions

Certain other operator functions are related to **TPoint** but are not member functions; see page 232 for more information.

TPReadObjects TOBJSTRM.H



TPReadObjects (together with **TPWrittenObjects**) solves the difficult problem of spurious duplications when writing and reading objects to and from streams via pointers. This class maintains a database of all objects that have been read from the current object stream. This is used by **ipstream** when it reads a pointer from a stream to determine if other addresses for that objects exist. With this mechanism, if *ptr1* and *ptr2* point to the same streamable object and you write both pointers to a **opstream**, only one copy of the object is saved. When you come to read back from

the stream, only one copy of **ptr1* is created, and both *ptr1* and *ptr2* will still point to it.

Member functions

constructor `TPReadObjects();` **private**

This private constructor creates a non-streamable collection by calling the base **TNSSortedCollection** constructor. It is accessible only by member functions and friends.

See also: **TNSSortedCollection::TNSortedCollection**

destructor `~TPReadObjects();` **private**

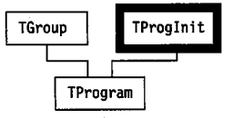
Sets the collection *limit* to 0 without destroying the collection (since the *shouldDelete* data member is set to *False*).

See also: **TNSCollection::~TNSCollection, TNSCollection::shouldDelete**

Friends

The class **ipstream** is a friend of **TPReadObjects**, so all its member functions can access the private members of **TPReadObjects**.

TProgInit APP.H



TProgInit is a virtual base class for **TProgram**. The **TProgram** constructor calls the **TProgInit** base constructor, passing to it the addresses of three initialization functions that create your status line, menu bar, and desk top. See the entries for the **TProgram** and **TApplication** constructors.

Member functions

constructor

```
TProgInit( TStatusLine *(*cStatusLine)( TRect r ),TMenuBar *(*cMenuBar)(
           TRect r ),TDeskTop *(*cDeskTop )( TRect r ) );
```

See the description under **TProgram**'s constructor.

createDeskTop `TDeskTop *(*createDeskTop)( TRect r );` **protected**

Creates the desk top with the given size.

See also: **TApplication::TApplication**

createMenuBar `TMenuBar *(*createMenuBar)( TRect r );` **protected**

Creates the menu bar with the given size.

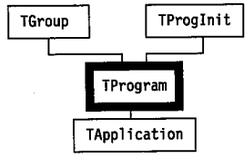
See also: **TApplication::TApplication**

createStatusLine `TStatusLine *(*createStatusLine)( TRect r );` **protected**

Creates the status line with the given size.

See also: **TApplication::TApplication**

TProgram APP.H



TProgram provides the basic template for all standard Turbo Vision applications. All such programs must be derived from **TProgram** or its immediate derived class, **TApplication**. **TApplication** differs from **TProgram** only in its default constructor and destructor. Both classes are provided for added flexibility when designing nonstandard applications. For most Turbo Vision work, your program will be derived from **TApplication**.

TProgram is derived from **TGroup** since it needs to contain (own) your **TDeskTop**, **TStatusLine**, and **TMenuBar** objects.

Data members

application `static TProgram *application;`

A pointer to the current application, set to **this** by the **TProgInit** constructor.

See also: **TProgInit** class

appPalette `static int appPalette;`

Indexes the default palette for this application as set by **initScreen**. The **TPalette** object corresponding to *appPalette* is returned by **TProgram::getPalette**.

See also: Palettes section below; **TProgram::getPalette**

deskTop `static TDeskTop *deskTop;`

A pointer to the current desk top object, set by a call to **createDeskTop** in the **TProgram** constructor. The resulting desk top is inserted into the **TProgram** group.

See also: **TProgInit::createDeskTop**, **TProgram::initDeskTop**

menuBar `static TMenuBar *menuBar;`

A pointer to the current menu bar object, set by a call to **createMenuBar** in the **TProgram** constructor. The resulting menu bar is inserted into the **TProgram** group.

See also: **TProgInit::createMenuBar**, **TProgram::initMenuBar**

statusLine `static TStatusLine *statusLine;`

A pointer to the current status line object, set by a call to **createStatusLine** in the **TProgram** constructor. The resulting status line is inserted into the **TProgram** group.

See also: **TProgInit::createStatusLine**, **TProgram::initStatusLine**

Member functions

constructor `TProgram();`

The **TProgram** constructor calls the **TProgInit** base constructor, passing to it the addresses of three init functions:

```
TProgram::TProgram() :
    TProgInit( &TProgram::initStatusLine, &TProgram::initMenuBar,
               &TProgram::initDeskTop
             ),
    ...
```

The **TProgInit** constructor creates a status line, menu bar, and desk top:

```
TProgInit::TProgInit( TStatusLine *(*cStatusLine)( TRect ), TMenuBar
                      *(*cMenuBar)( TRect ), TDeskTop *(*cDeskTop )( TRect )
```

```
                    ) :
  createStatusLine( cStatusLine ),
  createMenuBar( cMenuBar ),
  createDeskTop( cDeskTop )
  ...
```

If these calls are successful, the three objects are inserted into the TProgram group. The static member pointers *statusLine*, *menuBar*, *deskTop*, and *application* (= **this**) are set to point at these new objects. The **TGroup** constructor is also invoked to create a full screen view; the video buffer and default palettes are initialized; and the following state flags are set:

```
state = sfVisible | sfSelected | sfFocused | sfModal | sfExposed;
```

See also: **TProgInit**, **TGroup::TGroup**

destructor `virtual ~Program();`

Deletes the associated **deskTop**, **menuBar**, and **statusLine** objects, and sets the *application* static member to 0.

See also: **~TGroup**

getEvent `virtual getEvent(TEvent& event);`

The default **TView::getEvent** simply calls its owner's **getEvent**, and since a **TProgram** (or **TApplication**) object is the ultimate owner of every view, every **getEvent** call ends up in **TProgram::getEvent** (unless some view along the way has overridden **getEvent**).

TProgram::getEvent first checks if **TProgram::putEvent** has generated a pending event (in the static **TEvent** member *pending*); if so, **getEvent** returns that event. If there is no pending event, **getEvent** calls **getMouseEvent**; if that returns *evNothing*, it then calls **getKeyEvent**. If both return *evNothing*, indicating that no user input is available, **getEvent** calls **TProgram::idle** to allow "background" tasks to be performed while the application is waiting for user input. Before returning, **getEvent** passes any *evKeyDown* and *evMouseDown* events to the **statusLine** for it to map into associated *evCommand* hot key events.

See also: **TProgram::putEvent**, **getMouseEvent**, **getKeyEvent**

getPalette `virtual TPalette& getPalette() const;`

Returns the palette string given by the palette index in the *appPalette* static data member. **TProgram** supports three palettes, *apColor*, *apBlackWhite*, and *apMonochrome*. *appPalette* is initialized by **TProgram::initScreen**.

See also: **TProgram::initScreen**, **AppPalette**, *apXXXX* constants

handleEvent `virtual handleEvent(TEvent& event);`

Handles *Alt-1* through *Alt-9* keyboard events by generating an *evBroadcast* event with a *command* value of *cmSelectWindowNum* and an *infoInt* value of 1 to 9. **TWindow::handleEvent** reacts to such broadcasts by selecting the window if it has the given number.

Handles an *evCommand* event with a *command* value of *cmQuit* by calling **endModal**(*cmQuit*), which in effect terminates the application.

TProgram::handleEvent is almost always overridden to introduce handling of commands that are specific to your own application.

See also: **TGroup::handleEvent**

idle `virtual void idle();`

idle is called by **TProgram::getEvent** whenever the event queue is empty, allowing the application to perform background tasks while waiting for user input.

The default **TProgram::idle** calls *statusLine->update* to allow the status line to update itself according to the current help context. Then, if the command set has changed since the last call to **TProgram::idle**, an *evBroadcast* with a *command* value of *cmCommandSetChanged* is generated to allow views that depend on the command set to enable or disable themselves.

If you override **idle**, always make sure to call the inherited **idle**. Also, make sure that any tasks performed by your **idle** do not suspend the application for any noticeable length of time, since this would block user input and give an unresponsive feel to the application.

initDeskTop `static TDeskTop *initDeskTop(TRect);`

The address of this function is passed to the **TProgInit** constructor, which creates a **TDeskTop** object for the application and stores a pointer to it in the *deskTop* global variable. **initDeskTop** should never be called directly. **initDeskTop** is almost always overridden to instantiate a user defined **TDeskTop** instead of the default empty **TDeskTop**.

See also: **TProgram::TProgram**, **TDeskTop**

initMenuBar `static TMenuBar *initMenuBar(TRect);`

The address of this function is passed to the **TProgInit** constructor, which creates a **TMenuBar** object for the application and stores a pointer to it in the *menuBar* static member. **initMenuBar** should never be called directly. **initMenuBar** is almost always overridden to instantiate a user defined **TMenuBar** instead of the default empty **TMenuBar**.

See also: **TProgram::TProgram**, **TMenuBar**

initScreen `virtual void initScreen();`

Called by **TProgram::TProgram** and **TProgram::setScreenMode** every time the screen mode is initialized or changed. This is the member function that actually performs the updating and adjustment of screen mode-dependent variables for shadow size, markers and application palette.

See also: **TProgram::TProgram, TProgram::SetScreenMode**

initStatusLine `static TStatusLine *initStatusLine(TRect);`

The address of this function is passed to the **TProgInit** constructor, which creates a **TStatusLine** object for the application and stores a pointer to it in the *statusLine* static member. **iniTStatusLine** should never be called directly. **iniTStatusLine** is almost always overridden to instantiate a user defined **TStatusLine** instead of the default empty **TStatusLine**.

See also: **TProgram::TProgram, TStatusLine**

outOfMemory `virtual void outOfMemory();`

outOfMemory is called by **TProgram::validView** whenever it detects that *lowMemory* is *True*. **outOfMemory** should alert the user to the fact that there is not enough memory to complete an operation. For example, using the **messageBox** routine in the STDDLG header file:

```
virtual void TMyApp::outOfMemory
{
   messageBox("Not enough memory to complete operation.", 0, mfError +
              mfOKButton);
};
```

See also: **TProgram::validView**, *lowMemory*

putEvent `virtual void putEvent(TEvent& event);`

The default **TView::putEvent** simply calls its owner's **putEvent**, and since a **TProgram** (or **TApplication**) object is the ultimate owner of every view, every **putEvent** call ends up in **TProgram::putEvent** (unless some view along the way has overridden **putEvent**).

TProgram::putEvent stores a copy of the **event** structure in the **pending** global variable, and the next call to **TProgram::getEvent** will return that copy.

See also: **TProgram::getEvent, TView::putEvent**

run `virtual void run();`

Runs the **TProgram** by calling the **execute** member function (inherited from **TGroup**).

See also: **TGroup::execute**

setScreenMode `void setScreenMode(ushort mode);`

Sets the screen mode. *mode* is one of the constants *smCO80*, *smBW80*, or *smMono*, optionally with *smFont8x8* added to select 43- or 50-line mode on an EGA or VGA. **setScreenMode** hides the mouse, calls **TScreen::setVideoMode** to change the screen mode, sets up the screen buffer, initializes any screen mode-dependent variables, calls **changeBounds** with the new screen rectangle, and finally unhides the mouse.

See also: **TScreen::setVideoMode**, *smXXXX* constants

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

validView `TView *validView(TView *p);`

Checks the validity of **p*, a newly instantiated view, returning *p* if the view is valid, 0 if not. First, if *p* is 0, a value of 0 is returned. Second, if *lowMemory* is *True* upon the call to **validView**, the view given by *p* is deleted, **outOfMemory** is called, and a value of 0 is returned. Third, if the call `p->Valid(0)` returns *False*, the *p* is deleted and a value of 0 is returned. Otherwise, the view is considered valid, and *p* is returned.

validView is often used to validate a new view before inserting it in its owner group. The following statement, for example, shows a typical sequence of instantiation, validation, and insertion of a new window on the desk top (both **TProgram::validView** and **TGroup::insert** know how to ignore possible null pointers resulting from errors).

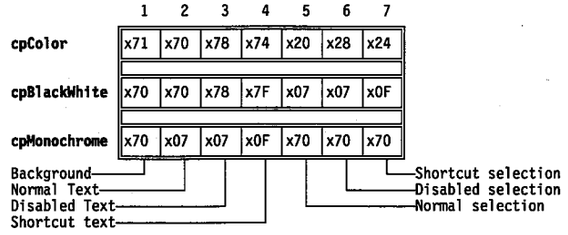
```
deskTop->insert(validView(new TMyWindow));
```

See also: *lowMemory*, **TProgram::outOfMemory**, **valid** member functions

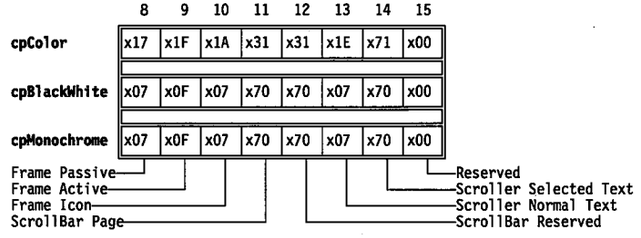
Palettes

The palette for an application object controls the final color mappings for all views in the application. All other palette mappings eventually result in the selection of an entry in the application's palette, which provides text attributes.

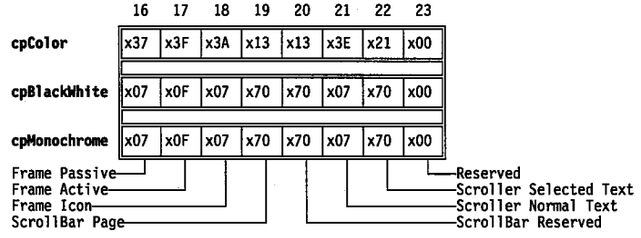
The first entry is used by **TBackground** for the background color. Entries 2 through 7 are used by both menu views and status lines.



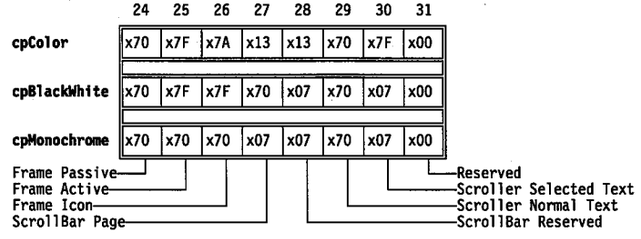
Entries 8 through 15 are used by blue windows.



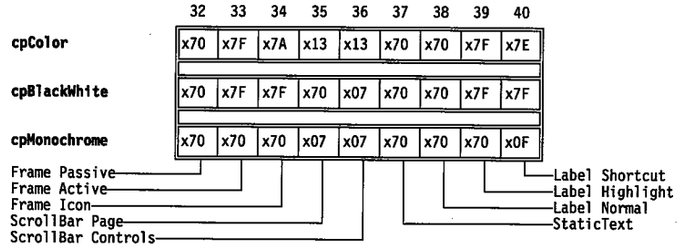
Entries 16 through 23 are used by cyan windows.

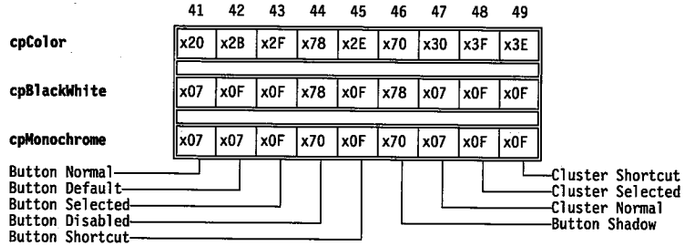


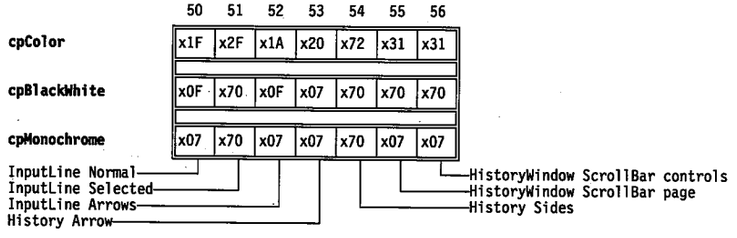
Entries 24 through 31 are used by gray windows.

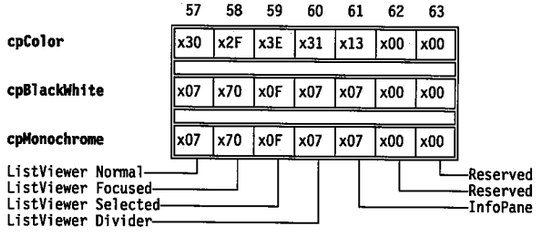


Entries 32 through 63 are used by dialog box objects. See **TDialog** for individual entries.

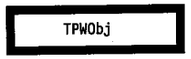








TPWObj — TOBJSTRM.H

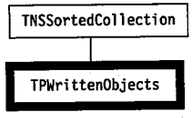


TPWObj is used internally by **TPWrittenObjects**.

Friends

The class **TPWrittenObjects** is a friend of **TPWobj**, so all its member functions can access the private members of **TPWObj**.

TPWrittenObjects — TOBJSTRM.H



TPWrittenObjects (together with **TPReadObjects**) solves the difficult problem of spurious duplications when writing and reading objects to and from streams via pointers. This class maintains a database of all objects that have been written to the current object stream. This is used by **opstream** when it writes a pointer onto a stream: it must determine if the object pointed to has already been written to the stream. With this mechanism, if *ptr1* and *ptr2* point to the same streamable object and you write both pointers to a **opstream**, only one copy of the object is saved. When you come to read back from the stream, only one copy of **ptr1* is created, and both *ptr1* and *ptr2* will still point to it.

Member functions

constructor `TPWrittenObjects();` **private**

This private constructor creates a non-streamable collection by calling the base **TNSSortedCollection** constructor. It is accessible only by the member functions and friends.

See also: **TNSSortedCollection::TNSortedCollection**

destructor `~TPWrittenObjects();` **private**

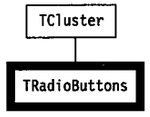
Sets the collection *limit* to 0 without destroying the collection (since the *shouldDelete* data member is set to *False*).

See also: **TNSCollection::~TNSCollection, TNSCollection::shouldDelete**

Friend

The class **opstream** is a friend of **TPWrittenObjects**, so all its member functions can access the private members of **TPWrittenObjects**.

TRadioButtons — DIALOGS.H



TRadioButtons objects are clusters of up to *maxCollectionSize* (16,380) controls with the special property that only one control button in the cluster can be selected at any moment. Selecting an unselected button will automatically deselect (restore) the previously selected button. Most of the functionality is derived from **TCluster**, including the constructors and destructor. Radio buttons are often associated with a **TLabel** object.

TRadioButtons interprets the inherited *TCluster::value* data member as the number of the "pressed" button, with the first button in the cluster being number 0.

Member functions

constructor `TRadioButtons( StreamableInit );` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws buttons as " ( ) " surrounded by a box.

mark `virtual Boolean mark(int item);`

Returns *True* if *item* is equal to *value*; that is, if the *item*'th button represents the current *value* data member (the "pressed" button).

See also: **TCluster::value, TCluster::mark**

movedTo `virtual void movedTo(int item);`

Assigns *item* to *value*.

See also: **TCluster::movedTo, TRadioButtons::mark**

press `virtual void press(int item);`

Assigns *item* to *value*. Called when the *item*'th button is pressed.

setData `virtual void setData(void *rec);`

Calls **TCluster::setData** to set *value*, then sets *sel* to *value*, since the selected item *is* the "pressed" button at startup.

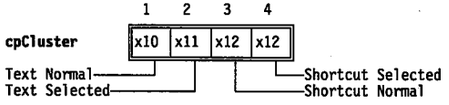
See also: **TCluster::setData**

Related functions

Certain operator functions are related to **TRadioButtons** but are not member functions; see page 232 for more information.

Palette

TRadioButtons objects use *cpCluster*, the default palette for all cluster objects, to map onto the sixteenth through eighteenth entries in the standard dialog palette.



TRect OBJECTS.H

TRect objects represent two **TPoint** objects (the top-left and bottom-right corners of the rectangle) together with several inline member functions for manipulating rectangles. The operators **==** and **!=** are overloaded to provide the comparison of two rectangles in a natural way. **TPoint** has data members *x* and *y*, the point's screen coordinates.

Data members

a `TPoint a;`

a is the point defining the top-left corner of a rectangle on the screen.

See also: **TPoint**

b `TPoint b;`

b is the point defining the bottom-right corner of a rectangle on the screen.

See also: **TPoint**

Member functions

constructor
```
TRect(int ax, int ay, int bx, int by);
TRect(TPoint topleft, TPoint bottomright);
```

Creates a **TRect** object and initializes it with *a.x = ax*; *a.y = ay*; and so on. Alternatively, you can construct a rectangle by supplying two **TPoint** arguments; in which case, *a* is set to *topleft* and *b* is set to *bottomright*.

constructor `TRect();`

Allows the creation of an uninitialized **TRect** object using **new** without arguments.

contains `Boolean contains(const TPoint& p) const;`

Returns *True* if the calling rectangle (including its boundary) contains the point *p*.

grow `void grow(int aDX, int aDY);`

Changes the size of the calling rectangle by subtracting *aDX* from *a.x*, adding *aDX* to *b.x*, subtracting *aDY* from *a.y*, and adding *aDY* to *b.y*.

intersect `void intersect(const TRect& r);`

Changes the location and size of the calling rectangle to the region defined by the intersection of the current rectangle with *r*.

isEmpty `Boolean isEmpty();`

Returns *True* if the rectangle contains no character-sized interior; otherwise, returns *False*. Empty means that *(a.x >= b.x || a.y >= b.y)*.

move `void move(int aDX, int aDY);`

Moves the calling rectangle by adding *aDX* to *a.x* and *b.x* and adding *aDY* to *a.y* and *b.y*.

operator == `Boolean operator == (const TRect& r) const;`

Returns *True* if *r* is the same as the calling rectangle; otherwise, returns *False*.

operator != `Boolean operator != (const TRect& r) const;`

Returns *True* if *r* is not the same as the calling rectangle; otherwise, returns *False*.

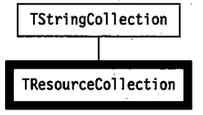
Union `void Union(const TRect& r);`

Changes the calling rectangle to be the union of itself and the rectangle *r*; that is, to the smallest rectangle containing both the object and *r*.

Related functions

Certain operator functions are related to **TRect** but are not member functions; see page 232 for more information.

TResourceCollection RESOURCE.H



TResourceCollection is a derivative of **TStringCollection**, which makes it a sorted, streamable collection. It is used with **TResourceFile** to implement collections of resources. A resource file is a stream that is indexed by key strings. Each resource item points to an object of type **TResourceItem** defined as follows:

```
struct TResourceItem
{
   long pos;
   long size;
   char *key;
};
```

The fields provide the stream position and item size for the resource item matching the string *key*. The overriding member functions of **TResourceCollection** are mainly concerned with handling the extra string element in its items. **TResourceCollection** is used internally by **TResourceFile** objects to maintain a resource file's index.

Data members

name `static const char * const name;`

Class name used by the stream manager.

Member functions

constructor `TResourceCollection( short aLimit, short aDelta );`

Creates a resource collection with initial size *aLimit* and the ability to resize by *aDelta*.

constructor `TResouceCollection( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TStringCollection::TStringCollection**

build

```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

freeItem

```
virtual void freeItem( void *item );
```

Frees the given item from the collection by deleting both the key and the item.

keyOf

```
virtual void *keyOf( void *item);
```

Returns the key of the given item. The implementation is

```
void* TResourceCollection::keyOf( void *item )
{
   return ((TResourceItem *)item)->key;
}
```

read

```
virtual void *read( ipstream& is);
```

Reads from the input stream *is* to *ts*.

See also: **TStreamableClass, TStreamable, ipstream**

readItem

```
void *TResourceCollection::readItem( ipstream& is );
```

Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to read the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, ipstream**

write

```
virtual void write( opstream& os );
```

Writes *ts* to the *os* stream.

See also: **TStreamableClass, TStreamable, opstream**

writeItem

```
void TResourceCollection::writeItem( void *obj, opstream& os );
```

Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to

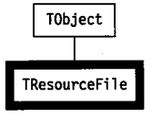
write the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TResourceCollection** but are not member functions; see page 232 for more information.

TResourceFile — RESOURCE.H



TResourceFile implements a stream (of type **fpstream**) that can be indexed by string keys. When objects are stored in a resource file using **TResourceFile::put**, a string key, which identifies the object, is also supplied. The objects can later be retrieved by specifying the string key in a call to **TResourceFile::get**.

To provide fast and efficient access to the objects stored in a resource file, **TResourceFile** objects store the keys in a sorted string collection (using the **TResourceCollection** class) along with the position and size of the resource data in the resource file. The data member *index* points to the associated **TResourceCollection** object, known appropriately as the index to the resource file.

As with all stream I/O, the classes of all objects written to and read from resource files must be streamable and must be registered (that is, notified to the stream manager—see Chapter 8, "Streamable objects").

TR
Std class

Data members

basePos

```
long basePos;
```

The base position of the stream (ignoring header information).

See also: **fpstream**

index

```
TResourceCollection *index;
```

A pointer to the associated **TResourceCollection** object.

See also: **TResourceCollection**

indexPos

```
long indexPos;
```

The current position of the stream relative to the base position.

modified

```
Boolean modified;
```

Set *True* if the resource file has been modified since the last **flush** call; otherwise *False*.

See also: **TResourceFile::flush**, **TResourceFile::put**

stream

```
fpstream *stream;
```

Pointer to the file stream associated with this resource file.

See also: **fpstream**

Member functions

constructor

```
TResourceFile( fpstream *aStream );
```

Initializes a resource file using the stream given by *aStream* and sets the *modified* data member to *False*. The stream must have already been initialized. For example,

```
TResourceFile *resFile = new TResourceFile(new fpstream("MYAPP.RES", ios::in |
                                                        ios::out));
```

During initialization, the **TResourceFile** constructor looks for a resource file header at the current position of the stream. If a header is not found, the constructor assumes that a new resource file is being created together with a new resource collection. You will not normally be concerned with the header internals, but advanced programmers may wish to know the following. The resource file header is defined by the following structure:

```
struct THeader
{
   ushort signature;
   union
   {
      Count_type count;
      Info_type info;
   };
};
```


where **Count_type** is

```
struct Count_type
{
   ushort lastCount;
   ushort pageCount;
};
```

and **Info_type** is

```
struct Info_type
{
   ushort infoType;
   long infoSize;
};
```

signature contains either 0x5a4d or 0x4246. If the signature is 0x5a4d, the **Count_type** field of the union is used; if the signature is 0x4246, the **Info_type** field is used. If the constructor sees an .EXE file signature at the current position of the stream, it seeks the stream to the end of the .EXE file image, and then looks for a resource file header there. Likewise, the constructor will skip over an overlay file that was appended to the .EXE file. This means that you can append both your overlay file and your resource file (in any order) to the end of your application's .EXE file. In all cases, *basePos* and *indexPos* are set to the correct values allowing for any headers.

See also: **~TResourceFile**

destructor

```
~TResourceFile();
```

Flushes the resource file, using **TResourceFile::flush**, and then deletes *index* and *stream*.

See also: **TResourceFile** constructor, **TResourceFile::flush**

count

```
short count();
```

Calls **index->getCount** to return the number of resource items stored in the associated **TResourceCollection**.

See also: **TResourceFile::getCount**

flush

```
void flush();
```

If the resource file has not been modified since the last **flush** (that is, if *modified* is *False*), **flush** does nothing. Otherwise, **flush** stores the updated index at the end of the stream and updates the resource header at the beginning of the stream. It then calls **stream->flush** and resets *modified* to *False*.

See also: **~TResourceFile, TResourceFile::modified, opstream::flush**

get `void *get( const char *key );`

Searches for the given *key* in the associated resource file collection (given by the pointer *index*). Returns 0 if the key is not found. Otherwise, **get seekg**'s the stream to the position given by the *pos* field in the **TResourceItem** object located at *key*. The object at (*basePos* + *pos*) is created and a pointer to it is returned. For example,

```
deskTop->insert(validView(resFile.get("editorWindow")));
```

See also: **TResourceCollection::at, TResourceFile::put, TApplication::validView, ipstream::seekg**

keyAt `const char *keyAt(short i);`

Uses **index->at**(*i*) to return the string key of the *i*'th resource in this resource file. The index of the first resource is zero and the index of the last resource is **TResourceFile::count** minus one. Using *count* and **keyAt**, you can iterate over all resources in a resource file.

See also: **TResourceFile::count, TResourceCollection::at**

put `void put(TStreamable *item, const char *key);`

Adds the streamable object given by *item* to the resource file with the key string given by *key* and sets *modified* to *True*. If the index already contains the *key*, then the new object replaces the old object; otherwise, the new object is appended in the correct indexed position of the resource file.

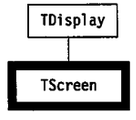
See also: **TResourceFile::get, TNSSortedCollection::search**

remove `void remove(const char *key);`

If the resource indexed by *key* is not found, **remove** does nothing. Otherwise it calls **index->free** to remove the resource.

See also: **TNSSortedCollection::search, TNSCollection::free**

TScreen — SYSTEM.H



TScreen provides low-level video attributes and functions. This class, and the other systems classes in SYSTEM.H, are listed briefly for guidance

only: they are used internally by Turbo Vision and you would not need to use them explicitly for normal applications. **TView** is a friend class of **TDisplay**.

Data members

checkSnow `static Boolean near checkSnow;`

True if the "check snow" feature is enabled; otherwise *False*.

cursorLines `static ushort near cursorLines;`

Holds the current cursor type, set by **setCrtData** with a call to **getCursorType**.

See also: **TDisplay::getCursorType, TScreen::setCrtData**

hiResScreen `static Boolean near hiResScreen;`

True if *screenHeight* is greater than 25; otherwise *False*.

See also: *TScreen::screenHeight*

screenBuffer `static uchar far * near screenBuffer;`

Points to the appropriate video buffer for the particular video configuration detected and its current mode.

screenHeight `static uchar near screenHeight;`

Holds the current screen height, set by **setCrtData** with a call to **getRows**.

See also: **TDisplay::getRows, TScreen::setCrtData**

screenMode `static ushort near screenMode;`

The current video mode.

See also: **TDisplay::getVideoMode**

screenWidth `static uchar near screenWidth;`

Holds the current screen width, set by **setCrtData** via a call to **getCols**.

See also: **TDisplay::getCols, TScreen::setCrtData**

startupCursor `static ushort near startupCursor;`

Holds the initial cursor type set by **initScreen** by way of the **TApplication/TProgram** constructors.

See also: **TProgram::initScreen, TDisplay::getCursorType**

TS
Std class

startupMode `static ushort near startupMode;`

Holds the initial video mode set by **initScreen** by way of the **TApplication/TProgram** constructors.

See also: **TProgram::initScreen**

Member functions

constructor `TScreen();`

Creates a **TScreen** object and calls **resume**. This initializes *startupMode* via **getCrtMode**; *startupCursor* via **getCursorType**; then sets the remaining data members by calling **setCrtData**.

See also: **TDisplay::getCrtMode, TDisplay::getCursorType, TScreen::setCrtData, TScreen::resume**

destructor `~TScreen();`

Calls **suspend**. This restores the screen mode to the startup mode, clears the screen, then restores the cursor to the startup cursor.

See also: *TDisplay::startupMode*, **TDisplay::setCrtMode, TScreen::clearScreen, TDisplay::setCursorType, TScreen::suspend**

clearScreen `static void clearScreen();`

Calls **TDisplay::clearScreen** with the current *screenWidth* and *screenHeight* as arguments.

See also: **TDisplay::clearScreen**

fixCrtMode `static ushort fixCrtMode( ushort vmode );`

If the lower byte of the given *vmode* is not *smMono*, *smCO80*, or *smBW80*, **fixCrtMode** returns *smCO80*. Used by **TScreen::setVideoMode** to handle nonstandard modes.

See also: *TDisplay::videoModes*

resume `static void resume();`

Called by the **TScreen** constructor to initialize *startupMode* via **getCrtMode**; *startupCursor* via **getCursorType**; then sets the remaining data members by calling **setCrtData**.

See also: **TDisplay::getCrtMode, TDisplay::getCursorType, TScreen::setCrtData, TScreen::TScreen**

setCrtData `static void setCrtData();`

Sets the *screenMode*, *screenWidth*, and *screenHeight* data members by calling **getCrtMode, getCols**, and **getRows**. The *hiResScreen* data member is set *True* or *False* depending on the value of *screenHeight*. Finally, depending on the current *screenMode*, the *screenBuffer* and *checkSnow* members are set.

See also: **TDisplay::getCrtMode, TDisplay::getCols**, and **TDisplay::getRows**.

setsVideoMode `static void setVideoMode( ushort vmode );`

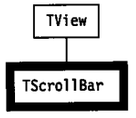
Sets the video mode to *vmode*, then adjusts the other **TScreen** data members as appropriate.

suspend `static void suspend();`

Called by the **TScreen** destructor. **suspend** restores the screen mode to the startup mode, clears the screen, then restores the cursor to the startup cursor.

See also: **TScreen::~TScreen**

TScrollBar VIEWS.H



Data members

arStep `short arStep;`

arStep is the amount added or subtracted to the scroll bar's *value* data member when an arrow area is clicked (*sbLeftArrow*, *sbRightArrow*, *sbUpArrow*, or *sbDownArrow*) or the equivalent keystroke made. The **TScrollBar** constructor sets *arStep* to 1 by default.

See also: **TScrollBar::setStep, TScrollBar::setParam, TScrollBar::scrollStep**

chars `TScrollChars chars;`

TScrollChars is defined as

```
typedef char TScrollChars[5];
```

chars is set with the five basic character patterns used to draw the scroll bar parts.

maxVal `short maxVal;`

maxVal represents the maximum value for the *value* data member. The **TScrollBar** constructor sets *maxVal* to zero by default.

See also: **TScrollBar::setRange**, **TScrollBar::setParams**

minVal `short minVal;`

minVal represents the minimum value for the *value* data member. The **TScrollBar** constructor sets *minVal* to zero by default.

See also: **TScrollBar::setRange**, **TScrollBar::setParams**

pgStep `short pgStep;`

pgStep is the amount added or subtracted to the scroll bar's *value* data member when a mouse click event occurs in any of the page areas (*sbPageLeft*, *sbPageRight*, *sbPageUp*, or *sbPageDown*) or an equivalent keystroke is detected (*Ctrl-←*, *Ctrl-→*, *PgUp*, or *PgDn*). The **TScrollBar** constructor sets *pgStep* to 1 by default. You can change *pgStep* using **TScrollBar::setStep**, **TScrollBar::setParams**, or **TScroller::setLimit**.

See also: **TScrollBar::setStep**, **TScrollBar::setParams**, **TScroller::setLimit**, **TScrollBar::scrollStep**

value `short value;`

The *value* data member represents the current position of the scroll bar indicator. This specially colored marker moves along the scroll bar strip to indicate the relative position (horizontally or vertically, depending on the scroll bar orientation) of the scrollable text being viewed relative to the total text available for scrolling.

Many events can directly or indirectly change *value*, such as clicking on the designated scroll bar parts, resizing the window, or changing the text in the scroller. Similarly, changes in *value* may need to trigger other events. The **TScrollBar** constructor sets *value* to zero by default.

See also: **TScrollBar::setvalue**, **TScrollBar::setParams**, **TScrollBar::scrollDraw**, **TScroller::handlevent**, **TScrollBar::TScrollBar**

Member functions

constructor `TScrollBar(const TRect& bounds);`

Creates and initializes a scroll bar with the given *bounds* by calling the **TView** constructor. *value*, *maxVal*, and *minVal* are set to zero. *pgStep* and *arStep* are set to 1. The shapes of the scroll bar parts are set to the defaults in **TScrollChars**.

If *bounds* produces *size.x* = 1, you get a vertical scroll bar; otherwise, you get a horizontal scroll bar. Vertical scroll bars have the *growMode* data member set to *gfGrowLoX* | *gfGrowHiX* | *gfGrowHiY*; horizontal scroll bars have the *growMode* data member set to *gfGrowLoY* | *gfGrowHiX* | *gfGrowHiY*.

constructor `TScrollBar( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

draw `void draw();`

Draws the scroll bar depending on the current *bounds*, *value*, and palette.

See also: **TScrollBar::scrollDraw**, **TScrollBar::value**

getPalette `virtual const char * getPalette() const;`

Returns *cpScrollBar*, the default scroll bar palette string, "\x04\x05\x05\".

handleEvent `virtual void handleEvent(TEvent& event);`

Handles scroll bar events by calling **TView::handleEvent**, then analyzing *event.what*. Mouse events are broadcast to the scroll bar's owner (see **message** function), which must handle the implications of the scroll bar changes (for example, by scrolling text). **TScrollBar::handleEvent** also determines which scroll bar part has received a mouse click (or equivalent keystroke). The *value* data member is adjusted according to the current *arStep* or *pgStep* values, and the scroll bar indicator is redrawn.

See also: **TView::handleEvent**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream** classes

scrollDraw `virtual void scrollDraw();`

scrollDraw is called whenever the *value* data member changes. This virtual member function defaults by sending a *cmScrollBarChanged* message to the scroll bar's owner:

```
message(owner, evBroadcast, cmScrollBarChanged, this);
```

See also: **TScrollBar::value**, **message**

scrollStep `virtual short scrollStep(short part);`

By default, **scrollStep** returns a positive or negative step value, depending on the scroll bar part given by *part*, and the current values of *arStep* and *pgStep*. The *part* argument should be one of the *sbXXXX* scroll bar part constants described in Chapter 16.

See also: **TScrollBar::setStep**, **TScrollBar::setParams**

setParams
```
void setParams(short aValue, short aMin, short aMax, short aPgStep, short
               aArStep);
```

setParams sets the *value*, *minVal*, *maxVal*, *pgStep*, and *arStep* data members with the given argument values. Some adjustments are made if your arguments conflict. For example, *minVal* cannot be set higher than *maxVal*, so if *aMax* < *aMin*, *maxVal* is set to *aMin*. *value* must lie in the closed range *[minVal,maxVal]*, so if *aValue* < *aMin*, *value* is set to *aMin*; and if *aValue* > *aMax*, *value* is set to *aMax*. The scroll bar is redrawn by calling **drawView**. If *value* is changed, **scrollDraw** is also called.

See also: **TView::drawView**, **TScrollBar::scrollDraw**, **TScrollBar::setRange**, **TScrollBar::setValue**

setRange `void setRange(short aMin, short aMax);`

setRange sets the legal range for the *value* data member by setting *minVal* and *maxVal* to the given arguments *aMin* and *aMax*. **setRange** calls **setParams**, so **drawView** and **scrollDraw** will be called if the changes require the scroll bar to be redrawn.

See also: **TScrollBar::setParams**

setStep `void setStep(short aPgStep, short aArstep);`

setStep sets the data members *pgStep* and *arStep* to the given arguments *aPgStep* and *aArStep*. This member function calls **setParams** with the other arguments set to their current values.

See also: **TScrollBar::setParams**, **TScrollBar::scrollStep**

setValue `void setValue(short aValue);`

setValue sets the *value* data member to *aValue* by calling **setParams** with the other arguments set to their current values. **drawView** and **scrollDraw** will be called if this call changes *value*.

See also: **TScrollBar::setParams**, **TView::drawView**, **TScrollBar::scrollDraw**, **TScroller::scrollTo**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

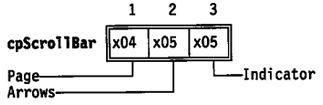
See also: **TStreamableClass**, **TStreamable**, **opstream**

Related functions

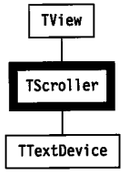
Certain operator functions are related to **TScrollBar** but are not member functions; see page 232 for more information.

Palette

Scroll bar objects use the default palette, *cpScrollBar*, to map onto the fourth and fifth entries in the standard application palette.



TScroller VIEWS.H



Data members

delta `TPoint delta;`

delta holds the *x* (horizontal) and *y* (vertical) components of the scroller's position relative to the virtual view being scrolled. Automatic scrolling is achieved by changing either or both of these components in response, for example, to scroll bar events that change the *value* data member(s). Conversely, manual scrolling changes *delta*, triggers changes in the scroll bar *value* data members, and leads to updating of the scroll bar indicators.

See also: **TScroller::scrollDraw**, **TScroller::scrollTo**

drawFlag `Boolean drawFlag;`

Set *True* if the scroller has to be redrawn.

See also: **TView::drawView**, **TScroller::drawLock**, **TScroller::checkDraw**

drawLock `uchar drawlock;`

A semaphore used to control the redrawing of scrollers.

See also: **TView::drawView**, **TScroller::drawFlag**, **TScroller::checkDraw**

hScrollBar `TScrollBar *hScrollBar;`

hScrollBar points to the horizontal scroll bar object associated with the scroller. If there is no such scroll bar, *hScrollBar* is 0.

limit `TPoint limit;`

limit.x and *limit.y* are the maximum allowed values for *delta.x* and *delta.y*

See also: **TScroller::delta**

vScrollBar `TScrollBar *vScrollBar;`

vScrollBar points to the vertical scroll bar object associated with the scroller. If there is no such scroll bar, *vScrollBar* is 0.

Member functions

constructor

```
TScroller(const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar
        *aVScrollBar);
```


Creates and initializes a **TScroller** object with the given size and scroll bars. Calls **TView** constructor to set the view's size. *option s* is set to *ofSelectable* and *eventMask* is set to *evBroadcast*. *aHScrollBar* should be 0 if you do not want a horizontal scroll bar; similarly, *aVScrollBar* should be 0 if you do not want a vertical scroll bar.

constructor `TScroller( StreamableInit streamableInit );` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**, **TView::options**, **TView::eventMask**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

changeBounds `virtual void changeBounds(const TRect& bounds)`

Changes the scroller's size by calling **setbounds**. If necessary, the scroller and scroll bars are then redrawn by calling **drawView** and **setLimit**.

See also: **TView::setbounds**, **TView::drawView**, **TScroller::setLimit**

checkDraw `void checkDraw();`

If *drawLock* is zero and *drawFlag* is *True*, *drawFlag* is set *False* and **drawView** is called. If *drawLock* is non-zero or *drawFlag* is *False*, **checkDraw** does nothing. **scrollTo** and **setLimit** each call **checkDraw** so that **drawView** is only called if needed.

getPalette `virtual TPalette& getPalette() const;`

Returns *cpScroller*, the default scroller palette string, "\x06\x07".

handleEvent `virtual void handleEvent(TEvent& event);`

Handles most events by calling **TView::handleEvent**. Broadcast events such as *cmScrollBarChanged* from either *hScrollBar* or *vScrollBar* result in a call to **TScroller::scrollDraw**.

See also: **TView::handleEvent**, **TScroller::scrollDraw**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **TStreamable**, **ipstream**

scrollDraw `virtual void scrollDraw();`

Checks to see if *delta* matches the current positions of the scroll bars. If not, *delta* is set to the correct value and **drawView** is called to redraw the scroller.

See also: **TView::drawView, TScroller::delta, TScroller::hScrollBar, TScroller::vScrollBar**

scrollTo `void scrollTo(short x, short y);`

Sets the scroll bars to (*x,y*) by calling *hScrollBar->setValue(x)* and *vScrollBar->setValue(y)*, and redraws the view by calling **drawView**, if necessary.

See also: **TView::drawView, TScroller::setValue, TScroller::checkDraw**

setLimit `void setLimit(short x, short y);`

Sets the *limit* data member and redraws the scrollbars and scroller if necessary.

See also: **TScroller::limit, TScroller::hScrollBar, TScroller::vScrollBar, TScrollBar::setParams, TScroller::checkDraw**

setState `virtual void setState(ushort aState, Boolean enable);`

This member function is called whenever the scroller's state changes. Calls **TView::setState** to set or clear the state flags in *aState*. If the new state is *sfSelected* and *sfActive*, **setState** displays the scroll bars; otherwise, they are hidden.

See also: **TView::setState**

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

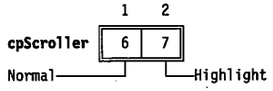
See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TScroller** but are not member functions; see page 232 for more information.

Palette

Scroller objects use the default palette, *cpScroller*, to map onto the sixth and seventh entries in the standard application palette.



TSItem — DIALOGS.H

TSItem is a simple, non-view class providing a singly-linked list of character strings. This class is useful where the full flexibility of string collections are not needed (see **TCluster** for example).

Data members

next `TSItem *next;`

Pointer to the next **TSItem** object in the linked list.

value `const char *value;`

The string for this **TSItem** object.

Member functions

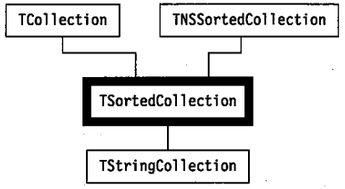
constructor `TSItem(const char *aValue, TSItem *aNext);`

Creates a **TSItem** object with the given values.

destructor `~TSItem();`

Destroys the **TSItem** object by calling **delete** *value*.

TSortedCollection OBJECTS.H



The abstract class **TSortedCollection** is a specialized derivative of both **TCollection** and **TNSSortedCollection** implementing streamable collections sorted by a key (with or without duplicates). No instances of **TSortedCollection** are allowed. It exists solely as a base for other standard or user-defined derived classes.

Sorting is implied by the pure virtual (and private) member function **compare**, which you must override in your derived classes to provide your own definition of element ordering. As new items are added they are automatically inserted in the order given by **compare**. Items can be located using the **search** function inherited from **TNSSortedCollection**. The virtual **indexOf** function (also inherited from **TNSSortedCollection**), which returns a pointer for **compare**, can also be overridden if **compare** needs additional information.

For streamable sorted collections, you must use this class. Apart from streamability, the two classes **TSortedCollection** and **TNSSortedCollection** offer the same functionality.

Member functions

constructor `TSortedCollection(ccIndex aLimit, ccIndex aDelta);`

Invokes the **TCollection** constructor to set *count*, *items*, and *limit* to zero; calls **setLimit**(*aLimit*) to set the collection limit to *aLimit*, then sets *delta* to *aDelta*. Note that *ccIndex* is a typedef'd **int**. *duplicates* is set to *False*. If you want to allow duplicate keys, you must set *duplicates True*.

constructor `TTSortedCollection( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TCollection::TCollection, TCollection** data members

compare `virtual int compare(void *key1, void *key2) = 0;` **private**

compare is a pure virtual function that must be overridden in all derived classes (or redefined as pure virtual).

See also: **TNSSortedCollection::compare**

read `void read( ipstream& is );` **protected**

Reads a sorted collection from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

readItem `void *TSortedCollection::readItem( ipstream& is );`

Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to read the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, ipstream**

write `void write( opstream& os );` **protected**

Writes the associated **TSortedCollection** object to the output stream *os*.

See also: **TStreamableClass, TStreamable, opstream** classes

writeItem `void TSortedCollection::writeItem( void *obj, opstream& os );`

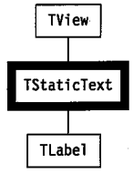
Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to write the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TSortedCollection** but are not member functions; see page 232 for more information.

TStaticText DIALOGS.H



TStaticText objects represent the simplest possible views: they contain fixed text and they ignore all events passed to them. They are generally used as messages or passive labels. Descendants of **TStaticText**, such as **TLabel** objects, usually perform more active roles.

Data member

text `const char *text;` **protected**

A pointer to the (constant) text string to be displayed in the view.

Member functions

constructor `TStaticText(const TRect& bounds, const char *aText);`

Creates a **TStaticText** object of the given size by calling **TView**(*bounds*), then sets *text* to **newStr**(*aText*).

constructor `TStaticText( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView, newStr**

destructor `~StaticText();`

Disposes of the *text* string, then calls **~View** to destroy the object.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws the text string inside the view, word wrapped if necessary. A `'\n'` in the text indicates the beginning of a new line. A line of text is centered in the view if the string begins with 0x03 (*Ctrl-C*).

getPalette `virtual TPalette& getPalette() const;`

Returns the default palette string, *cpStaticText*, "\x06".

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

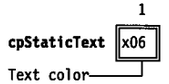
See also: **TStreamableClass, TStreamable, opstream**

Related functions

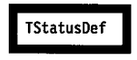
Certain operator functions are related to **TStaticText** but are not member functions; see page 232 for more information.

Palette

Static text objects use the default palette, *cpStaticText*, to map onto the sixth entry in the standard dialog palette.



TStatusDef MENUS.H



A **TStatusDef** object is not a view but represents a status line definition used by a **TStatusLine** view to display context-sensitive status lines. The *next* data member points to the next **TStatusDef** object in a list of status lines (or 0 if this is the last such). *min* and *max* define the range of help

contexts that correspond to the status line. *items* points to a list of status items that make up the status line.

A **TStatusLine** object has a pointer called *defs* to a linked list of **TStatusDef** objects. **TStatusLine** always displays the first status item for which the current help context value is within *min* and *max*. **TStatusLine::update** can be called from **TProgram::idle** to regularly update the status line view.

Data members

items `TStatusItem *items;`

items points to a list of status items that make up the status line. A value of 0 indicates that there are no status items.

See also: **TStatusLine** class, **TStatusItem** class

min, max `ushort min, max;`

The minimum and maximum help context values for which this status definition is associated. **TStatusLine** always displays the first status item for which the current help context value is within *min* and *max*.

See also: **TStatusLine::draw**

next `TStatusDef *next;`

A nonzero *next* points to the next **TStatusDef** object in a list of status definitions. A 0 value indicates that this **TStatusDef** object is the last such in the list.

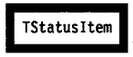
Member functions

constructor

```
TStatusDef(ushort aMin, ushort aMax, TStatusItem *someItems, TStatusDef
        *aNext);
```

Creates a **TStatusDef** object with the given values.

TStatusItem MENUS.H



A **TStatusItem** object is not a view but represents a component (status item) of a linked list associated with a **TStatusLine** view. The latter can display a status line and handle status line events such as hot keys and clicking on items. Status items can be visible or invisible. In the latter case, the item serves only to define a hot key. *next* points to the next **TStatusItem** in the list, or is 0 if this is the last item. *text* is a character string holding an item legend (such as "~Alt-X~ Exit"). If *text* is 0, the status item is invisible. *keyCode* contains the scan code of the hot key associated with the item, or zero if there is no such hot key. *command* holds the command event to be generated when the scan item is selected via hot key or mouse click.

TStatusItem serves two purposes: it controls the visual appearance of the status line, and it defines hot keys by mapping key codes to commands. **TProgram::getEvent** calls **TStatusLine::handleEvent** for all *evKeyDown* events and the items in the current status line are scanned for matching key codes. If a match is found, the *evKeyDown* event is converted to the *evCommand* event given by the *command* data member in the matching status item object.

Data members

command `ushort command;`

The value of the command associated with this status item.

keyCode `ushort keyCode;`

This is the scan code for the associated hot key.

next `TStatusItem *next;`

A nonzero *next* points to the next **TStatusItem** object in the linked list associated with a status line. A 0 value indicates that this is the last item in the list.

text `const char *text;`

The text string to be displayed for this status item. If 0, no legend will display, meaning that the status item is intended only to define a hot key using the *keyCode* member.

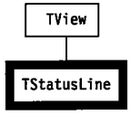
See also: **TStatusLine** class

Member functions

constructor `TStatusItem(const char *aText, ushort key, ushort cmd, TStatusItem *aNext = 0);`

Creates a **TStatusItem** object with the given values.

TStatusLine MENUS.H



The **TStatusLine** object is a specialized view, usually displayed at the bottom of the screen. Typical status line displays are lists of available hot keys, displays of available memory, time of day, current edit modes, and hints for users. The items to be displayed are set up in a linked list using **initStatusLine** called by **TApplication**, and the one displayed depends on the help context of the currently focused view. Like the menu bar and desk top, the status line is normally owned by a **TApplication** group.

Status line items are **TStatusItem** objects which contain data members for a text string to be displayed on the status line, a key code to bind a hot key (typically a function key or an *Alt*-key combination), and a command to be generated if the displayed text is clicked on with the mouse or the hot key is pressed.

Each status line object contains a linked list of status line *defs* (objects of class **TStatusDef**), which define a range of help contexts and a list of status items to be displayed when the current help context is in that range. In addition, *hints* or predefined strings can be displayed according to the current help context.

Data members

defs `TStatusDef *defs;`

A pointer to the current linked list of **TStatusDef** objects. The list to use is determined by the current help context.

See also: **TStatusDef, TStatusLine::update, TStatusLine::hint**

items `TStatusItem *items;`

A pointer to the current linked list of **TStatusItem** records.

See also: **TStatusItem**

Member functions

constructor `TStatusLine(const TRect& bounds, TStatusDef& aDefs);`

Creates a **TStatusLine** object with the given *bounds* by calling **TView**(*bounds*). The *ofPreProcess* bit in *options* is set, *eventMask* is set to include *evBroadcast*, and *GrowMode* is set to *gfGrowLoY* | *gfGrowHiX* | *gfGrowHiY*. The *defs* data member is set to *aDefs*. If *aDefs* is 0, *items* is set to 0; otherwise, *items* is set to *aDefs->items*.

constructor `TStatusLine( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TView::TView**

destructor `~TStatusLine();`

Disposes of all the *items* and *defs* in the **TStatusLine** object, then calls **~View**.

See also: **~TView**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws the status line by writing the *text* string for each status item that has one, then any **hint**s defined for the current help context, following a divider bar. **draw** uses the appropriate palettes, *cpNormal*, *cpSelect*, *cpNormDisabled*, or *cpSelDisabled*, depending on each item's status.

See also: **TStatusLine::hint**

getPalette

```
virtual const char * getPalette() const;
```

Returns the default palette string, *cpStatusLine*, "\x02\x03\x04\x05\x06\x07".

handleEvent

```
virtual void handleEvent(TEvent& event);
```

Handles events sent to the status line by calling **TView::handleEvent**, then checking for three kinds of special events. Mouse clicks that fall within the rectangle occupied by any status item generate a command event, with *event.what* set to the *command* in that status item. Key events are checked against the *keyCode* data member in each item; a match causes a command event with that item's *command*. Broadcast events with the command *cmCommandSetChanged* cause the status line to redraw itself to reflect any hot keys that might have been enabled or disabled.

See also: **TView::handleEvent, TStatusLine::draw**

hint

```
virtual const char* hint(ushort aHelpCtx);
```

By default, **hint** returns a 0 string. You must override it to provide a context-sensitive hint string for the *aHelpCtx* argument. A nonzero string will be drawn on the status line after a divider bar.

See also: **TStatusLine::draw**

read

```
virtual void *read( ipstream& is);
```

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream** classes

update

```
void update();
```

Updates the status line by selecting the correct items from the lists in *defs*, depending on the current help context, and then calls **drawView** to redraw the status line if the items have changed.

See also: **TStatusLine::defs**

write

```
virtual void write( opstream& os);
```

Writes to the output stream *os*.

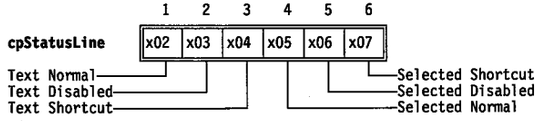
See also: **TStreamableClass, TStreamable, opstream** classes

Related functions

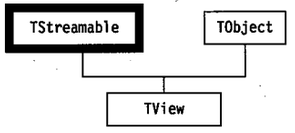
Certain operator functions are related to **TStatusLine** but are not member functions; see page 232 for more information.

Palette

Status lines use the default palette *cpStatusLine* to map onto the second through seventh entries in the standard application palette.



TStreamable TOBJSTRM.H



TView has two base classes, **TObject** and the abstract class **TStreamable**. All the viewable classes, derived ultimately from **TView**, therefore also inherit from **TStreamable**. Several non-view classes, such as **TCollection**, **TStrListMaker**, **TStringList**, and **T*xxx*Init**, also have **TStreamable** as a base class. Such classes are known as *streamable*, meaning that their objects can be written to and read from streams using the Turbo Vision stream manager.

If you want to develop your own streamable classes, make sure that **TStreamable** is somewhere in their ancestry. Using an existing streamable class as a base class, of course, is an obvious way of achieving this.

Since **TStreamable** is an abstract class, no objects of this class can be instantiated. Before a streamable class can be used with streams, the class must override the three pure virtual functions **streamableName**, **read**, and **write**.

TS
Std class

Member functions

read `virtual void *read( ipstream& is) = 0;` **protected**

This pure virtual function must be overridden (or redeclared as pure virtual) in every derived class. The overriding **read** function for each streamable class must read the necessary data members from the ipstream object *is*. **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: **ipstream, TStreamableClass**

streamableName `virtual const char *streamableName() const = 0;` **private**

TStreamable has no constructor. This function must be overridden (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 streams by the stream manager. For example, **TView** overrides the function as follows:

```
virtual const char *streamableName() const { return TViewName; }
```

TView::TViewName is a static character array holding the name "TView". The name returned must be a unique, 0-terminated string, so the safest strategy is to use the name of the streamable class.

See also: **TStreamableClass, opstream, ipstream**

write `virtual void write( opstream& os) = 0;` **protected**

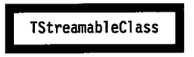
This pure virtual function must be overridden (or redeclared as pure virtual) in every derived class. The overriding **write** function for each streamable class must write the necessary data members to the opstream object *os*. **write** is usually implemented by calling the base class's **write** (if any), then inserting any additional data members in the derived class.

See also: **TStreamableClass, opstream**

Friends

The classes **opstream** and **ipstream** are friends of **TStreamable**, so all their member functions can access the private members of **TStreamable**.

TStreamableClass TOBJSTRM.H



TStreamableClass is used by **TStreamableTypes** and **pstream** in the registration of streamable classes.

Member functions

constructor `TStreamableClass( const char *n, BUILDER b, int d );`

Creates a **TStreamable** object with the given name and the given builder function, then calls **registerType**. Each streamable class ***TClassname*** has a **build** member function. There are also the familiar non-member overloaded **>>** and **<<** operators for stream I/O associated with each streamable class. 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 an **extern** reference such as:

```
extern TStreamableClass registerTClassName;
```

where **TClassName** is the name of the class for which objects need to be streamed. (Note that *registerTClassName* 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.

BUILDER is **typedef**ed as follows:

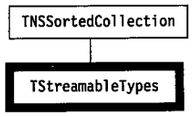
```
typedef TStreamable *(*BUILDER)();
```

See also: **TStreamable, TStreamableTypes, read, write, build, TStreamableTypes::registerTypes, ipstream, opstream**

Friends

The classes **TStreamableTypes, opstream,** and **ipstream** are friends of **TStreamableClass**, so all their member functions can access the private members of **TStreamableClass**.

TStreamableTypes TOBJSTRM.H



TStreamableTypes, derived privately from **TNSSortedCollection**, maintains a database of all registered streamable types used in an application. **opstream** and **ipstream** use this database to determine the correct **read** and **write** functions for particular objects. Because of the private derivation, all the inherited members are private within **TStreamableTypes**.

Member functions

constructor `TStreamableTypes();`

Calls the base **TNSCollection** constructor to create a **TStreamableTypes** collection.

See also: **TNSCollection::TNSCollection**

destructor `~TStreamableTypes();`

Sets the collection *limit* to 0 without destroying the collection (since the *shouldDelete* data member is set to *False*).

See also: **TNSCollection::~TNSCollection, TNSCollection::shouldDelete**

lookup
```
const TStreamableClass *lookup( const char *name );
```

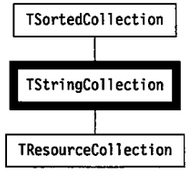
Returns a pointer to the class in the collection corresponding to the argument *name*, or returns 0 if no match.

registerTypes
```
void registerTypes( const TStreamableClass *d );
```

Registers the argument class by inserting *d* in the collection.

See also: **TNSCollection::insert, TStreamableClass**

TStringCollection RESOURCE.H



TStringCollection is a simple derivative of **TSortedCollection** implementing a sorted list of ASCII strings. **TStringCollection::compare** is overridden to provide the conventional lexicographic ASCII string ordering. You can override **compare** to allow for other orderings, such as those for non-English character sets.

Member functions

constructor
```
TStringCollection(short aLimit, short aDelta);
```

Creates a **TStringCollection** object with the given values.

constructor
```
TStringCollection( StreamableInit streamableInit);
```
protected

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TSortedCollection::TSortedCollection**

build
```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

compare
```
virtual int compare(void *key1, void *key2);
```

Compares the "strings" *key1* and *key2* as follows: return <0, 0, >0 if *key1* < *key2*; 0 if *key1* = *key2*; and +1 if *key1* > *key2*.

See also: **TSortedCollection::search**

freeItem
```
virtual void freeItem(void *item);
```

TS
Std class

Removes the string *item* from the sorted collection and disposes of the string.

read `virtual void *read( ipstream& is );`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream**

readItem `void *TStringCollection::readItem( ipstream& is );`

Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to read the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, ipstream**

write `virtual void write( opstream& os );`

Writes to the output stream *os*.

See also: **TStreamableClass, TStreamable, opstream**

writeItem `void TStringCollection::writeItem( void *obj, opstream& os );`

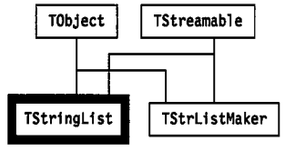
Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to write the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TStringCollection** but are not member functions; see page 232 for more information.

TStringList RESOURCE.H



TStringList provides a mechanism for accessing strings stored on a stream. Each string in a string list is identified by a unique number (`ushort key`) between 0 and 65,535. String lists take up less memory than normal string literals, since the strings are stored on a stream instead of in memory. Also, string lists permit easy internationalization, as the strings are not hard-coded in your program.

TStringList has member functions only for accessing strings; you must use **TStrListMaker** to create string lists.

TStrIndexRec The small class **TStrIndexRec** is used with string lists. It is defined as follows in RESOURCE.H:

```
class TStrIndexRec
{
public:
   TStringIndexRec();       // constructor sets count=0
   ushort key;
   ushort count;
   ushort offset;
}
```

The *index* data member in **TStringList** points to a **TStrIndexRec** object.

Member functions

constructor `TStringList( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TStrListMaker::TStrListMaker, TStringList::get**

destructor `~TStringList();`

Deallocates the memory allocated to the string list.

See also: **TStrListMaker::TStrListMaker, ~TStringList**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

get `void get (char *dest, ushort key);`

Returns in *dest* the string given by *key*, or an empty string if there is no string with the given *key*.

See also: **TStrListMaker::put**

read `virtual void *read( ipstream& is);`

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream** classes

write `virtual void write( opstream& os);`

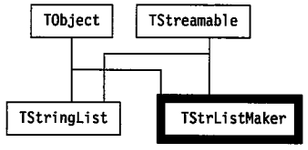
Writes to the output stream *os*.

See also: **TStreamableClass, TStreamable, opstream** classes

Related functions

Certain operator functions are related to **TStringList** but are not member functions; see page 232 for more information.

TStrListMaker RESOURCE.H



TStrListMaker is a simple object type used to create string lists for use with **TStringList**.

Member functions

constructor `TStrListMaker(ushort aStrSize, ushort aIndexSize);`

Creates an in-memory string list of size *aStrSize* with an index of *aIndexSize* elements. A string buffer and an index buffer of the specified size is allocated on the heap.

aStrSize must be large enough to hold all strings to be added to the string list—each string occupies its length plus a final 0.

As strings are added to the string list (using **TStrListMaker::put**), a string index is built. Strings with contiguous keys (such as and *sError* in the example above) are recorded in one index record, up to 16 at a time. *aIndexSize* must be large enough to allow for all index records generated as strings are added. Each index entry occupies 6 bytes.

constructor `TStrListMaker( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TStringList::put, ~TStrListMaker**

destructor `~TStrListMaker();`

Frees the memory allocated to the string list maker.

See also: **TStrListMaker::TStrListMaker**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData, TStreamable**

put `void put(ushort key, const char *s);`

Adds the given string *s* to the calling string list (with the given numerical *key*).

write `virtual void write( opstream& os);`

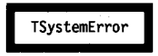
Writes to the output stream *os*.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TStrListMaker** but are not member functions; see page 232 for more information.

TSystemError SYSTEM.H



TSystemError provides system error handlers and associated services. Most of its members are private and will not be of direct interest in normal Turbo Vision applications.

Data members

ctrlBreakHit `static Boolean near ctrlBreakHit;`

Set *True* if *Ctrl-Break* is keyed during program execution.

Member functions

constructor `TSystemError();`

Creates a **TSystemError** object and installs the system error handler by calling **resume**.

destructor `~TSystemError();`

Removes the system error handler by calling **suspend**.

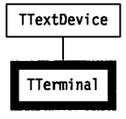
resume `static void resume();`

Installs the system error handler.

suspend `static void suspend();`

Removes the system error handler.

TTerminal TEXTVIEW.H



TTerminal implements a "dumb" terminal with buffered string reads and writes. The default is a cyclic buffer of 64K bytes.

Data members

buffer `char *buffer;`

Pointer to the first byte of the terminal's buffer.

bufSize `ushort bufSize;`

The size of the terminal's buffer in bytes.

queBack `ushort queBack;`

Offset (in bytes) of the last byte stored in the terminal buffer.

queFront `ushort queFront;`

Offset (in bytes) of the first byte stored in the terminal buffer.

Member functions

constructor
```
TTerminal(const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar
          *aVScrollBar, ushort aBufSize);
```

Creates a **TTerminal** object with the given *bounds*, horizontal and vertical scroll bars, and buffer by calling **TTextDevice::TTextDevice** with the *bounds* and scroller arguments, then creating a buffer (pointed to by *buffer*) with *bufSize* equal to *aBufSize*. *growMode* is set to *gfGrowHiX* | *gfGrowHiY*. *queFront* and *queBack* are both initialized to 0, indicating an empty buffer. The cursor is shown at the view's origin, (0,0).

See also: **TScroller::TScroller, TTextDevice::TTextDevice**

destructor `~TTerminal();`

Deallocates the buffer and calls **~TTextDevice**.

See also: **~TScroller, ~TTextDevice**

bufDec `void bufDec(ushort& val);`

Used to manipulate queue offsets with wrap around: If *val* is zero, *val* is set to (*bufSize* – 1); otherwise, *val* is decremented.

See also: **TTerminal::bufInc**

bufInc `void bufInc(ushort& val);`

Used to manipulate a queue offsets with wrap around: Increments *val* by 1, then if *val* >= *bufSize*, *val* is set to zero.

See also: **TTerminal::bufDec**

canInsert

```
Boolean canInsert(ushort amount);
```

Returns *True* if the number of bytes given in *amount* can be inserted into the terminal buffer without having to discard the top line. Otherwise, returns *False*.

do_sputn

```
int do_sputn( const char *s, int );
```

Overrides the corresponding function in class **streambuf**. This is an internal function that is called whenever a character string is to be inserted into the internal buffer.

draw

```
virtual void draw();
```

Called whenever the **TTerminal** scroller needs to be redrawn; for example, when the scroll bars are clicked on, the view is unhidden or resized, the *delta* values are changed, or when added text forces a scroll.

nextLine

```
ushort nextLine(ushort pos);
```

Returns the buffer offset of the start of the line that follows the position given by *pos*.

See also: **TTerminal::prevLines**

prevLines

```
ushort prevLines(ushort pos, ushort Lines);
```

Returns the offset of the start of the line that is *Lines* lines previous to the position given by *pos*.

See also: **TTerminal::nextLine**

queEmpty

```
Boolean queEmpty();
```

Returns *True* if *queFront* is equal to *queBack*.

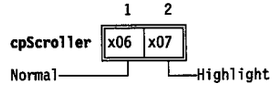
See also: **TTerminal::queFront**, **TTerminal::queBack**

Friends

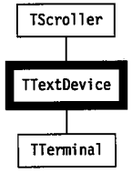
The function **genRefs** is a friend of **TTerminal**.

Palette

Terminal objects use the default palette, *cpScroller*, to map onto the sixth and seventh entries in the standard application palette.



TTextDevice TEXTVIEW.H



TTextDevice is a scrollable TTY-type text viewer/device driver. Apart from the data members and member functions inherited from **TScroller**, **TTextDevice** defines virtual member functions for reading and writing strings from and to the device. **TTextDevice** exists solely as a base type for deriving real terminal drivers. **TTextDevice** uses **TScroller's** destructor.

Member functions

constructor

```
TTextDevice(const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar
        *aVScrollBar);
```

Creates a **TTextDevice** object with the given *bounds*, horizontal and vertical scroll bars calling **TTextScroller::TTextScroller** with the *bounds* and scroller arguments.

See also: **TScroller::TScroller**

do_sputn

```
int do_sputn( const char *s, int );
```

Overrides the corresponding function in class **streambuf**. This is an internal function that is called whenever a character string is to be inserted into the internal buffer.

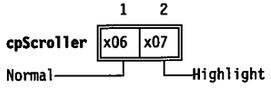
overflow

```
int overflow( int );
```

Overrides the corresponding function in class **streambuf**. When the internal buffer in a **streambuf** is full and the **iostream** associated with that **streambuf** tries to put another character into the buffer, **overflow** is called. Its argument is the character that caused the overflow. In **TTextDevice**, the underlying **streambuf** has no buffer, so every character results in an overflow. Classes derived from **TTextDevice**, such as **TTerminal**, should treat this character simply as another character in the output stream.

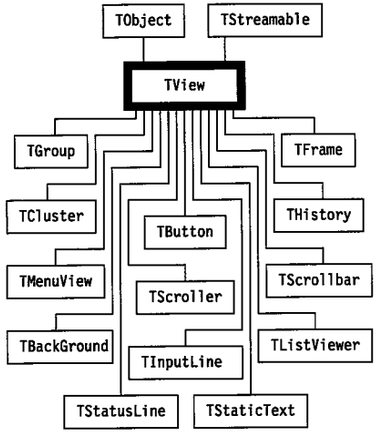
Palette

Text device objects use the default palette *cpScroller* to map onto the sixth and seventh entries in the standard application palette.



TView VIEWS.H

To avoid clutter in the hierarchy charts, we have omitted much detail. See page 75 for the full hierarchy.



Most programs make use of the **TView** derivatives: **TFrame, TScrollBar, TScroller, TListViewer, TGroup,** and **TWindow** objects.

TView objects are rarely instantiated in Turbo Vision programs. The **TView** class exists to provide basic data and functionality for its derived classes.

Data members

commandSetChanged

```
static Boolean commandSetChanged;
```

Set to *True* whenever the view's command set is changed via an **enable, disable**, or **setCommand** call.

See also: **TView::enableCommand(s), TView::disableCommand(s), TView::setCommands**

curCommandSet

```
static TCommandSet curCommandSet;
```

Holds the set of commands currently enabled for this view. Initially, the following commands are disabled: *cmZoom, cmClose, cmResize, cmNext, cmPrev*. This data member is constantly monitored by **handleEvent** to determine which of the received command events needs to be serviced. *curCommandSet* should not be altered directly: use the appropriate **set, enable**, or **disable** calls.

See also: *cmXXXX* constants, **TView::setCommands, TView::enableCommand(s), TView::disableCommand(s)**

cursor

```
TPoint cursor;
```

The location of the hardware cursor within the view. The cursor is visible only if the view is focused (*sfFocused*) and the cursor turned on (*sfCursorVis*). The shape of the cursor is either an underline or block (determined by *sfCursorIns*).

See also: **setCursor, showCursor, hideCursor, normalCursor, blockCursor**

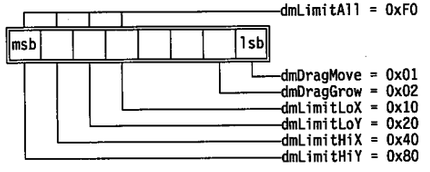
dragMode

```
uchar dragMode;
```

Determines how the view should behave when mouse-dragged.

The *dragMode* bits are defined as follows:

Figure 13.1 *dragMode* bit mapping



The *dragMode* masks are defined in Chapter 16 under "*dmXXXX dragMode* constants."

See also: **TView::dragView**

errorAttr `static uchar errorAttr;`

Attribute used to signal an invalid palette selection. For example, **mapColor** returns *errorAttr* if it is called with an invalid *color* argument. By default, *errorAttr* is set to "0CF", which shows as flashing red on white.

See also: **TView::mapColor**

eventMask `ushort eventMask;`

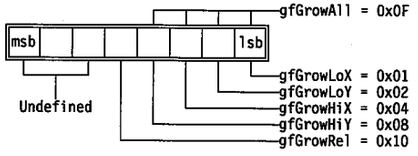
eventMask is a bit mask that determines which event classes will be recognized by the view. The default *eventMask* enables *evMouseDown*, *evKeyDown*, and *evCommand*. Assigning 0xFFFF to *eventMask* causes the view to react to all event classes; conversely, a value of zero causes the view to not react to any events. For detailed descriptions of event classes, see "*evXXXX* event constants" in Chapter 16.

See also: **TView::handleEvent**

growMode `uchar growMode;`

Determines how the view will grow when its owner view is resized. *growMode* is assigned one or more of the following *growMode* masks:

Figure 13.2 *growMode* bit mapping



Example: `growMode = gfGrowLoX | gfGrowLoY;`

See also: *gfXXXX growMode* constants

helpCtx `ushort helpCtx;`

The help context of the view. When the view is focused, this data member will represent the help context of the application, unless the context number is *hcNoContext*, in which case there is no help context.

See also: **TView::getHelpCtx**.

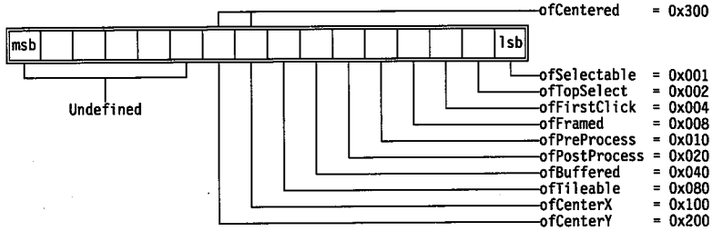
next `TView *next;`

Pointer to next peer view in Z-order. If this is the last subview, *next* points to *owner*'s first subview.

options `ushort options;`

The *options* word flags determine various behaviors of the view. The *options* bits are defined as follows:

Figure 13.3 *options* bit flags



For detailed descriptions of the *options* flags, see "*ofXXXX* option flag constants" in Chapter 16. See also page 197 and *ff* in Chapter 10.

origin `TPoint origin;`

The (*x*, *y*) coordinates, relative to the owner's *origin*, of the top-left corner of the view.

See also: **moveTo**, **locate**

owner `TGroup *owner;`

owner points to the **TGroup** object that owns this view. If 0, the view has no owner. The view is displayed within its owner's view and will be clipped by the owner's bounding rectangle.

showMarkers `static Boolean showMarkers;`

Used to indicate whether indicators should be placed around focused controls. **TProgram::initScreen** sets *showMarkers* to *True* if the video mode is monochrome; otherwise it is *False*. The value may, however, be set *on* in color and black and white modes if desired.

See also: **TProgram::initScreen**, *specialChars*

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

size `TPoint size;`

The size of the view.

See also: **growTo**, **locate**

state `ushort state;`

The state of the view is represented by bits set or clear in the *state* data member. Many **TView** member functions test and/or alter the *state* data member by calling **TView::setState.**

TView::getState*(aState)* returns *True* if the view's *state* is *aState*. The *state* bits are represented mnemonically by *sfXXXX* constants, described in Chapter 16 under "*sfXXXX* state flag constants."

Member functions

constructor `TView(const TRect& bounds);`

Creates a **TView** object with the given *bounds* rectangle. **TView::TView** calls the **TObject** constructor and sets the data members of the new **TView** to the following values:

constructor `TView( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

cursor	(0, 0)
dragMode	*dmLimitLoY*
eventMask	*evMouseDown* | *evKeyDown* | *evCommand*
growMode	0
helpCtx	*hcNoContext*
next	0
options	0
origin	(bounds.A.x, bounds.A.y)
owner	0
size	(bounds.B.x – bounds.A.x, bounds.B.y – bounds.A.y)
state	*sfVisible*

See also: **TObject::TObject**

destructor `~TView();`

Hides the view and then, if it has an owner, removes it from the group.

blockCursor `void blockCursor();`

Sets *sfCursorIns* to change the cursor to a solid block. The cursor will only be visible if *sfCursorVis* is also set (and the view is visible).

See also: *sfCursorIns*, *sfCursorVis*, **TView::normalCursor, TView::showCursor, TView::hideCursor**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

calcBounds `virtual void calcBounds(TRect& bounds, TPoint delta);`

When a view's owner changes size, the owner repeatedly calls **calcBounds** and **changeBounds** for all its subviews. **calcBounds** must calculate the new bounds of the view given that its owner's size has changed by *delta*, and return the new bounds in *bounds*.

TView::calcBounds calculates the new bounds using the flags specified in the *TView::growMode* data member.

See also: **TView::getBounds, TView::changeBounds,** *gfXXXX growMode* constants

changeBounds `virtual void changeBounds(const TRect& bounds);`

changeBounds must change the view's bounds (*origin* and *size* data members) to the rectangle given by the *bounds* parameter. Having changed the bounds, **changeBounds** must then redraw the view. **changeBounds** is called by various **TView** member functions, but should never be called directly.

TView::changeBounds first calls **setBounds***(bounds)* and then calls **drawView.**

See also: **TView::locate, TView::moveTo, TView::growTo**

clearEvent `void clearEvent(TEvent& event);`

Standard member function used in **handleEvent** to signal that the view has successfully handled the event. Sets *event.what* to *evNothing* and *event.message.infoPtr* to **this.**

See also: **handleEvent** member functions

commandEnabled `static Boolean commandEnabled(ushort command);`

Returns *True* if the given *command* is currently enabled; otherwise it returns *False*. Note that when you change a modal state, you can then disable and enable commands as you wish; when you return to the previous modal state, however, the original command set will be restored.

See also: **TView::disableCommand, TView::enableCommand, TView::setCommands.**

containsMouse

```
Boolean containsMouse(TEvent& event);
```

Returns *True* if a mouse event occurs inside the calling view, otherwise returns *False*.

See also: **mouseInView**

dataSize

```
virtual ushort dataSize();
```

dataSize must be used to return the size of the data read from and written to data records by **setData** and **getData**. The data record mechanism is typically used only in views that implement controls for dialog boxes.

TView::dataSize returns zero to indicate that no data was transferred.

See also: **TView::getData, TView::setData**

disableCommand

```
static void disableCommand(ushort command);
```

Disables the given command. If the command set is changed by the call, *commandSetChanged* is set *True*.

See also: **TView::enableCommand, TView::disableCommands**

disableCommands

```
static void disableCommands(TCommandSet& commands);
```

Disables the commands specified in the *commands* argument.

See also: **TView::commandEnabled, TView::enableCommands, TView::setCommands**

dragView

```
virtual void dragView(TEvent& event, uchar mode, TRect& limits, TPoint
minSize, TPoint maxSize);
```

Drags the view using the dragging mode given by the *dmXXXX* flags set in the *mode* argument. *limits* specifies the rectangle (in the owner's coordinate system) within which the view can be moved, and *min* and *max* specify the minimum and maximum sizes the view can shrink or grow to. The event leading to the dragging operation is needed in **event** to distinguish mouse dragging from use of the cursor keys.

See also: **TView::dragMode**, *dmXXXX* drag mode constants

draw

```
virtual void draw();
```

Called whenever the view must draw (display) itself. **draw** must cover the entire area of the view. This member function must be overridden appropriately for each derived class. **draw** is seldom called directly, since it is more efficient to use **drawView**, which draws only views that are exposed; that is, some or all of the view is visible on the screen. If required, **draw** can call **getClipRect** to obtain the rectangle that needs

redrawing, and then only draw that area. For complicated views, this can improve performance noticeably.

See also: **TView::drawView**

drawCursor

```
void drawCursor();
```

If the view is *sfFocused*, the cursor is reset with a call to **resetCursor**.

See also: **TView::resetCursor**

drawHide

```
void drawHide(TView *lastView);
```

Calls **drawCursor** followed by **drawUnderView**. The latter redraws all subviews (with shadows if required) until the given *lastView* is reached.

drawShow

```
void drawShow(TView *lastView);
```

Calls **drawView**, then if *state* has the *sfShadow* bit set, **drawUnderView** is called to draw the shadow.

drawUnderRect

```
void drawUnderRect(TRect& r, TView *lastView);
```

Calls **owner->clip.intersect**(*r*) to set the area that needs drawing. Then all the subviews from the next view to the given *lastView* are drawn using **drawSubViews**. Finally, *owner->clip* is reset to *owner->getExtent*.

See also: **TView::drawSubViews**

drawUnderView

```
void drawUnderView(Boolean doShadow, TView *lastView);
```

Calls **drawUnderRect**(*r*, *lastView*), where *r* is the calling view's current *bounds*. If *doShadow* is *True*, the view's bounds are first increased by *shadowSize*.

See also: **TView::drawUnderRect**, *shadowSize*

drawView

```
void drawView();
```

Calls **draw** if **exposed** returns *True*, indicating that the view is exposed (see *sfExposed*). If **exposed** returns *False*, **drawView** does nothing. You should call **drawView** (not **draw**) whenever you need to redraw a view after making a change that affects its visual appearance.

See also: **TView::draw, TGroup::redraw, TView::exposed**

enableCommand

```
static void enableCommand(ushort command);
```

Enables the given command. If the command set is changed by this call, *commandSetChanged* is set *True*.

See also: **TView::disableCommand, TView::enableCommands**

enableCommands

```
static void enableCommands(TCommandSet& commands);
```

Enables all the commands in the *commands* argument. If the command set is changed by this call, *commandSetChanged* is set *True*.

See also: **TCommandSet**, *commandSetChanged*, **TView::disableCommands**, **TView::getCommands**, **TView::commandEnabled**, **TView::setCommands**.

endModal

```
virtual void endModal(ushort command);
```

Calls **topView** to seek the top most modal view. If there is none such (that is, if **topView** returns 0) no further action is taken. If there is a modal view, that view calls **endModal**, and so on.

The net result is that **endModal** terminates the current modal state. The *command* argument is passed to the **execView** that created the modal state in the first place.

See also: **TGroup::execView**, **TGroup::execute**, **TGroup::endModal**

eventAvail

```
Boolean eventAvail();
```

Calls **getEvent** and returns *True* if an event is available. Calls **putEvent** if an event is available.

See also: **TView::mouseEvent**, **TView::keyEvent**, **TView::getEvent**, **TView::putEvent**

execute

```
virtual ushort execute();
```

execute is called from **TGroup::execView** whenever a view becomes modal. If a view is to allow modal execution, it must override **execute** to provide an event loop. The value returned by **execute** will be the value returned by **TGroup::execView**.

The default **TView::execute** simply returns *cmCancel*.

See also: *sfModal*, **TGroup::execute**, **TGroup::execView**, *cmCancel*

exposed

```
Boolean exposed();
```

Returns *True* if any part of the view is visible on the screen.

See also: *sfExposed*, **TView::drawView**

getBounds

```
TRect getBounds();
```

Returns the current value of *size*, the bounding rectangle of the view in its owner's coordinate system. *TRect::a* is set to *origin*, and *TRect::b* is set to the sum of *origin* and *size*.

See also: **TView::origin**, **TView::size**, **TView::calcBounds**, **TView::changeBounds**, **TView::setBounds**, **TView::getExtent**

getClipRect

```
TRect getClipRect();
```


Returns the *clip* data member: the minimum rectangle that needs redrawing during a call to **draw**. For complicated views, **draw** can use **getClipRect** to improve performance noticeably.

See also: **TView::draw**, **TView::drawView**

getColor

```
ushort getColor(ushort color);
```

Maps the palette indices in the low and high bytes of *color* into physical character attributes by tracing through the palette of the view and the palettes of all its owners.

See also: **TView::getPalette**, **TView::mapColor**

getCommands

```
static void getCommands(TCommandSet& commands);
```

Returns, in the *commands* argument, the current command set.

See also: **TView::commandsEnabled**, **TView::enableCommands**, **TView::disableCommands**, **TView.setCommands**

getData

```
virtual void getData(void *rec);
```

getData must copy *DataSize* bytes from the view to the data record given by the *rec* pointer. The data record mechanism is typically used only in views that implement controls for dialog boxes. The default **TView::getData** does nothing.

See also: **TView::dataSize**, **TView::setData**

getEvent

```
virtual void getEvent(TEvent& event);
```

Returns the next available event in the *event* argument. Returns *evNothing* if no event is available. By default, it calls the view's owner's **getEvent**.

See also: **TView::eventAvail**, **TProgram::idle**, **TView::handleEvent**, **TView::putEvent**

getExtent

```
TRect getExtent();
```

Returns the extent rectangle of the view. *TRect::a* is set to (0, 0), and *TRect::b* is set to *size*.

See also: **TView::origin**, **TView::size**, **TView::calcBounds**, **TView::changeBounds**, **TView::setBounds**, **TView::getBounds**

getHelpCtx

```
virtual ushort getHelpCtx();
```

getHelpCtx returns the view's help context.

The default **TView.getHelpCtx** returns the value in the *helpCtx* data member, or returns *hcDragging* if the view is being dragged (see *sfDragging*).

See also: *helpCtx*

getPalette `virtual TPalette& getPalette() const;`

getPalette must return a string representing the view's palette. This can be 0 (empty string) if the view has no palette. **getPalette** is called by **writeChar**, and **writeStr** when converting palette indices to physical character attributes. The default return value of 0 causes no color translation to be performed by this view. **getPalette** is almost always overridden in derived classes.

See also: **TView::getColor**, **TView::mapColor**, **write***XXX*

getState `Boolean getState(ushort aState);`

Returns *True* if the state given in *aState* is set in the data member *state*.

See also: *state*, **TView::setState**

growTo `void growTo(short x, short y);`

Grows or shrinks the view to the given size using a call to **TView::locate**.

See also: **TView::origin**, **TView::size**, **TView::locate**, **TView::moveTo**

handleEvent `virtual void handleEvent(TEvent& event);`

handleEvent is the central member function through which all Turbo Vision event handling is implemented. The *what* data member of the *event* parameter contains the event class (*evXXXX*), and the remaining *event* data members further describe the event. To indicate that it has handled an event, **handleEvent** should call **clearEvent**. **handleEvent** is almost always overridden in derived classes.

TView::handleEvent handles *evMouseDown* events as follows: If the view is not selected (*sfSelected*) and not disabled (*sfDisabled*), and if the view is selectable (*ofSelectable*), then the view selects itself by calling **Select**. No other events are handled by **TView::handleEvent**.

See also: **TView::clearEvent**

hide `void hide();`

Hides the view by calling **setState** to clear the *sfVisible* flag in *state*.

See also: *sfVisible*, **TView::setState**, **TView::show**

hideCursor `void hideCursor();`

Hides the cursor by clearing the *sfCursorVis* bit in *state*.

See also: *sfCursorVis*, **TView::showCursor**

keyEvent `void keyEvent(TEvent& event);`

Returns, in the *event* variable, the next *evKeyDown* event. It waits, ignoring all other events, until a keyboard event becomes available.

See also: **TView::getEvent**, **TView::eventAvail**

locate `void locate(TRect& bounds);`

Changes the bounds of the view to those of the *bounds* argument. The view is redrawn in its new location. **locate** calls **sizeLimits** to verify that the given *bounds* are valid, and then calls **changeBounds** to change the bounds and redraw the view.

See also: **TView::growTo**, **TView::moveTo**, **TView::changeBounds**

makeFirst `void makeFirst();`

Moves the view to the top of its owner's subview list. A call to **makeFirst** corresponds to **putInFrontOf**(*owner->first()*).

See also: **TView::putInFrontOf**

makeGlobal `TPoint makeGlobal(TPoint source);`

Converts the *source* point coordinates from local (view) to global (screen) and returns the result.

See also: **TView::makeLocal**

makeLocal `TPoint makeLocal(TPoint source);`

Converts the *source* point coordinates from global (screen) to local (view) and returns the result. Useful for converting the *event.where* data member of an *evMouse* event from global coordinates to local coordinates; for example, `mouseLoc = makeLocal (eventWhere)`.

See also: **TView::MakeGlobal**, **TView::mouseInView**

mapColor `uchar mapColor(uchar);`

Maps the given *color* to an offset into the current palette. **mapColor** works by calling **getPalette** for each owning group in the chain. It succesively maps the offset in each palette until the ultimate owning palette is reached. If *color* is invalid (for example, out of range) for any of the palettes encountered in the chain, **mapColor** returns *errorAttr*.

See also: **TView::getPalette**, **TView::errorAtrr**

mouseEvent `Boolean mouseEvent(TEvent& event, ushort mask);`

Sets the next mouse event in the *event* argument. Returns *True* if this event is in the *mask* argument. Also returns *False* if an *evMouseUp* event occurs. This member function lets you track a mouse while its button is down; for example, in drag block-marking operations for text editors.

Here's an extract of a **handleEvent** routine that tracks the mouse with the view's cursor.

```
virtual void TMyView::handleEvent(TEvent& event)
{
   TView::handleEvent(event);
   switch ( event.what )
   {
      case evMouseDown:
         repeat
         {
            makeLocal(event.where, mouse);
            setCursor(mouse.x, mouse.y);
         }
         until !(mouseEvent(event, evmouseMove));
         clearEvent(event);
         break;
   }
    ...
};
```

See also: *evXXX* event masks, **TView::keyEvent**, **TView::getEvent**

mouseInView

```
Boolean mouseInView(TPoint mouse);
```

Returns *True* if the *mouse* argument (given in *global* coordinates) is within the calling view.

See also: **TView::MakeLocal**, **TView::MakeGlobal**

moveTo

```
void moveTo(short x, short y);
```

Moves the *origin* to the point (x,y) relative to the owner's view. The view's *size* is unchanged.

See also: *TView::origin*, *TView::size*, **TView::locate**, **TView::growTo**

nextView

```
TView *nextView();
```

Returns a pointer to the next subview in the owner's subview list. A 0 is returned if the calling view is the last one in its owner's list.

See also: **TView::prevView**, **TView::prev**, **TView::next**

normalCursor

```
void normalCursor();
```


Clears the *sfCursorIns* bit in *state*, thereby making the cursor into an underline. If *sfCursorVis* is set, the new cursor will be displayed.

See also: *sfCursorIns*, *sfCursorVis*, **TView::hideCursor**, **TView::blockCursor**, **TView::hideCursor**

prev

```
TView *prev();
```

Returns a pointer to the previous subview in the owner's subview list. If the calling view is the first one in its owner's list, **prev** returns the last view in the list. Note that **TView.prev** treats the list as circular, whereas **TView::prevView** treats the list linearly.

See also: **TView::nextView**, **TView::prevView**, **TView::next**

prevView

```
TView *prevView();
```

Returns a pointer to the previous subview in the owner's subview list. 0 is returned if the calling view is the first one in its owner's list. Note that **TView::prev** treats the list as circular, whereas **TView::prevView** treats the list linearly.

See also: **TView::nextView**, **TView::prev**

putEvent

```
virtual void putEvent(TEvent& event);
```

Puts the event given by **event** into the event queue, causing it to be the next event returned by **getEvent**. Only one event can be pushed onto the event queue in this fashion. Often used by views to generate command events; for example,

```
event.what = evCommand;
event.command = cmSaveAll;
event.infoPtr = 0;
putEvent(event);
```

The default **TView::putEvent** calls the view's owner's **putEvent**.

See also: **TView::eventAvail**, **TView::getEvent**, **TView::handleEvent**

putInFrontOf

```
void putInFrontOf(TView *target);
```

Move the calling view in front of the **target** view in the owner's subview list. The call

```
MyView.putInFrontOf(owner->first);
```

is equivalent to **MyView.makeFirst**. This member function works by changing pointers in the subview list. Depending on the position of the other views and their visibility states, **putInFrontOf** may obscure (clip)

underlying views. If the view is selectable (see *ofSelectable*) and is put in front of all other subviews, then the view becomes selected.

See also: **TView::makeFirst**

read

```
virtual void *read( ipstream& is);
```

Reads from the input stream *is*.

See also: **TStreamableClass, TStreamable, ipstream** classes

resetCursor

```
virtual void resetCursor();
```

Resets the cursor.

select

```
void select();
```

Selects the view (see *sfSelected*). If the view's owner is focused, then the view also becomes focused (see *sfFocused*). If the view has the *ofTopSelect* flag set in its *options* data member, then the view is moved to the top of its owner's subview list (using a call to **TView::makeFirst**).

See also: *sfSelected*, *sfFocused*, *ofTopSelect*, **TView::makeFirst**

setBounds

```
void setBounds(const TRect& bounds);
```

Sets the bounding rectangle of the view to the value given by the *bounds* parameter. The *origin* data member is set to *bounds.a*, and the *size* data member is set to the difference between *bounds.b* and *bounds.a*. The **setBounds** member function is intended to be called only from within an overridden **changeBounds** member function—you should never call **setBounds** directly.

See also: *TView::origin*, *TView::size*, **TView::calcBounds, TView::changeBounds, TView::getBounds, TView::getExtent**

setCommands

```
static void setCommands(TCommandSet& commands);
```

Changes the current command set to the given *commands* argument.

See also: **TView::enableCommands, TView::disableCommands**

setCursor

```
void setCursor(short x, short y);
```

Moves the hardware cursor to the point (*x*,*y*) using view-relative (local) coordinates. (0,0) is the top-left corner.

See also: **TView::makeLocal, TView::hideCursor, TView::showCursor**

setData

```
virtual void setData(void *rec);
```


setData must copy **dataSize** bytes from the data record given by *rec* to the view. The data record mechanism is typically used only in views that implement controls for dialog boxes.

The default **TView::setData** does nothing.

See also: **TView::dataSize, TView::getData**

setState

```
virtual void setState(ushort aState, Boolean enable);
```

Sets or clears a state flag in the *state* data member. The *aState* parameter specifies the state flag to modify (see *sfXXXX*), and the *enable* parameter specifies whether to turn the flag off (*False*) or on (*True*). **setState** then carries out any appropriate action to reflect the new state, such as redrawing views that become exposed when the view is hidden (*sfVisible*), or reprogramming the hardware when the cursor shape is changed (*sfCursorVis* and *sfCursorIns*).

setState is sometimes overridden to trigger additional actions that are based on state flags. **TFrame**, for example, overrides **setState** to redraw itself whenever a window becomes selected or is dragged:

```
void TFrame::setState( ushort aState, Boolean enable )
{
   TView::setState(aState, enable);
   if (aState & (sfActive | sfDragging) != 0)
      drawView();
}
```

Another common reason to override **setState** is to enable or disable commands that are handled by a particular view:

```
void TMyView::setState( ushort aState, Boolean enable )
{ TCommandSet myCommands;
  myCommands.enableCmd(cmCut);
  myCommands.enableCmd(cmCopy);
  myCommands.enableCmd(cmPaste);
  myCommands.enableCmd(cmClear);
  TView::setState( aState, enable );
  if ( aState = sfSelected )
  {
     if enable
        enableCommands(myCommands)
     else
        disableCommands(myCommands)
  }
}
```

See also: **TView::getState**, *TView::state*, *sfXXXX* state flags

show `void show();`

If the view is *sfVisible*, nothing happens. Otherwise, **show** displays the view by calling **setState** to set the *sfVisible* flag in *state*.

See also: **TView::setState**

showCursor `void showCursor();`

Turns on the hardware cursor by setting *sfCursorVis*. Note that the cursor is invisible by default.

See also: *sfCursorVis*, **TView::hideCursor**

sizeLimits `virtual void sizeLimits(TPoint& min, TPoint& max);`

Sets, in the *min* and *max* arguments, the minimum and maximum values that the *size* data member may assume.

The default **TView::sizeLimits** returns (0, 0) in *min* and *owner->size* in *max*. If *owner* is 0, *max.x* and *max.y* are both set to MAXSHORT.

See also: *TView::size*

topView `TView *topView();`

Returns a pointer to the current modal view, or 0 if none such.

valid `virtual Boolean valid(ushort command);`

Use this member function to check the validity of a view after it has been constructed or at the point in time when a modal state ends (due to a call to **endModal**).

A *command* argument of **cmValid** (*zero*) indicates that the view should check the result of its constructor: **valid**(*cmValid*) should return *True* if the view was successfully constructed and is now ready to be used, *False* otherwise.

Any other (nonzero) *command* argument indicates that the current modal state (such as a modal dialog box) is about to end with a resulting value of *command*. In this case, **valid** should check the validity of the view.

It is the responsibility of **valid** to alert the user in case the view is invalid.

The default **TView::valid** simply returns *True*.

See also: **TGroup::valid**, **TDialog::valid**, **TProgram::validView**

write `virtual void write( opstream& os);`

Writes to the output stream *os*.

See also: **TStreamableClass**, **TStreamable**, **opstream** classes

writeBuf
```
void writeBuf( short x, short y, short w, short h, const void far *b);
void writeBuf( short x, short y, short w, short h, const TDrawBuffer& b);
```

Writes the given buffer to the screen starting at the coordinates (*x*,*y*), and filling the region of width *w* and height *h*. Should only be used in **draw** member functions. The *buf* pointer is usually of type **TDrawBuffer&**, but it can be any array of words, each word containing a character in the low byte and an attribute in the high byte.

See also: **TView::draw**

writeChar `void writeChar(short x, short y, short c, uchar color, short count);`

Beginning at the point (*x*,*y*), writes *count* copies of the character *c* in the color determined by the *color*'th entry in the current view's palette. Should only be used in **draw** member functions.

See also: **TView::draw**

writeCStr `void writeCStr(short x, short y, char far *cstr, uchar color);`

Writes the "control" string *cstr* with the color attributes of the *color*'th entry in the view's palette, beginning at the point (*x*,*y*). **WriteCStr** recognizes the tilde (~) to toggle attributes and colors. Should only be used in **draw** member functions.

writeLine
```
void writeLine(short x, short y, short w, short h, const void far *buf);
void writeLine(short x, short y, short w, short h, const TDrawBuffer&
               buf);
```

Writes the line contained in the buffer *buf* to the screen, beginning at the point (*x*,*y*) within the rectangle defined by the width *w* and the height *h*. If *h* is greater than 1, the line will be repeated *h* times. Should only be used in **draw** member functions.

See also: **TView::draw**

writeStr `void writeStr(short x, short y, const char *str, uchar color);`

Writes the string *str* with the color attributes of the *color*'th entry in the view's palette, beginning at the point (*x*,*y*). Should only be used in **draw** member functions.

See also: **TView::draw**

Friends

The function **genRefs** is a friend of **TView**.

Related functions

Certain operator functions are related to **TView** but are not member functions; see page 232 for more information.

TVMemMgr BUFFERS.H

See Chapter 6, "Writing safe programs," for more on the safety pool.

TVMemMgr, in conjunction with **TBufEntryList** and the global operator **new**, provides low-level memory management for Turbo Vision applications. In particular, **TVMemMgr** manages the safety pool. For most applications, **TVMemMgr** and **TVBufListEntry** objects work behind the scenes without the need for specific programmer action or intervention.

Data members

safetyPool `static void *safetyPool;` **private**

safetyPool points to the safety pool memory allocation. If this value is zero, no safety pool has been allocated or if the safety pool is exhausted.

safetyPoolSize `static size_t safetyPoolSize;` **private**

The size in bytes of the current safety pool. This private member is updated by **TVMemMgr::resizeSafetyPool**. The default safety pool size is determined by the constant DEFAULT_SAFETY_POOL_SIZE, which is declared in BUFFERS.H and is currently set at 4,096 bytes.

inited `static int inited;` **private**

This data member indicates whether safety pool initialization has been attempted; it is strictly for internal use.

Member functions

constructor `TVMemMgr::TVMemMgr();`

Does nothing if safety pool already initialized (that is, if *inited* = 1). Otherwise, calls **resizeSafetyPool** to establish a default safety pool with the size given by the constant DEFAULT_SAFETY_POOL_SIZE. The latter is currently set to 4,096 (bytes) in BUFFERS.H. The constructor also sets *inited* to 1.

allocateDiscardable `static void allocateDiscardable(void *&adr, size_t sz);`

For internal use only. Tries to allocate a cache buffer (**TBufListEntry** object) of size sz. If successful, *adr* returns a pointer to the allocated buffer; otherwise, *adr* is set to 0. **TGroup::getBuffer** calls **allocateDiscardable** with *adr* set to the **TGroup::buffer** data member.

See also: **TVMemMgr::freeDiscardable**, operator **new**, **TGroup::getBuffer**

freeDiscardable `static void freeDiscardable( void *block );`

For internal use only. Frees the buffer allocated at *block* by an earlier **allocateDiscardable** call. **TGroup::freeBuffer** calls **freeDiscardable** with block set to the **TGroup::buffer** data member.

See also: **TVMemMgr::allocateDiscardable**, **TGroup::freeBuffer**

resizeSafetyPool `static void resizeSafetyPool( size_t sz = DEFAULT_SAFETY_POOL_SIZE );`

Resizes the safety pool to *sz* bytes. The previous safety pool, if one exists, is freed first. *inited* is set to 1, *safetyPool* is set to point to the new safety pool, and *safetyPoolSize* is set to *sz*. If the *sz* argument is omitted, the default value is assumed. If *sz* is 0, both *safetyPool* and *safetyPoolSize* are set to 0.

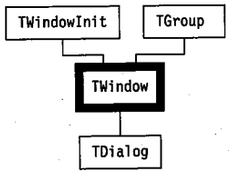
See also: **TVMemMgr** constructor, DEFAULT_SAFETY_POOL_SIZE

safetyPoolExhausted `static int safetyPoolExhausted();`

Returns 1 (*True*) if the safety pool is initialized *and* its allocation is exhausted. Otherwise, returns 0 (*False*). **safetyPoolExhausted** is called by the global function **lowMemory**.

See also: **lowMemory**, operator **new**

TWindow VIEWS.H



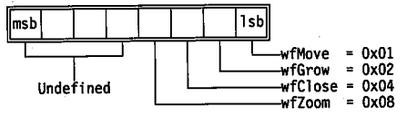
A **TWindow** object is a specialized group that typically owns a **TFrame** object, an interior **TScroller** object, and one or two **TScrollBar** objects. These attached subviews provide the "visibility" to the **TWindow** object. The **TFrame** object provides the familiar border, a place for an optional title and number, and functional icons (close, zoom, drag). **TWindow** objects have the "built-in" capability of moving and growing via mouse drag or cursor keystrokes. They can be zoomed and closed via mouse clicks in the appropriate icon regions. They also "know" how to work with scroll bars and scrollers. Numbered windows from 1 to 9 can be selected with the *Alt-n* keys (*n* = 1 to 9).

TWindow inherits multiply from **TGroup** and the virtual base class **TWindowInit**. The latter provides a constructor and **createFrame** member function used in creating and inserting a frame object. A similar technique is used for **THistoryWindow** and **THistInit**.

Data members

flags `uchar flags;`

The *flags* data member contains combinations of the following bits:



For definitions of the window flags, see "*wf*XXXX window flag constants" in Chapter 16.

frame `TFrame *frame;`

frame is a pointer to this window's associated **TFrame** object.

See also: **TWindow::initFrame**

number `short number;`

The number assigned to this window. If **TWindow::number** is between 1 and 9, the number will appear in the frame title, and the window can be selected with the *Alt-n* keys (*n* = 1 to 9).

palette `short palette;`

Specifies which palette the window is to use: *wpBlueWindow*, *wpCyanWindow*, or *wpGrayWindow*. The default palette is *wpBlueWindow*.

See also: **TWindow::getPalette**, *wpXXXX* constants

title `const char *title;`

A character string giving the (optional) title that appears on the frame.

zoomRect `TRect zoomRect;`

The normal, unzoomed boundary of the window.

Member functions

constructor `TWindow(const TRect& bounds, const char *aTitle, short aNumber);`

Calls the **TGroup** constructor to set *bounds*. Sets default *state* to *sfShadow*. Sets default *options* to (*ofSelectable* | *ofTopSelect*). Sets default *growMode* to *gfGrowAll* | *gfGrowRel*. Sets default *flags* to (*wfMove* | *wfGrow* | *wfClose* | *wfZoom*). Sets the *title* data member to *aTitle*, the *number* data member to *aNumber*. Calls **initFrame** by default, and if the resulting *frame* pointer is nonzero, inserts it in this window's group. Finally, the default *zoomRect* is set to the given *bounds*.

constructor `TWindow( StreamableInit streamableInit);` **protected**

Each streamable class needs a "builder" to allocate the correct memory for its objects together with the initialized vtable pointers. This is achieved by calling this constructor with an argument of type **StreamableInit**. Refer also to Chapter 8.

See also: **TFrame::initFrame**

destructor `~TWindow();`

Deletes title, then disposes of the window and any subviews by calling the parent destructor(s).

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

close

```
virtual void close();
```

Calls **valid**(*cmClose*); if *True* is returned, the calling window is deleted.

See also: **TView:valid**

getPalette

```
virtual TPalette& getPalette() const;
```

Returns the palette string given by the palette index in the *palette* data member.

See also: **TWindow::palette**

getTitle

```
virtual const char *getTitle(short maxSize);
```

Returns *title*, the window's title string.

See also: **TWindow::title**

handleEvent

```
virtual void handleEvent(TEvent& event);
```

First calls **TGroup::handleEvent**, and then handles events specific to a **TWindow** as follows:

The following *evCommand* events are handled if the *TWindow::flags* data member permits that operation: *cmResize* (move or resize the window using the **TView::dragView** member function), *cmClose* (close the window by creating a *cmCancel* event), and *cmZoom* (zoom the window using the **TWindow::zoom** member function).

evKeyDown events with a *keyCode* value of *kbTab* or *kbShiftTab* are handled by selecting the next or previous selectable subview (if any).

An *evBroadcast* event with a *command* value of *cmSelectWindowNum* is handled by selecting the window if the *event.infoInt* data member is equal to *number*.

See also: **TGroup::handleEvent**, *wfXXXX* constants, **TWindow::dragView**, **TWindow::zoom**

initFrame

```
virtual void initFrame(TRect);
```

Creates a **TFrame** object for the window and stores a pointer to the frame in the **TWindow::frame** data member. TWindow's constructor calls **initFrame**; it should never be called directly. You can override **initFrame** to instantiate a user-defined class derived from **TFrame** instead of the standard **TFrame**.

See also: **TWindow::TWindow**

read

```
virtual void *read( ipstream& is);
```

Reads from the input stream *is*.

See also: **TStreamable**, **ipstream** classes

setState

```
virtual void setState(ushort aState, Boolean enable);
```

First calls **TGroup::setState**(*aState, enable*). Then, if *aState* is equal to *sfSelected*, activates or deactivates the window and all its subviews using a call to **setState**(*sfActive, enable*), and calls **TView::enableCommands** or **TView::disableCommands** for *cmNext, cmPrev, cmResize, cmClose*, and *cmZoom*.

See also: **TGroup::setState**, **TView::enableCommands**, **TView::disableCommands**

shutDown

```
virtual void shutDown();
```

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

sizeLimits

```
virtual void sizeLimits(TPoint& min, TPoint& max);
```

Overrides **TView::sizeLimits**. First calls **TView::sizeLimits**(*min, max*) and then changes *min* to the minimum window size, currently preset to (16, 6).

See also: **TView::sizeLimits**

standardScrollBar

```
TScrollBar *standardScrollBar(ushort aOptions);
```

Creates, inserts, and returns a pointer to a "standard" scroll bar for the window. "Standard" means the scroll bar fits onto the frame of the window without covering the corners or the resize icon.

The *aOptions* parameter can be either *sbHorizontal* to produce a horizontal scroll bar along the bottom of the window or *sbVertical* to produce a vertical scroll bar along the right side of the window. Either may be combined with *sbHandleKeyboard* to allow the scroll bar to respond to arrows and page keys from the keyboard in addition to mouse clicks.

See also: *sbXXXX* scroll bar constants.

write

```
virtual void write( opstream& os);
```

Writes to the output stream *os*.

See also: **TStreamableClass**, **TStreamable**, **opstream** classes

zoom `virtual void zoom();`

Zooms the calling window. This member function is usually called in response to a *cmZoom* command (triggered by a click on the zoom icon). **zoom** takes into account the relative sizes of the calling window and its owner, and the value of *zoomRect*.

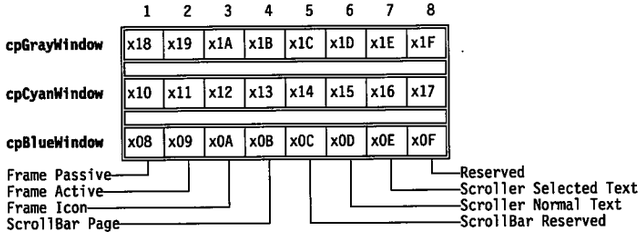
See also: *cmZoom*, *zoomRect*, **TView::locate**

Related functions

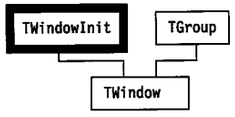
Certain operator functions are related to **TWindow** but are not member functions; see page 232 for more information.

Palette

Window objects use the default palettes *cpBlueWindow* (for text windows), *cpCyanWindow* (for messages), and *cpGrayWindow* (for dialog boxes).



TWindowInit VIEWS.H



TWindow inherits multiply from **TGroup** and the virtual base class **TWindowInit**. The latter provides a constructor and **createFrame** member function used in creating and inserting a frame object. A similar technique is used for **TProgram**, **THistoryWindow**, and **TDeskTop**.

Member functions

constructor `TWindowInit( TFrame *(*cFrame) ( TRect bounds ) );`

This constructor takes a function address argument, usually `&TWindow::initFrame`. The **TWindow** constructor invokes `TGroup(bounds)` and `TWindowInit(&TWindow::initFrame)` to create a window object and associated frame. The latter is inserted in the window group object.

createFrame `TFrame *(*createFrame) ( TRect bounds );`

Called by the **TWindowInit** constructor to create a **TFrame** object with the given bounds and return a pointer to it. A 0 pointer indicates lack of success in this endeavor.

C H A P T E R

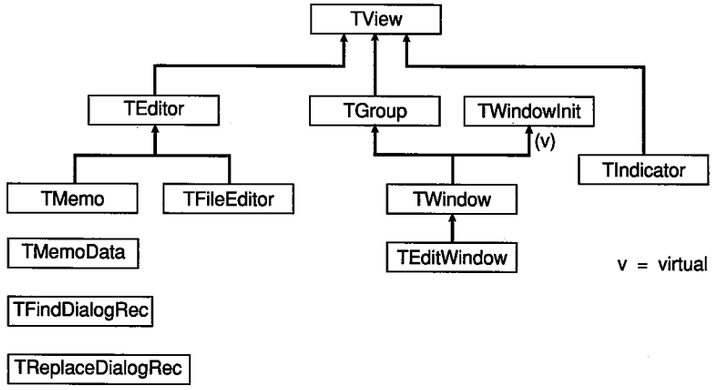
14

The editor classes

TEditor, derived from **TView**, implements a small, fast 64K editor for use in Turbo Vision applications. It features mouse support, undo, clipboard cut, copy and paste, autoindent and overwrite modes, key bindings, and search and replace. You can use this editor for editing files and as a multi-line memo field in dialogs or forms. The following figure presents the overall hierarchy.

Figure 14.1
The editor classes

***TEditor** is a streamable class, inheriting from **TStreamable** by way of **TView**.*



TEditor EDITORS.H

TEditor is the base class for all editors. It implements most of the editor's functionality. If a **TEditor** object is created, it allocates a buffer of the given size out of the heap. The buffer is initially empty.

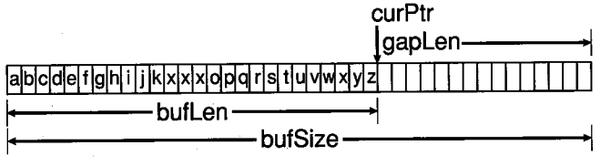
The use of **TEditor** is demonstrated in TVEDIT.CPP for editing files and TVFORMS.CPP as a memo data member. Both of these file can be found in the demos directory of the distribution diskettes.

The TEditor buffer

TEditor implements a *buffer gap editor*. It stores the text in two pieces. Any text before the cursor is stored at the beginning of the buffer, and text after the cursor is stored at the end of the buffer. The space between the text is called the *gap*. When a character is inserted into the editor it is inserted into the gap. The editor supports undo by recording deleted text in the gap and maintaining the the number of characters inserted and deleted. When asked to perform an undo, it deletes the characters that were inserted, copies the deleted characters to the beginning of the gap, and positions the cursor after the formerly-deleted text.

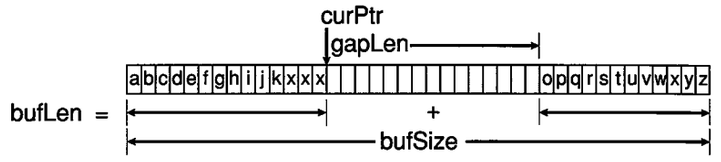
To illustrate how the buffer operates, consider the following diagram of an editor buffer after the characters "abcdefghijkxxxopqrstuvwxyz" are inserted.

Figure 14.2
Buffer after text is inserted



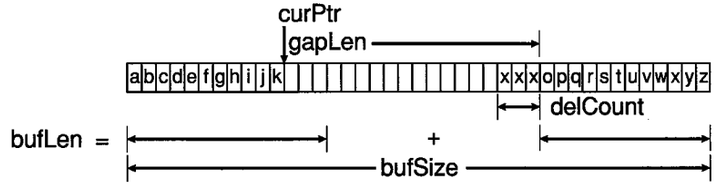
curPtr records the position of the cursor, *gapLen* is the length of the gap, and *bufLen* is the total number of characters in the buffer. *bufSize* is the size of the buffer, which is always the sum of *gapLen* and *bufLen*. If the cursor is then moved to just after the "xxx" characters, the buffer would look like this:

Figure 14.3
Buffer after cursor movement



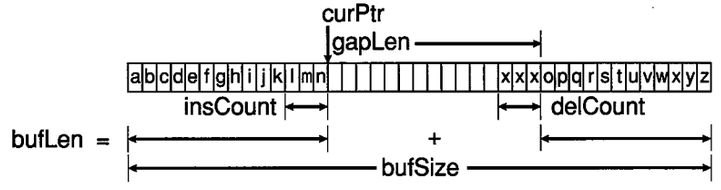
Note that the gap is kept in front of the cursor. This allows for quick insertion of characters without moving any text. If "xxx" is deleted using the backspace key, the characters are copied to the bottom of the gap and the cursor is moved backwards. The *delCount* data member records the number of characters deleted.

Figure 14.4
Buffer after "xxx" is deleted



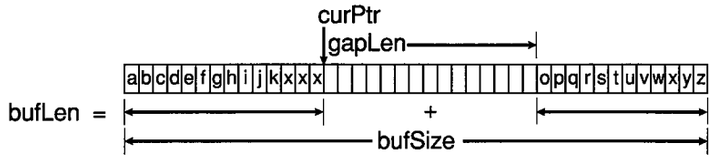
When characters are inserted, the insertion count, *insCount*, is incremented to record how many characters to delete with an undo. If "lmn" are now typed, the buffer would look like this:

Figure 14.5
Buffer after "lmn" is inserted



insCount records the number of characters inserted. If an undo is now requested, the characters "lmn" are deleted and the characters "xxx" are copied on top of them, restoring the buffer to what it was before the edits.

Figure 14.6
Buffer after undo



If the cursor is moved before the undo is performed, all undo information is lost because the gap moves. Undo will only undo operations done between cursor movements. As soon as the cursor moves, the edits performed are considered "accepted." Note also that undo takes space inside the buffer, which could prevent the user from inserting text. The space can be recovered by moving the cursor.

Block selection

The selection or block mark is always either before or after the cursor. If text is inserted into the editor, either through a key press or through **insertText**, the contents of the selection are replaced by the inserted text. If there is no selection, the text is just inserted. The selection is marked by the data members *selStart* and *selEnd*. The selection can be set by the call **setSelect**, which also moves the cursor.

Options

TEditor provides several options, the states of which are stored in Boolean data members. *canUndo* indicates whether the editor records undo information. Since **undo** takes space temporarily from inserts, you might find it advantageous to disable the undo feature. This is done automatically for the clipboard. *overwrite* indicates whether the editor is in overwrite or insert mode. *autoIndent* records whether the editor will, when the *Enter* key is pressed, indent the cursor to the column of the first non-space character of the previous line. This is convenient if the editor is used to edit source code.

Key bindings

Keys are bound to many of the key mappings used in the Borland IDE. The only exceptions are the block commands. Since **TEditor** does not use persistent blocks, the block commands are simulated by copying the information to and from the clipboard. For example, *Ctrl-K Ctrl-B* begins selecting text. *Ctrl-K Ctrl-K* copies the text to the clipboard. *Ctrl-K Ctrl-C* pastes

the contents from the clipboard to the editor. Block selection can be started by holding down the shift key with any of the cursor movement commands instead of using the *Ctrl-K* bindings.

These key bindings can be changed by overriding the **convertEvent** member function which translates the given key event to a command event.

Constants

The constants in this section are defined in EDITORS.H.

maxLineLength is defined to be 256.

Table 14.1
TEditor constants

Item	Value	Item	Value
Command constants:			
cmSave	= 80,	*cmPageDown*	= 509,
cmSaveAs	= 81,	*cmTextStart*	= 510,
cmFind	= 82,	*cmTextEnd*	= 511,
cmReplace	= 83,	*cmNewLine*	= 512,
cmSearchAgain	= 84;	*cmBackSpace*	= 513,
cmCharLeft	= 500	*cmDelChar*	= 514,,
cmCharRight	= 501	*cmDelWord*	= 515,,
cmWordLeft	= 502	*cmDelStart*	= 516,,
cmWordRight	= 503	*cmDelEnd*	= 517,,
cmLineStart	= 504	*cmDelLine*	= 518,,
cmLineEnd	= 505	*cmInsMode*	= 519,,
cmLineUp	= 506	*cmStartSelect*	= 520,,
cmLineDown	= 507	*cmHideSelect*	= 521,,
cmPageUp	= 508	*cmIndentMode*	= 522,,
cmUpdateTitle	= 523;		
TEditor dialog constants:			
edOutOfMemory	= 0,	*edSaveAs*	= 6,
edReadError	= 1,	*edFind*	= 7,
edWriteError	= 2,	*edSearchFailed*	= 8,
edCreateError	= 3,	*edReplace*	= 9,
edSaveModify	= 4,	*edReplacePrompt*	= 10;
edSaveUntitled	= 5,		
TEditor editor flags:			
efCaseSensitive	= 0x0001,	*efReplaceAll*	= 0x0008,
efWholeWordsOnly	= 0x0002,	*efDoReplace*	= 0x0010,
efPromptOnReplace	= 0x0004,	*efBackupFiles*	= 0x0100;
Editor palettes:			
cpIndicator	"\x02\x03"		
cpEditor	"\x06\x07"		
cpMemo	"\x1A\x1B"		

Data members

autoIndent `Boolean autoIndent;`

True if the editor is in autoindent mode.

buffer `char *buffer;`

Pointer to the buffer used to hold the text.

bufLen `ushort bufLen;`

The amount of text stored between the start of the buffer and the current cursor position. See page 416.

bufSize `ushort bufSize;`

Size of the buffer.

canUndo `Boolean canUndo;`

True if the editor is to support **undo**. Otherwise *False*.

See also: **TEditor::undo**

clipboard `static TEditor *clipboard;`

Pointer to the clipboard. Any **TEditor** can be the clipboard; it just needs be assigned to this variable. The clipboard should not support undo (i.e., its *canUndo* should be false).

curPos `TPoint curPos;`

The line/column location of the cursor in the file.

curPtr `ushort curPtr;`

Offset of the cursor.

delCount `ushort delCount;`

Number of characters in the end of the gap that were deleted from the text. Used to implement undo.

delta `TPoint delta;`

The top line and leftmost column shown in the view.

drawLine `int drawLine;`

Column position on the screen where inserted characters are drawn. Used internally by **draw**.

drawPtr `ushort drawPtr;`

Buffer offset corresponding to the current cursor. Used internally by **draw**.

editorDialog `static TEditorDialog editorDialog;`

The **TEditorDialog** data type is a pointer to function returning **ushort** and taking one **int** argument and a variable number of additional arguments. It is defined in EDITORS.H as follows:

```
typedef ushort (*TEditorDialog)( int, ... );
```

editorDialog is a function pointer used by **TEditor** objects to display various dialog boxes. Since dialog boxes are very application-dependent, a **TEditor** object does not display its own dialog boxes directly. Instead it controls them through this function pointer. For an example of an *editorDialog* function, see TVEDIT.CPP.

The various dialog values, passed in the first **int** argument, are self-explanatory:

edOutOfMemory
edReadError
edWriteError
edCreateError
edSaveModify
edSaveUntitled
edSaveAs
edFind
edSearchFailed
edReplace
edReplacePrompt

The default *editorDialog*, *defEditorDialog*, simply returns *cmCancel*.

editorFlags `static ushort editorFlags;`

editorFlags contains various flags for use in the editor:

efCaseSensitive	Default to case-sensitive search
efWholeWordsOnly	Default to whole words only search
efPromptOnReplace	Prompt on replace
efReplaceAll	Replace all occurrences.
efDoReplace	Do replace.
efBackupFiles	Create .BAK files on saves.

The default value is *efBackupFiles* | *efPromptOnReplace*.

findStr `static char findStr[80];`

Stores the last string value used for a find operation.

See also: **TEditor::doSearchReplace**

gapLen `ushort gapLen;`

The size of the "gap" between the text before the cursor and the text after the cursor. See page 416 for an explanation.

hScrollBar `TScrollBar *hScrollBar;`

Pointer to the horizontal scroll bar; 0 if the scroll bar does not exist.

indicator `TIndicator *indicator;`

Pointer to the indicator; 0 if the indicator does not exist.

insCount `ushort insCount;`

Number of characters inserted into the text since the last cursor movement. Used to implement undo.

isValid `Boolean isValid;`

True if the view is valid. Used by the **valid** member function.

See also: **TEditor::valid**

keyState `int keyState;`

Indicates that a special key, such as *Ctrl-K*, has been pressed. Used by **handleEvent** to keep track of "double" control keys such as *Ctrl-K-H* and *Ctrl-K-B*.

limit `TPoint limit;`

The maximum number of columns to display, and the number of lines in the file. Records the limits of the scroll bars.

lockCount `uchar lockCount;`

Holds the lock count semaphore that controls when a view is redrawn. *lockCount* is incremented by **lock** and decremented by **unlock**.

See also: **TGroup::lock**, **TGroup::unlock**, **TGroup::lockFlag**

modified `Boolean modified;`

True if the buffer has been modified.

overwrite `Boolean overwrite;`

True if in overwrite mode; otherwise the editor is in insert mode.

replaceStr `static char replaceStr[80];`

Stores the last string value of a replace operation.

See also: **TEditor::doSearchReplace**

selecting `Boolean selecting;`

True if the editor is in selecting mode (that is, *Ctrl-K Ctrl-B* has been pressed).

selEnd `ushort selEnd;`

The offset of the end of the text selected by *Ctrl-K Ctrl-K*. See page 418.

selStart `ushort selStart;`

The offset of the start of the text selected by *Ctrl-K Ctrl-B*. See page 418.

updateFlags `uchar updateFlags;`

A set of flags indicating the state of the editor. **doUpdate** and other member functions examine these flags to determine whether the view needs to be redrawn.

vScrollBar `TScrollBar *vScrollBar;`

Pointer to the vertical scroll bar; 0 if the scroll bar does not exist.

Member functions

constructor

```
TEditor( const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar
        *aVScrollBar, TIndicator *anIndicator, ushort aBufSize);
```

Calls **TView**(*bounds*) by creating a view with the given bounds. The *hScrollBar*, *vScrollBar*, *indicator*, and *bufSize* data members are set from the given arguments. The scroll bar and indicator arguments can be set to 0 if you do not want these objects. The following default values are set:

canUndo	*True*
selecting	*False*
overwrite	*False*
autoIndent	*False*
lockCount	0
keyState	0
growMode	*gfGrowHiX* | *gfGrowHiY*
options	*ofSelectable*
eventMask	*evMouseDown* | *evKeyDown* | *evCommand* | *evBroadcast*

The buffer is allocated and cleared. If insufficient memory exists, the *edOutOfMemory* dialog box is displayed using *editorDialog*, and *isValid* is set *False*. Otherwise *isValid* is set *True*. The data members associated with

the editor buffer are initialized in the obvious way: *bufLen* to 0, *gapLen* to *bufSize*, *selStart* to 0, *modified* to *False*, and so on.

See also: **TView::Tview**, **TEditor::editorDialog**, **TScrollBar** class.

destructor `virtual ~TEditor();`

Destroys the editor and deletes the buffer.

bufChar `char bufChar( ushort p);`

Returns the *p*'th character in the file, factoring in the gap.

bufPtr `ushort bufPtr( ushort p);`

Returns the offset into *buffer* of the *p*'th character in the file, factoring in the gap.

build `static TStreamable *build();`

Called to create an object in certain stream reading situations.

See also: **TStreamableClass**, **ipstream::readData**

changeBounds `virtual void changeBounds( const TRect& bounds );`

Changes the views bounds, adjusting the delta value and redrawing the scrollbars and view if necessary. Overridden to ensure the file stays within view if the parent size changes.

charPos `int charPos( ushort p, ushort target );`

Calculates and returns the actual cursor position by examining the characters in the buffer between *p* and *target*. Any tab codes encountered are counted as spaces modulo the tab setting.

charPtr `ushort charPtr( ushort p, int target );`

The reverse of **charPos**. Calculates and returns the buffer position corresponding to a cursor position.

checkScrollBar `void checkScrollBar( const TEvent& event, TScrollBar *p, int& d);`

Called by **handleEvent** in response to a *cmScrollBarChanged* broadcast event. If the scroll bar's current *value* is different from *d*, the scroll bar is redrawn.

clipCopy `Boolean clipCopy();`

Returns *False* if this editor has no active clipboard. Otherwise, copies the selected text from the editor to the clipboard using

```
clipboard->insertFrom( this );
```


The selected text is deselected (highlight removed) and the view redrawn. Returns *True* if all goes well.

See also: **TEditor::insertFrom**

clipCut `void clipCut();`

The same as for **clipCopy**, but the selected text is deleted after being copied to the clipboard.

See also: **TEditor::clipCopy**, **TEditor::insertFrom**

clipPaste `void clipPaste();`

The reverse of **clipCopy**: the contents of the clipboard (if any) are copied to the current position of the editor using

```
insertFrom( clipboard );
```

See also: **TEditor::clipCopy**, **TEditor::insertFrom**

convertEvent `virtual void convertEvent( TEvent& ev );`

Used by **handleEvent** to provide basic editing operations by converting various key events to command events. You can change or extend these default key bindings by overriding the **convertEvent** member function. See page 418.

See also: **TEditor::handleEvent**

cursorVisible `Boolean cursorVisible();`

Returns *True* if the cursor (insertion point) is visible within the view.

deleteRange `void deleteRange( ushort startPtr, ushort endPtr, Boolean delSelect);`

If *delSelect* is *True* and a current selection exists, the current selection is deleted; otherwise, the range *startPtr* to *endPtr* is selected and deleted.

See also: **TEditor::deleteSelect**

deleteSelect `void deleteSelect();`

Deletes the selection if one exists. For example, after a successful **clipCopy**, the selected block is deleted.

See also: **TEditor::deleteRange**

doneBuffer `virtual void doneBuffer();`

Deletes the buffer.

doSearchReplace `void doSearchReplace();`

Can be used in both find and find/replace operations, depending on the state of *editorFlags* and user-dialog box interactions. If *edDoReplace* is not set, **doSearchReplace** acts as a simple search for *findStr* with no replacement. Otherwise, this function aims at replacing occurrences of *findStr* with *replaceStr*. In all cases, if the target string not found, an **editorDialog(** *edSearchFailed*) call is invoked. If *efPromptOnReplace* is set in *editorFlags*, an *edReplacePrompt* dialog box appears. Replacement then depends on the user response. If *efRelaceAll* is set, replacement proceeds for all matching strings without prompting until a *cmCancel* command is detected.

See also: **TEditor::findStr, TEditor::replaceStr, TEditor::editorDialog, TEditor::find, TEditor::replace, TEditor::editorFlags**

doUpdate

```
void doUpdate();
```

If *updateFlags* is 0, nothing happens. Otherwise, the view and its scrollbars are updated and redrawn depending on the state of the *updateFlags* bits. For example, if *ufView* is set, the view is redrawn with **drawView**. If the view is *sfActive*, the command set is updated with **updateCommands**. After these updates, *updateFlags* is set to 0.

draw

```
virtual void draw();
```

Overrides **TView::draw** to draw the editor.

drawLines

```
void drawLines( int y, int count, ushort linePtr );
```

Draws *count* copies of the line at *linePtr*, starting at line position *y*.

find

```
void find();
```

Finds occurrences of the existing *findStr* or a new user-supplied string. **find** displays an editor dialog inviting the input of a find string or the acceptance of the existing *findStr*. If a new find string is entered, it will replace the previous *findStr* (unless the user cancels). **find** first creates a **TFindDialogRec** object defined as follows:

```
struct TFindDialogRec
{
  TFindDialogRec( const char *str, ushort flgs )
  {
    strcpy( find, str );
    options = flgs;
  }
  char find[80];
  ushort options;
};
```


The constructor is called with *str* set to the current *findStr*, and *flgs* set to the current *editorFlags*. The *edFind* editor dialog then invites change or acceptance of the *findStr*. Finally, **doSearchReplace** is called for a simple find-no-replace (*efDoReplace* switched off).

See also: **TEditor::findStr, TEditor::replaceStr, TEditor::doSearchReplace, TEditor::editorDialog, TEditor::replace, TEditor::editorFlags**

formatLine

```
void formatLine(void *buff, ushort linePtr, int x, ushort color );
```

Formats the line at *linePtr* in the given *color* and sets result in *buff*. Used by **drawLines**.

getMousePtr

```
ushort getMousePtr( TPoint m);
```

Returns the buffer character pointer corresponding to the point *m* on the screen.

See also: **TEditor::charPtr**

getPalette

```
virtual TPalette& getPalette() const;
```

Returns the default **TEditor** palette, *cpEditor*, defined as "\x06\x07\". Override if you wish to change the palette of the editor.

handleEvent

```
virtual void handleEvent( TEvent& ev );
```

Provides the event handling for the editor. Override if you wish to extend the commands the editor handles. The default handler calls **TView::handleEvent(** *ev*), then converts all relevant editing key events to command events by calling **convertEvent**.

hasSelection

```
Boolean hasSelection();
```

Returns *True* if a selection has been made; that is, if *selStart* does not equal *selEnd*. If these two data members are equal, no selection exists, and *False* is returned.

See also: **TEditor::setSelect**

hideSelect

```
void hideSelect();
```

Sets *selecting* to *False* and hides the current selection with `setSelect( curPtr, CurPtr, false )`.

See also: **TEditor::setSelect, TEditor::selecting**

initBuffer

```
virtual void initBuffer();
```

Allocates a buffer of size *bufSize* and sets *buffer* to point at it.

insertBuffer

```
Boolean insertBuffer( char *p, ushort offset, ushort length, Boolean
                      allowUndo, Boolean selectText);
```

This is the lowest-level text insertion member function. It inserts *length* bytes of text from the array *p* (starting at `p[offset]`) into the buffer (starting at the *curPtr*). If *allowUndo* is set *True*, **insertBuffer** records undo information. If *selectText* is set *True*, the inserted text will be selected. **insertBuffer** returns *True* for a successful operation. Failure invokes a suitable dialog box and returns *False*. **insertBuffer** is used by **InsertFrom** and **InsertText**; you will seldom need to call it directly.

See also: **TEditor::insertFrom, TEditor::insertText**

insertFrom

```
virtual Boolean insertFrom( TEditor *editor );
```

Inserts the current block-marked selection from the argument *editor* into this editor. This member function implements **clipCut**, **clipCopy**, and **clipPaste**. The implementation may help you understand the **insertFrom** and **insertBuffer** functions:

```
Boolean TEditor::insertFrom( TEditor *editor )
{
   return insertBuffer( editor->buffer, editor->bufPtr(editor->selStart),
                        editor->selEnd - editor->selStart, canUndo,
                        isClipboard());
}
```

Note the the *allowUndo* argument is set to the value of the data member *canUndo*. The *selectText* argument will be *True* if there is an active clipboard for this editor.

See also: **TEditor::insertBuffer, TEditor::isClipboard**

insertText

```
Boolean insertText( void *text, ushort length, Boolean selectText );
```

Copies *length* bytes from the given *text* into this object's buffer. If *selectText* is *True*, the inserted text will be selected. This is a simplified version of **InsertBuffer**.

See also: **TEditor::insertBuffer**

isClipboard

```
Boolean isClipboard();
```

Returns *True* if this editor has an attached clipboard; otherwise returns *False*.

See also: **TEditor::clipboard**

lineEnd

```
ushort lineEnd( ushort p);
```

Returns the buffer pointer (offset) of the end of the line containing the given pointer *p*.

See also: **TEditor::lineStart**

lineMove

```
ushort lineMove( ushort p, int count);
```

Moves the line containing the pointer (offset) *p* up or down *count* lines depending on the sign of *count*.

See also: **TEditor::prevLine, TEditor::nextLine**

lineStart

```
ushort lineStart( ushort p);
```

Returns the buffer pointer (offset) of the start of the line contaning the given pointer *p*.

See also: **TEditor::lineEnd**

lock

```
void lock();
```

Increments the semaphore **lockCount**.

See also: **TEditor::lockCount, TEditor::unlock**

newLine

```
void newLine();
```

Inserts a newline (carriage rreturn/line feed pair) at the current pointer. If *autoIndent* is set, appropriate tabs (if needed) are also inserted at the start of the new line.

See also: **TEditor::autoIndent**

nextChar

```
ushort nextChar( ushort p );
```

Returns the buffer offset for the character following the one at the given offset *p*.

nextLine

```
ushort nextLine( ushort p);
```

Returns the buffer offset for the start of the line following the line containing the given offset *p*.

nextWord

```
ushort nextWord( ushort p);
```

Returns the buffer offset for the start of the word following the word containing the given offset *p*.

prevChar

```
ushort prevChar( ushort p);
```

Returns the buffer offset for the character preceding the one at the given offset *p*.

prevLine

```
ushort prevLine( ushort );
```

Returns the buffer offset for the start of the line preceding the line containing the given offset *p*.

prevWord `ushort prevWord( ushort );`

Returns the buffer offset corresponding to the start of the word preceding the word containing the current cursor.

read `virtual void *read( ipstream& is );`

Reads an editor object from the input stream *is*.

See also: **TStreamableClass**, **ipstream**

replace `void replace();`

Replaces occurrences of the existing *findStr* (or a new user-supplied find string) with the existing *replaceStr* (or a new user-supplied replace string). **replace** displays an editor dialog inviting the input of both strings or the acceptance of the existing *findStr* and *replaceStr*. If a new string are entered, they will replace the previous values (unless the user cancels). **replace** first creates a **TReplaceDialogRec** object defined as follows:

```
struct TReplaceDialogRec
{
  TReplaceDialogRec( const char *str, const char *rep, ushort flgs )
  {
    strcpy( find, str );
    strcpy( replace, rep );
    options = flgs;
  }
  char find[80];
  char replace[80];
  ushort options;
};
```

The constructor is called with *str* and *rep* set to the current *findStr* and *replaceStr*, and with *flg* set to the current *editorFlags*. The *edReplace* editor dialog then invites change or acceptance of the two strings. Finally, **doSearchReplace** is called for a find-replace operation (*efDoReplace* switched on).

See also: **TEditor::findStr**, **TEditor::replaceStr**, **TEditor::doSearchReplace**, **TEditor::editorDialog**, **TEditor::find**, **TEditor::editorFlags**

scrollTo `void scrollTo( int x, int y);`

Move column *x* and line *y* to the upper-left corner of the editor.

search `Boolean search( const char *findStr, ushort opts);`

Search for the given string in the editor buffer (starting at *CurPtr*) with the given options. The options are:

efCaseSensitive	Case sensitive search
efWholeWordsOnly	Find whole words only

Returns *True* if a match is found; otherwise returns *False*. If a match is found, the matching text is selected.

setBufLen `void setBufLen( ushort length);`

Sets *bufLen* to *length*, then adjusts *gapLen* and *limit* accordingly. *selStart*, *selEnd*, *curPtr*, *delta.x*, *delta.y*, *drawLine*, *drawPtr*, *delCount*, and *insCount* are all set to 0. *curPos* is set to *delta*, *modified* is set to *False*, and the view is updated and redrawn as needed. The **TEditor** constructor calls **setBufLen**(0). **setBufLen** is also used by **insertBuffer**.

See also: **TEditor::update**, **TEditor::TEditor**, **TEditor** data members, **TEditor::insertBuffer**. See page 416.

setBufSize `virtual Boolean setBufSize( ushort newSize );`

Should be called before changing the buffer size to *newSize*. It should return *True* if the the buffer can be of this new size. By default, it returns *True* if *newSize* is less than or equal to *bufSize*.

setCmdState `void setCmdState( ushort command, Boolean enable );`

Enables or disables the given command depending on whether *enable* is *True* or *False* and whether the editor is *sfActive*. The command is always disabled if the editor is not the selected view. Offers a convenient alter_native to **enableCommands** and **disableCommands**.

setCurPtr `void setCurPtr( ushort p, uchar selectMode );`

Calls **setSelect** and repositions *curPtr* to the offset *p*. Some adjustments may be made, depending on the value of *selectMode*, if *curPtr* is at the beginning or end of a selected text.

See also: **TEditor::setSelect**, **TEditor::curPtr**

setSelect `void setSelect( ushort newStart, ushort newEnd, Boolean curStart);`

Sets the selection to the given offsets into the file, and redraws the view as needed. This member function will either place the cursor in front of or behind the selection, depending on the value (*True* or *False*, respectively) of *curStart*.

setState `virtual void setState( ushort aState, Boolean enable );`

Overrides **TView::setState** to hide and show the indicator and scroll bars. It first calls **TView::setState** to enable and disable commands. If you wish to enable and disable additional commands, override **updateCommands** instead. This is called whenever the command states should be updated.

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

startSelect `void startSelect();`

Called by **handleEvent** when a *Ctrl-K Ctrl-B* selection is detected. Hides the previous selection and sets *selecting* to *True*.

See also: **TEditor::handleEvent, TEditor:hideSelect**

toggleInsMode `void toggleInsMode();`

Toggles the *overwrite* data member from *True* to *False* and from *False* to *True*. Changes the cursor shape by calling **setState**.

See also: **TEditor::overwrite**, *sfCursorIns*

trackCursor `void trackCursor( Boolean center );`

Forces the cursor to be visible. If *center* is *True*, the cursor is forced to be in the center of the screen in the *y* (line) direction. The *x* (column) position is not changed.

undo `void undo();`

Undoes the changes since the last cursor movement.

unlock `void unlock();`

Decrements the data member *lockCount* until it reaches the value 0, at which point a **doUpdate** is triggered. The lock/unlock mechanism prevents over-frequent redrawing of the view.

See also: **TEditor:doUpdate, TEditor::lock, TEditor:lockCount**

update `void update( uchar aFlags );`

Sets *aFlags* in the *updateFlags* data member. If lockCount is 0, calls **doUpdate**.

See also: **TEditor:doUpdate, TEditor::lock, TEditor:lockCount, TEditor::updateFlags**

updateCommands `void updateCommands();`

Called whenever the commands should be updated. This is used to enable and disable commands such as *cmUndo*, *cmClip*, and *cmCopy*.

valid `virtual Boolean valid( ushort command );`

Returns whether the view is valid for the given *command*. By default it returns the value of *isValid*, which is *True* if *buffer* is not 0.

write `virtual void write( opstream& os );`

Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

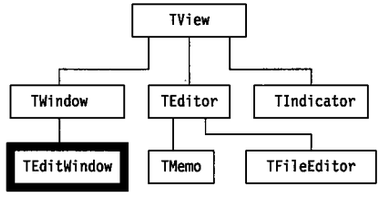
Friends

The function **genRefs** is a friend of **TEditor**.

Related functions

Certain operator functions are related to **TEditor** but are not member functions; see page 232 in Chapter 13 for more information.

TEditWindow EDITORS.H



TEditWindow is a window designed to hold a **TFileEditor** or the clipboard. It will change its title to display the file name being edited and will initialize scroll bars and an indicator for the editor. If the name passed to **TEditWindow** is blank, it assumes that you are initializing the clipboard.

Data members

editor `TFileEditor *editor;`

Pointer to the editor associated with this window.

Member functions

constructor

```
TEditWindow( const TRect& bounds, const char *fileName, int aNumber);
```

Creates a **TEditWindow** object that will edit the given file name with window number *aNumber*. Initializes a framed, tileable window with scroll bars and an indicator. If *fileName* is 0, it is assumed to be an untitled file. If *editor* is equal to *clipboard*, the editor is assumed to be the clipboard.

build

```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

close

```
virtual void close();
```

Overrides **TWindow::close** to hide rather than close the window if the editor is a clipboard.

See also: **TWindow::close**

getTitle

```
virtual const char *getTitle( int );
```

Returns the name of the file being edited, or "Clipboard" if the editor is the clipboard.

handleEvent

```
virtual void handleEvent( TEvent& );
```

Handles *cmUpdateTitle* to redraw the frame of the window. Used in **TFileEditor::saveAs** to change the title of the window if the file being edited changes names.

See also: **TFileEditor::SaveAs**

read

```
virtual void *read(ipstream& os);
```

Reads from the input stream *is*.

See also: **TStreamableClass**, **ipstream**

write

```
virtual void write(opstream& os);
```

Writes to the output stream *os*.

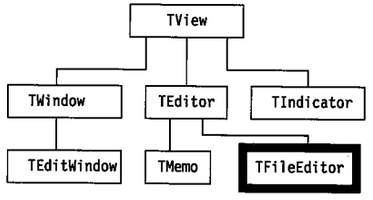
See also: **TStreamableClass**, **opstream**

Related functions

Certain operator functions are related to **TEditWindow** but are not member functions; see page 232 in Chapter 13 for more information.

TFileEditor

EDITORS.H



TFileEditor is a specialized derivative of **TEditor** for editing the contents of a file.

Data members

fileName

```
char fileName[MAXPATH];
```

The name of the file being edited.

Member functions

constructor

```
TFileEditor( const TRect& bounds, TScrollBar *aHScrollBar, TScrollBar
             *aVScrollBar, TIndicator *anIndicator, const char *aFileName
             );
```

Creates a **TFileEditor** object with the given scroll bars and indicator and loads the contents of the given file. If the file is not found or invalid, an error message will be displayed and the object's **valid** member function will return *False*. The *options* variable is set to *sfSelectable* and the *eventMask* set to allow the handling of broadcast events. Any of *aHScrollBar*, *aVScrollBar*, or *anIndicator* arguments can be set to 0 if you do not want them.

build

```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

doneBuffer

```
virtual void doneBuffer();
```

Deletes the buffer.

handleEvent

```
virtual void handleEvent( TEvent& event);
```

Calls **TEditor::handleEvent**, then handles *cmSave* and *cmSaveAs* commands. The *cmSave* command invokes **save**; the *smSaveAs* command invokes **saveAs**.

See also: **TFileEditor::save, TFileEditor::saveAs**

initBuffer

```
virtual void initBuffer();
```

Allocates *bufSize* bytes of memory for the file editor buffer.

loadFile

```
Boolean loadFile();
```

Reads the *fileName* file from disk and checks for errors. Returns *True* if all is well; otherwise returns *False*. Depending on the reason for failure, the *edOutOfMemory* or *edReadError* dialog box is displayed.

See also: **TEditor::editorDialog**

read

```
virtual void *read(ipstream& os);
```

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

save

```
Boolean save();
```

Calls **saveAs** if the file being edited is "Untitled" (that is, no *fileName* is allocated) and returns the return value from **saveAs**. If there is a valid *fileName*, **saveFile** is invoked, and **save** returns the return value of **saveFile**.

See also: **TFileEditor::saveAs, TFileEditor::saveFile**

saveAs

```
Boolean saveAs();
```

Invokes the *edSaveAs* dialog, which prompts for a "save as" file name. If a valid file name is supplied, the current text will be saved with this name using the **saveFile** member function. The file editor's owner is informed of this event via a broadcast *cmUpdateTitle* message. **saveAs** returns *True* if the **saveFile** call is successful, otherwise *False* is returned. *False* is also returned if the *edSaveAs* dialog is cancelled.

See also: **TFileEditor::saveFile, TEditor::editorDialog**

saveFile

```
Boolean saveFile();
```


Saves the *fileName* file to disk. Returns *False* if the save fails; otherwise returns *True*. If *editorFlags* has the *efBackupFiles* bit set, a .BAK file is created. The *edCreateError* or *edWriteError* dialog box will be diaplayed to indicate the reason for failure.

See also: **TEditor::editorDialog, TFileEditor::saveAs, TFileEditor::save**

setBufSize

```
virtual Boolean setBufSize( ushort newSize);
```

Overrides **TEditor::setBufSize** to grow and shrink the buffer. Will grow and shrink the buffer in 4K byte increments. *gapLen* is adjusted appropriately.

updateCommands

```
virtual void updateCommands();
```

Calls **TEditor::updateComands**, then enables the *cmSave* and *cmSaveAs* commands. These commands are only valid if the selected view is an editor, otherwise they should be disabled.

See also: **TEditor:updateCommands**

valid

```
virtual Boolean valid( ushort );
```

Overrides **TEditor::valid** to warn that the file might need saving before the program exits. The *edSaveUntitled* or *edSaveModify* dialogs are displayed as appropriate. Returns *False* if the user cancels the save.

write

```
virtual void write(opstream& os);
```

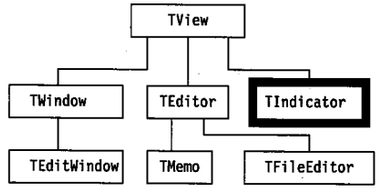
Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

Related functions

Certain operator functions are related to **TFileEditor** but are not member functions; see page 232 in Chapter 13 for more information.

TIndicator — EDITORS.H



TIndicator is the line and column counter in the lower left corner of the edit window. It is initialized by the **TEditWindow** constructor and passed as the fourth argument to the **TEditor** constructor.

Data members

location `TPoint location;`

Stores the location to display. Updated by a **TEditor**.

modified `Boolean modified;`

True if the associated **TEditor** has been modified.

See also: **TIndicator::draw**

Member functions

constructor `TIndicator( const TRect& bounds);`

Creates a **TIndicator** object.

build `static TStreamable *build();`

Called to create an object in certain stream reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws the indicator. If *modified* is *True*, a special character (ASCII value 15) is displayed.

getPalette `virtual TPalette& getPalette() const;`

Returns *cpIndicator* = "\x02\x03" (the **TIndicator** default palette).

read `virtual void *read(ipstream& os);`

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

setState `virtual void setState( ushort aState, Boolean enable );`

Draws the indicator in the frame-dragging color if the view is being dragged.

setValue `void setValue( const TPoint& aLocation, Boolean aModified);`

Method called by **TEditor** to update and display the values of the data members of the associated **TIndicator** object.

write `virtual void write(opstream& os);`

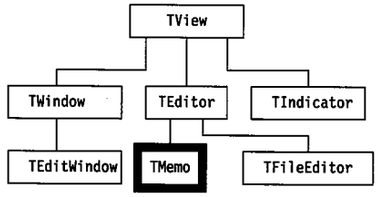
Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

Related functions

Certain operator functions are related to **TIndicator** but are not member functions; see page 232 in Chapter 13 for more information.

TMemo EDITORS.H



TMemo, which is derived from **TEditor**, is designed for insertion into a dialog or form. It supports **dataSize** and allows the *Tab* key to be processed by **TDialog**. It also has a different palette from that of **TEditor**. **dataSize** use the following structure:

```
struct TMemoRec
{
   ushort length;
   char *buffer;
}
```

giving a size of `(bufSize + sizeof( ushort ))`.

Member functions

constructor `TMemo();`

Creates a **TMemo** object via its base constructors.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, ipstream::readData

dataSize `virtual ushort dataSize();`

Used with **setData** and **getData** when saving and restoring **TMemo** objects. Used with **getData** and **setData**, which are inherited from **TView**. By default it returns `(sizeOf( ushort ) + bufSize)`.

See also: **TView::getData, TView::setData**

getPalette `virtual TPalette& getPalette() const;`

Returns *cpMemo* = "\x1A\x1B", the default memo palette.

handleEvent `virtual void handleEvent( TEvent& event);`

Prevents **TMemo** from handling *kbTab* key events; otherwise handles events the same as a **TEditor**.

See also: **TEditor::handleEvent**

read `virtual void *read( ipstream& is );`

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

write `virtual void write( opstream& os );`

Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

Related functions

Certain operator functions are related to **TMemo** but are not member functions; see page 232 in Chapter 13 for more information.

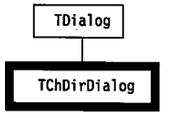
C H A P T E R

15

Implementing standard dialog boxes

This chapter describes the extension to Turbo Vision that provides many standard dialog box capabilities.

TChDirDialog — STDDLG.H



TChDirDialog implements a dialog box labeled "Change Directory". An input line is provided to accept a directory name from the user. A history window and directory list box with vertical scroll bar are available to show recent directory selections and a tree of all available directories. Mouse and keyboard selections can be made in the usual way by highlighting and clicking. The displayed options are:

Directory **n**ame
Directory **t**ree
OK (the default)
Chdir
Revert
Help

TChDirDialog::handleEvent generates the appropriate commands for these selections.

TDirListBox is a friend of **TChDirDialog**, so that the member functions of **TDirListBox** can access the private members of **TChDirDialog**: **setUpDialog**, *dirInput*, *dirList*, *okButton*, and *chDirButton*.

Command constants

The following commands are generated by **TChDirDialog**:

Table 15.1 TChDirDialog commands

Constant	Value	Meaning
cmChangeDir	1005	Generated when Chdir is selected.
cmRevert	1006	Generated when Revert is selected.

Member functions

constructor `TChDirDialog(ushort aOptions, ushort histId);`

Creates a change directory dialog object with the given history ID. The *aOptions* argument can be set as follows:

- *cdNormal*: option to use the dialog immediately.
- *cdNoLoadDir*: option to initialize the dialog without loading the current directory into the dialog. Used if you intend using **setData** to reset the directory or prior to storage on a stream.
- *cdHelpButton*: option to put a help button in the dialog.

The constructor creates and inserts a **TInputLine** object (labeled "Directory ~n~ame"), a **THistory** object, a vertical scroll bar, a **TDirListBox** object (labeled "Directory ~t~ree"), and three buttons ("O~K~", "~C~hdir", "~R~evert"). If *aOptions* has the *cdHelpButton* flag set, a fourth help button is created. Unless the *cdNoLoadDir* option is set, the dialog box is loaded with the current directory.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, ipstream::readData

dataSize `virtual ushort dataSize();`

By default, **dataSize** returns 0. Override to return the size (in bytes) of the data used by **getData** and **setData** to store and retrieve dialog box input data.

See also: **TView::dataSize**, **TView::setData**, **TView::getData**

getData `virtual void getData(void *rec);`

By default, **getData** does nothing. Override to copy **dataSize** bytes from the view to *rec*. Used in combination with **dataSize** and **setData** to store and retrieve dialog box input data.

See also: **TView::dataSize**, **TView::setData**, **TView::getData**

handleEvent `virtual void handleEvent(TEvent& event);`

Calls **TDialog::handleEvent** then processes *cmRevert* (restore previously current directory) and *cmChangeDir* (switch to selected directory) events. The dialog is redrawn if necessary.

See also: **TDialog::handleEvent**

read `virtual void *read(ipstream& os);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **ipstream**

setData `virtual void setData(void *rec);`

By default, **setData** does nothing. Override to copy **dataSize** bytes from *rec* to the view. Used in combination with **dataSize** and **getData** to store and retrieve dialog box input data.

See also: **TView::dataSize**, **TView::setData**, **TView::getData**

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

valid `virtual Boolean valid(ushort command);`

Returns *True* if *command* is other than *cmOk*. If the OK button has been "pressed," **valid** checks the input line and returns *True* for a valid directory. An invalid directory invokes the "Invalid directory" message box and returns *False*.

See also: **TDialog::valid**

write `virtual void write(opstream& os);`

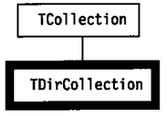
Writes to the output stream *os*.

See also: **TStreamableClass**, **opstream**

Related functions

Certain operator functions are related to **TChDirDialog** but are not member functions; see page 232 in Chapter 13 for more information.

TDirCollection STDDLG.H



TDirCollection is a simple **TCollection** derivative used for storing **TDirEntry** objects. **TDirCollection** is a streamable class, inheriting **TStreamable** from its base class.

Member functions

constructor `TDirCollection(ccIndex aLimit, ccIndex aDelta) : TCollection(aLimit, aDelta);`

Calls the base **TCollection** constructor to create a directory collection with the given *limit* and *delta*.

See also: **TCollection::TCollection**

at `TDirEntry *at(ccIndex index);`

Returns a pointer to the **TDirEntry** object indexed by *index* in this directory collection.

See also: **TNSCollection::at**

atInsert `void atInsert(ccIndex index, TDirEntry *item);`

Inserts the given *item* into the collection at the given *index* and moves the following items down one position. The collection will be expanded by *delta* if the insertion causes the *limit* to be exceeded.

See also: **TNSCollection::atInsert**

atPut `void atPut(ccIndex index, TDirEntry *item);`

Replaces the item at *index* with the given *item*.

See also: **TNSCollection::atPut**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

firstThat `TDirEntry *firstThat(ccTestFunc Test, void *arg);`

This iterator returns a pointer to the first **TDirEntry** object in the collection for which the **Test** function returns *True*. See **TNSCollection::firstThat** for a complete explanation.

See also: **TDirCollection::lastThat**

free `void free(TDirEntry *item);`

Removes (deletes) the given item from the collection and frees the space in the collection.

See also: **TNSCollection::free. TNSCollection::remove**

indexOf `virtual ccIndex indexOf(TDirEntry *item);`

Returns the *index* of the given *item* in this directory collection.

See also: **TNSCollection::indexOf**

insert `virtual ccIndex insert(TDirEntry *item);`

Inserts the *item* into the collection, and adjust the other indexes if necessary. By default, insertions are made at the end of the collection. The index of the inserted item is returned.

see also: **TNSCollection::insert**

lastThat `TDirEntry *lastThat(ccTestFunc Test, void *arg);`

This iterator scans the collection from the last **TDirEntry** object to first. It returns a pointer to the first (that is, the nearest to the end) item in the collection for which the *Test* function returns *True*. See **TNSCollection::lastThat** for a complete explanation.

See also: **TDirCollection::firstThat**

read `virtual void *read(ipstream& os, void * t);`

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

readItem `void *TDirCollection::readItem( ipstream& is );`

Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to read the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, ipstream**

remove

```
void remove(TDirEntry *item);
```

Removes (deletes) the given *item* from this collection. (The space in the collection is not freed).

See also: **TNSCollection::remove, TNSCollection::free**

write

```
virtual void write(opstream& os);
```

Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

writeItem

```
void TDirCollection::writeItem( void *obj, opstream& os );
```

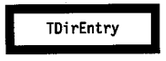
Called for each item in the collection. You'll need to override these in everything derived from **TCollection** or **TSortedCollection** in order to write the items correctly. **TSortedCollection** already overrides this function.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TDirCollection** but are not member functions; see page 232 in Chapter 13 for more information.

TDirEntry — STDDLG.H



TDirEntry is a simple class providing directory paths and descriptions. **TDirEntry** objects are stored in **TDirCollection** objects. **TDirEntry** has two private *char* * data members that can be accessed via the member functions **dir** and **text**.

Member functions

constructor

```
TDirEntry(const char *txt, const char *dir);
inline TDirEntry::TDirEntry(const char *txt, const char *dir) :
       displayText(newStr(txt)), directory(newStr(dir))
```

dir

```
char *dir();
```

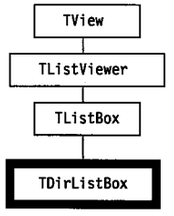
Returns the current directory (the value of the private member *directory*).

text

```
char *text();
```

Returns the current display text (the value of the private member *displayText*).

TDirListBox — STDDLG.H



TDirListBox is a specialized derivative of **TListBox** for displaying and selecting directories stored in a **TDirCollection** object. By default, these are displayed in a single column with an optional vertical scroll bar.

Member functions

constructor

```
TDirListBox(const TRect& bounds, TScrollBar *aScrollBar);
```

Calls `TListBox::TListBox(bounds, 1, aScrollBar)` to create a single-column list box with the given bounds and vertical scroll bar.

See also: **TListBox::TListBox**

destructor

```
~TDirListBox();
```

Calls its base destructors to dispose of the list box.

build

```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

getText

```
virtual void getText(char *dest, short item, short maxLen);
```

Grabs the text string at index *item* and copies it to *dest*.

handleEvent

```
virtual void handleEvent(TEvent& event);
```

Handles double-click mouse events with `putEvent(cmChangeDir)`. This allows a double click to change to the selected directory. Other events are handled by **TListBox::handleEvent**.

See also: **TView::putEvent, TListBox::handleEvent**

isSelected

```
virtual Boolean isSelected(short item);
```

Returns True if *item* is selected, otherwise returns *False*.

list

```
TDirCollection *list();
```

Returns a pointer to the **TDirCollection** object currently associated with this directory list box.

See also: **TSortedListBox::list**

newDirectory

```
void newDirectory(const char *);
```

Deletes the existing **TDirEntry** object associated with this list box and replaces it with the file collection given by *aList*. The first item of the new collection will receive the focus.

See also: **TSortedListBox::newList**

read

```
virtual void *read(ipstream& os);
```

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

setState

```
virtual void setState(ushort aState, Boolean enable);
```

By default, uses the ancestral **TListViewer::setState**.

See also: **TListViewer::setState**

write

```
virtual void write(opstream& os);
```

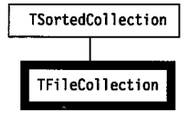
Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

Related functions

Certain operator functions are related to **TDirListBox** but are not member functions; see page 232 in Chapter 13 for more information.

TFileCollection STDDLG.H



TFileCollection is a simple derivative of **TSortedCollection** offering a sorted collection of file names. File names are stored as objects of type **TSearchRec**, defined as follows:

```
struct TSearchRec
{
    uchar attr;
    long time;
    long size;
    char name[MAXFILE+MAXEXT-1];
};
```

The fields of **TSearchRec** have their usual DOS meanings. Drive and directory paths are handled separately with **TDirEntry** objects.

Member functions

constructor

```
TFileCollection(ccIndex aLimit, ccIndex aDelta) :
                TSortedCollection(aLimit, aDelta) {}
```

Calls the base **TSortedCollection** constructor to create a collection with the given *limit* and *delta*.

See also: **TSortedCollection::TSortedCollection**

at

```
TSearchRec *at(ccIndex index)
```

Returns a pointer to the **TSearchRec** object indexed by *index* in this file collection.

See also: **TNSCollection::at**

atInsert

```
void atInsert(ccIndex index, TSearchRec *item)
```

Inserts the **TSearchRec** file referenced by *item* into the collection at the given *index* and moves the following items down one position. The collection will be expanded by *delta* if the insertion causes the *limit* to be exceeded.

See also: **TNSCollection::atInsert**

atPut `void atPut(ccIndex index, TSearchRec *item)`

Replaces the **TSearchRec** file found at *index* with the given *item*.

See also: **TNSCollection::atPut**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

compare `virtual int compare(void *key1, void *key2);` **private**

Performs a standard file string compare and returns a value depending on the results.

- It returns 0 if the file names at *key1* and *key2* are the same.
- It returns > 0
  - if the *key1* name is lexicographically higher than that at *key2*
  - if *key1* name is the directory ".."
  - if *key1* is a directory and *key2* is not a directory
- It returns < 0
  - if the *key1* name is lexicographically lower than that at *key2*
  - if *key2* references the directory ".."
  - if *key2* is a directory and *key1* is not a directory.

See also: **TSortedCollection::compare**

firstThat `TSearchRec *firstThat(ccTestFunc Test, void *arg);`

This iterator returns a pointer to the first **TSearchRec** object in the collection for which the *Test* function returns *True*. See **TNSCollection::firstThat** for a complete explanation.

See also: **TFileCollection::lastThat**

free `void free(TSearchRec *item)`

Removes (deletes) the given **TSearchRec** file *item* from the collection and frees the space in the collection.

See also: **TNSCollection::free. TNSCollection::remove**

indexOf `virtual ccIndex indexOf(TSearchRec *item)`

Returns the *index* of the given **TSearchRec** file *item* in this file collection.

See also: **TNSCollection::indexOf**

insert `virtual ccIndex insert(TSearchRec *item)`

Inserts the **TSearchRec** *item* into the collection, and adjusts the other indexes if necessary. By default, insertions are made at the end of the collection. The index of the inserted item is returned.

See also: **TNSCollection::insert**

lastThat `TSearchRec *lastThat(ccTestFunc Test, void *arg);`

This iterator scans the collection from last item to first. It returns a pointer to the first item (that is, the nearest the end) in the collection for which the *Test* function returns *True*. See **TNSCollection::lastThat** for a complete explanation.

See also: **TFileCollection::firstThat**

read `virtual void *read(ipstream& os);`

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

readItem `void *TFileCollection::readItem( ipstream& is );`

Called for each item in the collection. You'll need to override these in order to read the items correctly.

See also: **TStreamableClass, TStreamable, ipstream**

remove `void remove(TSearchRec *item)`

Removes (deletes) the given **TSearchRec** file *item* from this file collection. (The space in the collection is not freed).

See also: **TNSCollection::remove, TNSCollection::free**

write `virtual void write(opstream& os);`

Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

writeItem `void TFileCollection::writeItem( void *obj, opstream& os );`

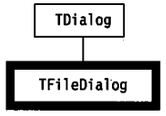
Called for each item in the collection. You'll need to override these in order to write the items correctly.

See also: **TStreamableClass, TStreamable, opstream**

Related functions

Certain operator functions are related to **TFileCollection** but are not member functions; see page 232 in Chapter 13 for more information.

TFileDialog STDDLG.H



TFileDialog implements a file dialog box, history pick list, and input line from which file names (including wildcards) can be input, edited, selected, and opened for editing.

Options

These options perform the specified operations.

Table 15.2 TFileDialog options

Constant	Value	Meaning
fdOKButton	0x0001	Put an OK button in the dialog.
fdOpenButton	0x0002	Put an Open button in the dialog.
fdReplaceButton	0x0004	Put a Replace button in the dialog.
fdClearButton	0x0008	Put a Clear button in the dialog.
fdHelpButton	0x0010	Put a Help button in the dialog.
fdNoLoadDir	0x0100	Do not load the current directory contents into the dialog when intialized. This means you intend to change the wildcard by using **setData** or intend to store the dialog on a stream.

Command constants

The following commands are returned by **TFileDialog**:

Table 15.3 TFileDialog command constants

Constant	Value	Meaning
cmFileOpen	1001	Returned when Open pressed.
cmFileReplace	1002	Returned when Replace pressed.
cmFileClear	1003	Returned when Clear pressed.
cmFileInit	1004	Used by **TFileDialog::valid**.

Data members

directory `const char *directory;`

The current directory.

fileList `TFileList *fileList;`

Pointer to the associated file list.

See also: **TFileList**

fileName `TFileInputLine *fileName;`

Pointer to the associated input line.

See also: **TFileInput**

wildCard `char wildCard[MAXPATH];`

The current drive, path, and file name.

Member functions

constructor

```
TFileDialog(const char *aWildCard, const char *aTitle, const char
            *inputName, ushort aOptions, uchar histId);
```

Creates a fixed-size, framed dialog box with the given title. A **TFileInputLine** object (referenced by the *fileName* data member) is created and inserted, labeled with *inputName* and with its *data* field set to *aWildCard*. The wild card argument is expanded (if necessary) to provide a **TFileList** object, referenced by the *fileList* data member. A **THistory** object is created and inserted with the given history ID. A vertical scroll bar is created and inserted. Depending on the *aOptions* flags, the appropriate buttons are set up (see **TFileDialog** options section). A **TFileInfoPane** object is created and inserted. If the *fdNoLoadDir* flag is not set, the files in the current directory are loaded into the file list.

See also: **TDialog::TDialog, TFileInputLine::TFileInputLine, TFileList::TFileList, THistory::THistory**

destructor `~TFileDialog();`

Deletes *directory*, then destroys the file dialog.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

getFileName `void getFileName(char *s);`

Takes the *fileName->data* field and expands it to a full path format. The result is set in *s*.

handleEvent `virtual void handleEvent(TEvent& event);`

Calls **TDialog::handleEvent**, then handles *cmFileOpen*, *cmFileReplace*, and *cmFileClear* events. These all call **endModal** and pass their commands to the view that opened the file dialog.

See also: **TDialog::handleEvent**, **TView::endModal**

read `virtual void *read(ipstream& os);`

Reads from the input stream *is*.

See also: **TStreamableClass**, **ipstream**

shutDown `virtual void shutDown();`

Used internally by **TObject::destroy** to ensure correct destruction of derived and related objects. **shutDown** is overridden in many classes to ensure the proper setting of related data members when **destroy** is called.

See also: Chapter 6, "Writing safe programs"

valid `virtual Boolean valid(ushort command);`

Returns *True* if *command* is *cmValid*, indicating a successful construction. Otherwise calls **TDialog::valid**. If this returns *True*, the current *fileName* is checked for validity. Valid names will return *True*. Invalid names invoke an "Invalid file name" message box and return *False*.

See also: **TDialog::valid**

write `virtual void write(opstream& os);`

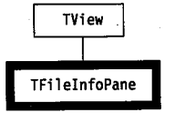
Writes to the output stream *os*.

See also: **TStreamableClass**, **opstream**

Related functions

Certain operator functions are related to **TFileDialog** but are not member functions; see page 232 in Chapter 13 for more information.

TFileInfoPane STDDLG.H



TFileInfoPane implements a simple, streamable view for displaying file information in the owning file dialog box.

Data members

file_block `TSearchRec file_block;`

The file name and attributes for this info pane. **TSearchRec** is defined as follows:

```
struct TSearchRec
{
    uchar attr;
    long time;
    long size;
    char name[MAXFILE+MAXEXT-1];
};
```

where the fields have their obvious DOS file meanings.

Member functions

constructor `TFileInfoPane(const TRect& bounds);`

Calls **TView::TView**(*bounds*) to create a file information pane with the given *bounds*. *evBroadcast* is set in *eventMask*.

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass, ipstream::readData**

draw `virtual void draw();`

Draws the file info pane in the default palette. The block size and date/time stamp are displayed.

getPalette `virtual TPalette& getPalette() const;`

Returns the default palette *cpInfoPane* = "\x1E"

handleEvent `virtual void handleEvent(TEvent& event);`

Calls **TView::handleEvent**, then handles broadcast *cmFileFocused* events (triggered when a new file is focused in a file list) by displaying the file information pane.

See also: **TView::handleEvent, TFileList** commands.

read `virtual void *read(ipstream& os);`

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

write `virtual void write(opstream& os);`

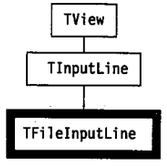
Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

Related functions

Certain operator functions are related to **TFileInfoPane** but are not member functions; see page 232 in Chapter 13 for more information.

TFileInputLine STDDLG.H



TFileInputLine implements a specialized **TInputLine** allowing the input and editing of file names, including optional paths and wild cards. A **TFileInputLine** object is associated with a file dialog box.

Member functions

constructor `TFileInputLine(const TRect& bounds, short aMaxLen);`

Calls `TInputLine::TInputLine(bounds, aMaxLen)` to create a file input line with the given bounds and maximum length. *evBroadcast* is set in the *eventMask*.

See also: **TInputLine::TInputLine**

build `static TStreamable *build();`

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, ipstream::readData

handleEvent `virtual void handleEvent(TEvent& event);`

Calls **TInputLine::handleEvent**, then handles broadcast *cmFileFocused* events by copying the entered file name into the input line's *data* data member and redrawing the view. If the edited name is a directory, the current file name in the owning **TFileDialog** object is appended first.

See also: **TInputLine::handleEvent**

read `virtual void *read(ipstream& os);`

Reads from the input stream *is*.

See also: **TStreamableClass, ipstream**

write `virtual void write(opstream& os);`

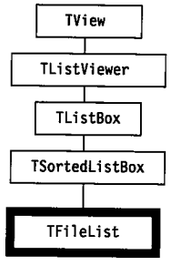
Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

Related functions

Certain operator functions are related to **TFileInputLine** but are not member functions; see page 232 in Chapter 13 for more information.

TFileList STDDLG.H



TFileList implements a sorted two-column list box of file names (held in a **TFileCollection** object). You can select (highlight) a file name by mouse or keyboard cursor actions. An optional vertical scroll bar is available. **TFileList** inherits most of its functionality from **TSortedListBox**.

Command constants

The following commands are broadcast by **TFileList**:

Table 15.4 TFileList commands

Constant	Value	Meaning
cmFileFocused	102	A new file was focused in the **TFileList** object.
cmFileDoubleClicked	103	A file was selected in the **TFileList** object.

Member functions

constructor

```
TFileList(const TRect& bounds, const char *aWildCard, TScrollBar
        *aScrollBar);
```

Calls the **TSortedListBox** constructor to create a two-column **TFileList** object with the given bounds and, if *aScrollBar* is non-zero, a vertical scrollbar. The *aWildCard* argument is a string giving drive, path, file, and extension (the standard DOS wild cards are expanded if present) from which the file collection is generated by calling **readDirectory**.

destructor

```
~TFileList();
```

Deletes the file list.

build

```
static TStreamable *build();
```

Called to create an object in certain stream-reading situations.

See also: **TStreamableClass**, **ipstream::readData**

focusItem

```
virtual void focusItem(short item);
```

Focuses the given item in the list. Calls **TSortedListBox::focusItem** and broadcasts a *cmFileFocused* event.

See also: **TSortedListBox::focusItem**

getText

```
virtual void getText(char *dest, short item, short maxLen);
```

Grabs the **TSearchRec** object at *item* and sets the file name in *dest*. "\\" is appended if the name is a directory.

handleEvent

```
virtual void handleEvent(TEvent& event);
```

Inherits the **TSortedListBox** event handler unmodified. This handles all the normal list box events: selecting and highlighting a file name by mouse or keyboard cursor movement.

See also: **TSortedListBox::handleEvent**, **TListBox::handleEvent**

list

```
TFileCollection *list();
```

Returns the private *items* data member, a pointer to the **TFileCollection** object currently associated with this file list box.

See also: **TSortedListBox::list**, **TListBox::items**

newList

```
void newList(TFileCollection *aList);
```

Calls **TSortedListBox::newList** to delete the existing **TFileCollection** object associated with this list box and replace it with the file collection given by *aList*. The first item of the new collection will receive the focus.

See also: **TSortedListBox::newList**

read

```
virtual void *read(ipstream& os);
```

Reads from the input stream *is*.

See also: **TStreamableClass**, **ipstream**

readDirectory

```
void readDirectory(const char *dir, const char *wildCard);
void readDirectory(const char *wildCard);
```

Expands the *wildCard* string to generate the file collection associated with this file list. The first form allows the separate submission of a relative or absolute path in the *dir* argument. Either '/' or '\' can be used as subdirectory separators (but '/' is converted to '\' for output). The resulting **TFileCollection** object (a sorted set of **TSearchRec** objects) is assigned to the private *items* data member (accessible via the **list** member function). If too many files are generated, a warning message box appears.

readDirectory knows about file attributes and will not generate hidden file names.

See also: **TFileList::TFileList**

write

```
virtual void write(opstream& os);
```

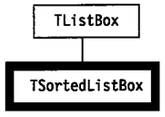
Writes to the output stream *os*.

See also: **TStreamableClass, opstream**

Related functions

Certain operator functions are related to **TFileList** but are not member functions; see page 232 in Chapter 13 for more information.

TSortedListBox STDDLG.H



TSortedListBox is a specialized **TListBox** derivative that maintains its items in a sorted sequence. It is intended as a base for other list box classes, such as **TFileList**.

Member functions

constructor

```
TSortedListBox(const TRect& bounds, ushort aNumCols, TScrollBar
               *aScrollBar);
```

Calls **TListBox::TListBox**(*bounds, aNumCols, aScrollBar*) to create a list box with the given size, number of columns, and vertical scroll bar. *shiftState* is set to 0 and the cursor set at the first item.

getKey

```
virtual void *getKey(const char *s);
```

private

You must define this private member function in all derived classes to provide a means of returning the key for the given string *s*. This will depend on the sorting strategy adopted in your derived class. By default, **getKey** returns *s*.

handleEvent

```
virtual void handleEvent(TEvent& event);
```


Calls **TListBox::handleEvent**, then handles the special key and mouse events used to select items from the list.

See also: **TListBox::handleEvent**

list

```
TSortedCollection *list();
```

Returns a pointer to the **TSortedCollection** object currently associated with this sorted list box. This gives access the the private *items* data member, a pointer to the items to be listed and selected. Derived sorted list box classes will typically override **list** to provide a pointer to objects of a class derived from **TSortedCollection**. For example, **TFileList::list** returns a pointer to the **TFileCollection** object specified by **TFileList::items**.

See also: **TSortedListBox::list, TListBox::list, TFileList::list**

newList

```
void newList(TSortedCollection *aList);
```

Calls **TListBox::newList** to delete the existing **TSortedCollection** object associated with this list box and replace it with the collection given by *aList*. The first item of the new collection will receive the focus.

See also: **TListBox::newList**

C H A P T E R

16

Global reference

The standard classes are described in Chapter 13; the editor and dialog classes are described in Chapters 14 and 15, respectively.

This chapter describes all the elements of Turbo Vision that are not part of the Turbo Vision standard class hierarchy. The elements listed in this chapter include typedefs, enumerations, constants, variables, and non-member (global) functions defined in the Turbo Vision header files. A typical entry looks like this:

sample function — SAMPLE.H

Declaration `void sample(int arg);`

Action **sample** takes an **int** argument and performs some useful function.

See also related classes, functions, and so on

ap*XXXX* constants — APP.H

Values The following application palette constants are defined:

Table 16.1 Application palette constants

Constant	Value	Meaning
apColor	0	Use palette for color screen
apBlackWhite	1	Use palette for LCD screen
apMonochrome	2	Use palette for monochrome screen

Action Constants beginning with *ap* designate which of three standard color palettes a Turbo Vision application should use. The three palettes are used for color, black and white, and monochrome displays.

Boolean enumeration — TTYPES.H

Declaration `enum Boolean { False, True };`

Action Assigns the **int** values 0 to *False* and 1 to *True*. Note that the tests `if ( tesfunc() ) {...}` and `if ( True == testfunc() ) {...}` may not be equivalent.

BUILDER typedef — TOBJSTRM.H

Declaration `typedef TStreamable*(*BUILDER)();`

Action Each streamable class has a builder function of type **BUILDER**. The builder provides raw memory of the correct size and initializes the vtable pointers when objects are created by certain stream read operations. The **read** function of the streamable class reads data from the stream into the raw object provided by the builder.

See also **TView::build**, **TStreamable**

bfXXXX constants — DIALOGS.H

Values The following button flags are defined:

Table 16.2 Button flags

Constant	Value	Meaning
bfNormal	0x00	Button is a normal button
bfDefault	0x01	Button is the default button
bfLeftJust	0x02	Button label is left-justified

Action A combination of these values is passed in the *aFlags* argument to the **TButton** constructor to determine the newly created button's style. *bfNormal* indicates a normal, non-default button. *bfDefault* indicates that the button will be the default button. It is the responsibility of the programmer to ensure that there is only one default button in a **TGroup**. The *bfLeftJust* value can be added to *bfNormal* or *bfDefault* and affects the position of the text displayed within the button: If clear, the label is centered; if set, the label is left-justified.

See also *TButton::flags*, **TButton::makeDefault**, **TButton::draw**

ccAppFunc typedef — TTYPES.H

Declaration `typedef void (*ccAppFunc)( void *item, void *arg );`

Action Used in iterator functions to provide an action function and argument list to be applied to a range of items in a collection.

See also **TNSCollection::forEach**, *ccTestFunc*

ccIndex typedef — TTYPES.H

Declaration `typedef int ccIndex;`

Action Used to index and count the items in a collection.

See also **TNSCollection::at**

ccNotFound constant — TTYPES.H

Declaration `const ccNotFound = -1;`

Action The *ccIndex* value returned by various collection-search functions if the search fails.

See also **TNSCollection::indexOf**

ccTestFunc typedef — TTYPES.H

Declaration `typedef Boolean (*ccTestFunc)( void *item, void *arg );`

Action Used in iterator functions to provide a test function and argument list to be applied to a range of items in a collection.

See also **TNSCollection::firstThat**, **TNSCollection::lastThat**, *ccAppFunc*

cm*XXXX* constants VIEWS.H

Action These constants represent Turbo Vision's predefined *commands*. They are passed in the **TEvent::command** data member of *evMessage* events (*evCommand* and *evBroadcast*), and cause the **handleEvent** methods of Turbo Vision's standard objects to perform various tasks.

Turbo Vision reserves constant values 0 through 99 and 256 through 999 for its own use. Standard Turbo Vision objects' event handlers respond to these predefined constants. You can define your own constants in the ranges 100 through 255 and 1,000 through 65,535 without conflicting with predefined commands.

Values The standard commands in Table 16.3 are defined by Turbo Vision and used by standard Turbo Vision objects.

Table 16.3: Standard command codes

Command	Value	Meaning
cmValid	0	Passed to **TView::valid** to check the validity of a newly instantiated view.
cmQuit	1	Causes **TProgram::handleEvent** to call **endModal(cmQuit)**, terminating the application. The status line or one of the menus typically contains an entry that maps *kbAltX* to *cmQuit*.
cmError	2	Never handled by any object. May be used to represent unimplemented or unsupported commands.
cmMenu	3	Causes **TMenuView::handleEvent** to call **execView** on itself to perform a menu selection process, the result of which may generate a new command through **putEvent**. The status line typically contains an entry that maps *kbF10* to *cmMenu*.
cmClose	4	Handled by **TWindow::handleEvent** if the *infoPtr* data member of the event record is 0 or points to the window. If the window is modal (such as a modal dialog), an *evCommand* with a value of *cmCancel* is generated through **putEvent**. If the window is modeless, the window's **close** member function is called if the window supports closing (see *wfClose* flag). A click in a window's close box generates an *evCommand* event with a *command* of *cmClose* and an *infoPtr* that points to the window. The status line or one of the menus typically contains an entry that maps *kbAltF3* to *cmClose*.
cmZoom	5	Causes **TWindow::handleEvent** to call **TWindow::zoom** on itself if the window supports zooming (see *wfZoom* flag) and if the *infoPtr* data member of the event record is 0 or points to the window. A click in a window's zoom box or a double-click on a window's title bar generates an *evCommand* event with a *command* of *cmZoom* and an *infoPtr* that points to the window. The status line or one of the menus typically contains an entry that maps *kbF5* to *cmZoom*.
cmResize	6	Causes **TWindow::handleEvent** to call **TView::dragView** on itself if the window supports resizing (see *wfMove* and *wfGrow* flags). The status line or one of the menus typically contains an entry that maps *kbCtrlF5* to *cmResize*.

Table 16.3: Standard command codes (continued)

cmNext	7	Causes **TDeskTop::handleEvent** to move the last window on the desktop in front of all other windows. The status line or one of the menus typically contains an entry that maps *kbF6* to *cmNext*.
cmPrev	8	Causes **TDeskTop::handleEvent** to move the first window on the desktop behind all other windows. The status line or one of the menus typically contains an entry that maps *kbShiftF6* to *cmPrev*.

The standard commands listed in Table 16.4 define the default behavior of dialog box objects.

Table 16.4: Dialog box standard commands

Command	Value	Meaning
cmOk	10	OK button was pressed
cmCancel	11	Dialog box was canceled by Cancel button, close icon, or *Esc* key
cmYes	12	Yes button was pressed
cmNo	13	No button was pressed
cmDefault	14	Default button was pressed

An event with one of the commands *cmOk*, *cmCancel*, *cmYes*, or *cmNo* causes a modal dialog's **TDialog::handleEvent** to terminate the dialog and return that value (by calling **endModal**). A modal dialog typically contains at least one **TButton** with one of these command values. **TDialog::handleEvent** will generate a *cmCancel* command event in response to a *kbEsc* keyboard event.

The *cmDefault* command causes the **TButton::handleEvent** of a default button (see *bfDefault* flag) to simulate a button press. **TDialog::handleEvent** will generate a *cmDefault* command event in response to a *kbEnter* keyboard event.

The standard commands listed in Table 16.5 are defined for use by standard views and editor applications.

Table 16.5: Standard view commands

Command	Value	Meaning
cmCut	20	Editor command codes
cmCopy	21	
cmPaste	22	
cmUndo	23	
cmClear	24	
cmTile	25	
cmCascade	26	
cmReceivedFocus	50	**TView::setState** uses the **message** function to send an *evBroadcast*

Table 16.5: Standard view commands (continued)

cmReleasedFocus	51	event with one of these values to its **TView::owner** whenever *sfFocused* is changed. The *infoPtr* of the event points to the view itself. This in effect informs any peer views that the view has received or released focus, and that they should update themselves appropriately. **TLabel** objects, for example, respond to these commands by highlighting or unhighlighting themselves when the peer view they label is focused or unfocused.
cmCommandSetChanged	52	The **TProgram::idle** member function generates an *evBroadcast* event with this value whenever it detects a change in the current command set (as caused by a call to **TView's enableCommands**, **disableCommands**, or **setCommands** methods). The *cmCommandSetChanged* broadcast is sent to the handleEvent of *every* view in the physical hierarchy (unless their *eventMask* specifically masks out *evBroadcast* events). If a view's appearance is affected by command set changes, it should react to *cmCommandSetChanged* by redrawing itself. **TButton**, **TMenuView**, and **TStatusLine** objects, for example, react to this command by redrawing themselves.
cmScrollBarChanged *cmScrollBarClicked*	53 54	A **TScrollBar** uses the **message** function to send an *evBroadcast* event with one of these values to its owner whenever its value changes or whenever the mouse is clicked on the scroll bar. The *infoPtr* of the event points to the scroll bar itself. Such broadcasts are reacted upon by any peer views controlled by the scroll bar, such as **TScroller** and **TListViewer** objects.
cmSelectWindowNum	55	Causes **TWindow::handleEvent** to call **TView::select** on itself if the *infoInt* of the event record corresponds to *TWindow::number*. **TProgram::handleEvent** responds to *Alt-1* through *Alt-9* keyboard events by broadcasting a *cmSelectWindowNum* event with an *infoInt* of 1 through 9.
cmListItemSelected	56	**TListViewer** message that an item has been selected.
cmRecordHistory	60	Causes a **THistory** object to "record" the current contents of the **TInputLine** object it controls. A **TButton** sends a *cmRecordHistory* broadcast to its owner when it is pressed, in effect causing all **THistory** objects in a dialog to "record" at that time.

See also **TView::handleEvent, TCommandSet**

cstrlen function — UTIL.H

Declaration `int cstrlen( const char *s );`

Action Returns the length of string *s*, where *s* is a control string using tilde characters ('~') to designate hot keys. The tildes are excluded from the length of the string, as they will not appear on the screen. For example, given the argument "~B~roccoli", **cstrlen** returns 8.

See also **moveCStr**

ctrlToArrow function — UTIL.H

Declaration `ushort ctrlToArrow( ushort keyCode);`

Action Converts a WordStar-compatible control key code to the corresponding cursor key code. If the low byte of *keyCode* matches one of the control key values in Table 16.6, the result is the corresponding *kbXXXX* constant. Otherwise, *keyCode* is returned unchanged.

Table 16.6 Control-key mappings

Keystroke	Lo(keyCode)	Result
Ctrl-A	0x01	*kbHome*
Ctrl-D	0x04	*kbRight*
Ctrl-E	0x05	*kbUp*
Ctrl-F	0x06	*kbEnd*
Ctrl-G	0x07	*kbDel*
Ctrl-S	0x13	*kbLeft*
Ctrl-V	0x16	*kbIns*
Ctrl-X	0x18	*kbDown*

DEFAULT_SAFETY_POOL_SIZE — BUFFERS.H

Declaration `const DEFAULT_SAFETY_POOL_SIZE = 4096;`

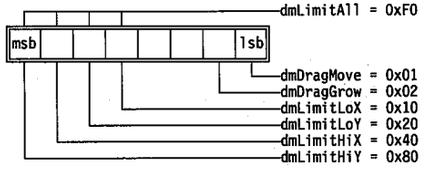
Action Gives the initial default safety pool size in bytes. You can change this value by editing the declaration. If you call **TVMemMgr::resizeSafetyPool** with no size argument, this default value is assumed.

See also **TVMemMgr::resizeSafetyPool**

dm*XXXX* constants — VIEWS.H

Values The *dragMode* bits are defined as follows:

Figure 16.1 *dragMode* bit flags



Action The drag mode constants are used to compose the *mode* argument of the **TView::dragView** method. They specify whether the view is allowed to move and/or change size, and how to interpret the *limits* argument.

The drag mode constants are defined as follows:

Table 16.7 Drag mode constants

Constant	Meaning
dmDragMove	Allow the view to move.
dmDragGrow	Allow the view to change size.
dmLimitLoX	The view's left-hand side cannot move outside *limits*.
dmLimitLoY	The view's top side cannot move outside *limits*.
dmLimitHiX	The view's right-hand side cannot move outside *limits*.
dmLimitHiY	The view's bottom side cannot move outside *limits*.
dmLimitAll	No part of the view can move outside *limits*.

The *dragMode* data member of a **TView** object may contain any combination of the *dmLimitXX* flags; by default, the **TView** constructor sets the data member to *dmLimitLoY*. Currently, the *dragMode* data member is used only in **TWindow::dragView** when a window is moved or resized.

EOS constant — TTYPES.H

Declaration
```
const char EOS = '\0';
```

Action A synonym for the end-of-string null character.

eventQSize constant — CONFIG.H

Declaration
```
const eventQSize = 16;
```

Action Specifies the size of the event queue.

ev*XXXX* constants — SYSTEM.H

Action These mnemonics indicate types of events to Turbo Vision event handlers. ev*XXXX* constants are used in several places: In the *event* data member of an event structure, in the *eventMask* data member of a view object, and in the *positionalEvents* and *focusedEvents* variables.

Values The following event flag values designate standard event types:

Table 16.8 Standard event flags

Constant	Value	Meaning
evMouseDown	0x0001	Mouse button depressed
evMouseUp	0x0002	Mouse button released
evMouseMove	0x0004	Mouse changed location
evMouseAuto	0x0008	Periodic event while mouse button held down
evKeyDown	0x0010	Key pressed
evCommand	0x0100	Command event
evBroadcast	0x0200	Broadcast event

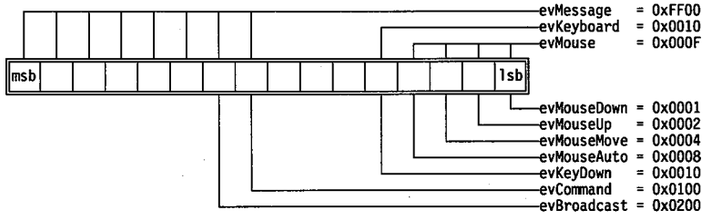
The following constants can be used to mask types of events:

Table 16.9 Standard event masks

Constant	Value	Meaning
evNothing	0x0000	Event already handled
evMouse	0x000F	Mouse event
evKeyboard	0x0010	Keyboard event
evMessage	0xFF00	Message (command, broadcast, or user-defined) event

The event mask bits are defined as follows:

Figure 16.2 Event mask bit mapping



The standard event masks can be used to determine whether an event belongs to a particular "family" of events. For example,

```
if ( (event.what & evMouse) != 0 ) doMouseEvent();
```

See also **TEvent, TView::eventMask, getKeyEvent, getMouseEvent, handleEvent,** *positionalEvents, focusedEvents*

focusedEvents constant — VIEWS.H

Declaration
```
focusedEvents = evKeyboard | evCommand;
```

Action Defines the event classes that are *focused events*. The *focusedEvents* and *positionalEvents* values are used by **TGroup::handleEvent** to determine how to dispatch an event to the group's subviews. If an event class isn't

contained in *focusedEvents* or *positionalEvents*, it is treated as a broadcast event.

See also *positionalEvents*, **TGroup::handleEvent**, **TEvent**, *evXXXX*

genRefs function — GENINC.H

Declaration
```
void genRefs();
```

genRefs is used by GENINC.CPP to generate assembler offsets for various class data members. GENINC.EXE creates the TVWRITE.INC file needed to build TV.LIB. **genRefs** is a friend function of **TDrawBuffer**, **TEditor**, **TEventQueue**, **TTerminal**, **TView**, and **TGroup**. It will be of interest only to advanced users who want to develop their own Turbo Vision libraries.

getAltChar function — UTIL.H

Declaration
```
char getAltChar( ushort keyCode);
```

Action Returns the character *ch* for which *Alt-ch* produces the 2-byte scan code given by the argument *keyCode*. This function gives the reverse mapping to **getAltCode**.

See also **getAltCode**

getAltCode function — UTIL.H

Declaration
```
ushort getAltCode( char ch );
```

Action Returns the 2-byte scan code (key code) corresponding to *Alt-ch*. This function gives the reverse mapping to **getAltChar**.

See also **getAltChar**

gfXXXX constants — VIEWS.H

Action These mnemonics set the *growMode* data member in all **TView** and derived objects. The bits set in *growMode* determine how the view will grow in relation to changes in its owner's size.

Values The *growMode* bits are defined as follows:

Figure 16.3 *growMode* bit mapping

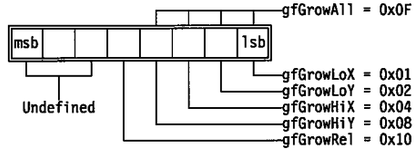


Table 16.10 Grow mode flag definitions

Note that LoX = left side; LoY = top side; HiX = right side; HiY = bottom side.

Constant	Meaning
gfGrowLoX	If set, the left-hand side of the view will maintain a constant distance from its owner's right-hand side. If not set, the movement indicated won't occur.
gfGrowLoY	If set, the top of the view will maintain a constant distance from the bottom of its owner.
gfGrowHiX	If set, the right-hand side of the view will maintain a constant distance from its owner's right side.
gfGrowHiY	If set, the bottom of the view will maintain a constant distance from the bottom of its owner's.
gfGrowAll	If set, the view will move with the lower-right corner of its owner.
gfGrowRel	For use with **TWindow** objects that are in the desktop. The view will change size relative to the owner's size, maintaining that relative size with respect to the owner even when switching between 25 and 43/50 line modes.

See also *TView::growMode*

hcXXXX constants — VIEWS.H

Values The following help context constants are defined:

Table 16.11 Help context constants

Constant	Value	Meaning
hcNoContext	0	No context specified
hcDragging	1	Object is being dragged

Action The default value of *TView::helpCtx* is *hcNoContext*, which indicates that there is no help context for the view. **TView::getHelpCtx** returns *hcDragging* whenever the view is being dragged (as indicated by the *sfDragging* state flag).

Turbo Vision reserves help context values 0 through 999 for its own use. Programmers may define their own constants in the range 1,000 to 65,535.

See also **TView::getHelpCtx, TStatusLine::update**

historyAdd function — UTIL.H

Declaration
```
void historyAdd( uchar id, const char *str);
```

Action Adds the string *str* to the history list indicated by *id*.

historyCount function — UTIL.H

Declaration
```
ushort historyCount( uchar id );
```

Action Returns the number of strings in the history list corresponding to ID number *id*.

historyStr function — UTIL.H

Declaration
```
const char *historyStr( uchar id, int index);
```

Action Returns the *index*'th string in the history list corresponding to ID number *id*.

inputBox function — MSGBOX.H

Declaration
```
ushort inputBox( const char *title, const char *aLabel, char *s, uchar
                 limit );
```

Action Displays an input box with the given title and label. Accepts input to string *s* with a maximum of *limit* characters.

See also **inputBoxRect**

inputBoxRect function — MSGBOX.H

Declaration
```
ushort inputBoxRect( const TRect &bounds, const char *title, const char
                     *aLabel, char *s, uchar limit );
```

Action Displays an input box with the given bounds, title, and label. Accepts input to string *s* with a maximum of *limit* characters.

See also **inputBox**

kb*XXXX* constants — TKEYS.H

Values The following values define keyboard states, and can be used when examining the keyboard shift state, which is stored in a byte at absolute address 0x40:0x17.

Table 16.12 Keyboard state and shift masks

Constant	Value	Meaning
kbRightShift	0x0001	Set if the Right Shift key is currently down
kbLeftShift	0x0002	Set if the Left Shift key is currently down
kbCtrlShift	0x0004	Set if the Ctrl key is currently down
kbAltShift	0x0008	Set if the Alt key is currently down
kbScrollState	0x0010	Set if the keyboard is in the Scroll Lock state
kbNumState	0x0020	Set if the keyboard is in the Num Lock state
kbCapsState	0x0040	Set if the keyboard is in the Caps Lock state
kbInsState	0x0080	Set if the keyboard is in the Ins Lock state

The following values define keyboard scan codes and can be used when examining the **TEvent::keyCode** data member of an *evKeyDown* event record:

Table 16.13 Alt-letter key codes

Constant	Value	Constant	Value
kbAltA	0x1E00	*kbAltN*	0x3100
kbAltB	0x3000	*kbAltO*	0x1800
kbAltC	0x2E00	*kbAltP*	0x1900
kbAltD	0x2000	*kbAltQ*	0x1000
kbAltE	0x1200	*kbAltR*	0x1300
kbAltF	0x2100	*kbAltS*	0x1F00
kbAltG	0x2200	*kbAltT*	0x1400
kbAltH	0x2300	*kbAltU*	0x1600
kbAltI	0x1700	*kbAltV*	0x2F00
kbAltJ	0x2400	*kbAltW*	0x1100
kbAltK	0x2500	*kbAltX*	0x2D00

Table 16.13: Alt-letter key codes (continued)

kbAltL	0x2600	*kbAltY*	0x1500
kbAltM	0x3200	*kbAltZ*	0x2C00

Table 16.14 Special key codes

Constant	Value	Constant	Value
kbAltEqual	0x8300	*kbEnd*	0x4F00
kbAltMinus	0x8200	*kbEnter*	0x1C0D
kbAltSpace	0x0200	*kbEsc*	0x011B
kbBack	0x0E08	*kbGrayMinus*	0x4A2D
kbCtrlBack	0x0E7F	*kbHome*	0x4700
kbCtrlDel	0x0600	*kbIns*	0x5200
kbCtrlEnd	0x7500	*kbLeft*	0x4B00
kbCtrlEnter	0x1C0A	*kbNoKey*	0x0000
kbCtrlHome	0x7700	*kbPgDn*	0x5100
kbCtrlIns	0x0400	*kbPgUp*	0x4900
kbCtrlLeft	0x7300	*kbrayPlus*	0x4E2B
kbCtrlPgDn	0x7600	*kbRight*	0x4D00
kbCtrlPgUp	0x8400	*kbShiftDel*	0x0700
kbCtrlPrtSc	0x7200	*kbShiftIns*	0x0500
kbCtrlRight	0x7400	*kbShiftTab*	0x0F00
kbDel	0x5300	*kbTab*	0x0F09
kbDown	0x5000	*kbUp*	0x4800

Table 16.15 Alt-number key codes

Constant	Value	Constant	Value
kbAlt1	0x7800	*kbAlt6*	0x7D00
kbAlt2	0x7900	*kbAlt7*	0x7E00
kbAlt3	0x7A00	*kbAlt8*	0x7F00
kbAlt4	0x7B00	*kbAlt9*	0x8000
kbAlt5	0x7C00	*kbAlt0*	0x8100

Table 16.16 Function key codes

Constant	Value	Constant	Value
kbF1	0x3B00	*kbF6*	0x4000
kbF2	0x3C00	*kbF7*	0x4100
kbF3	0x3D00	*kbF8*	0x4200
kbF4	0x3E00	*kbF9*	0x4300
kbF5	0x3F00	*kbF10*	0x4400

Table 16.17 Shift-function key codes

Constant	Value	Constant	Value
kbShiftF1	0x5400	*kbShiftF6*	0x5900
kbShiftF2	0x5500	*kbShiftF7*	0x5A00
kbShiftF3	0x5600	*kbShiftF8*	0x5B00
kbShiftF4	0x5700	*kbShiftF9*	0x5C00
kbShiftF5	0x5800	*kbShiftF10*	0x5D00

Table 16.18 Control-function key codes

Constant	Value	Constant	Value
kbCtrlF1	0x5E00	*kbCtrlF6*	0x6300
kbCtrlF2	0x5F00	*kbCtrlF7*	0x6400
kbCtrlF3	0x6000	*kbCtrlF8*	0x6500
kbCtrlF4	0x6100	*kbCtrlF9*	0x6600
kbCtrlF5	0x6200	*kbCtrlF10*	0x6700

Table 16.19 Alt-function key codes

Constant	Value	Constant	Value
kbAltF1	0x6800	*kbAltF6*	0x6D00
kbAltF2	0x6900	*kbAltF7*	0x6E00
kbAltF3	0x6A00	*kbAltF8*	0x6F00
kbAltF4	0x6B00	*kbAltF9*	0x7000
kbAltF5	0x6C00	*kbAltF10*	0x7100

See also *evKeyDown*, **getKeyEvent**

K Globals

lowMemory function — UTIL.H

Declaration `Boolean lowMemory();`

Action Calls **TVMemMgr::safetyPoolExhausted** to check the state of the safety pool.

See also **TVMemMgr::safetyPoolExhausted**

maxCollectionSize variable — CONFIG.H

Declaration
```
const maxCollectionSize = (int)((65536uL - 16)/sizeof( void * ));
```

Action *maxCollectionSize* determines the maximum number of elements that may be contained in a collection, which is essentially the number of pointers that can fit in a 64K memory segment.

maxFindStrLen constant — CONFIG.H

Declaration `const maxFindStrLen = 80;`

Action Gives the maximum length for a find string in **TEditor** applications.

maxReplaceStrLen constant — CONFIG.H

Declaration `const maxReplaceStrLen = 80;`

Action Gives the maximum length for a replacement string in **TEditor** applications.

maxViewWidth constant — VIEWS.H

Declaration `maxViewWidth = 132;`

Action Sets the maximum width of a view.

See also **TView::size**

mb*XXXX* constants — SYSTEM.H

Action These constants can be used when examining the **TEvent::buttons** data member of an *evMouse* event record. For example,

```
if ( (event.what == evMouseDown) &&  (event.buttons == mbLeftButton) )
   doLeftButtonDownAction();
```

Values The following constants are defined:

Table 16.20 Mouse button constants

Constant	Value	Meaning
mbLeftButton	0x01	Set if left button was pressed
mbRightButton	0x02	Set if right button was pressed

See also **getMouseEvent**

message function — UTIL.H

Declaration
```
void *message(TView *receiver, ushort what, ushort command, void
              *infoPtr);
```

Action **message** sets up a command event with the arguments *event*, *command*, and *infoPtr*, and then, if possible, invokes `receiver->handleEvent` to handle this event. **message** returns 0 if *receiver* is 0, or if the event is not handled successfully. If the event is handled successfully (that is, if **handleEvent** returns *event.what* as *evNothing*), the function returns *event.infoPtr*. The latter can be used to determine which view actually handled the dispatched event. The *event* argument is usually set to *evBroadcast*. For example, the default **TScrollBar::scrollDraw** sends the following message to the scroll bar's owner:

```
message( owner, evBroadcast, cmScrollBarChanged, this );
```

The above message ensures that the appropriate views are redrawn whenever the scroll bar's *value* changes.

See also **TView::handleEvent**, **TEvent**, *cmXXXX*, *evXXXX*

messageBox function — MSGBOX.H

Declaration
```
ushort messageBox( const char *msg, ushort aOptions );
ushort messageBox( ushort aOptions, const char *msg, ... );
```

Action The first form displays the given message. The second form uses *msg* as a format string using the addition parameters that follow it. *aOptions* is set to one of the message box constants listed under the *mfXXXX* entry.

See also *mfXXXX*

messageBoxRect function — MSGBOX.H

Declaration
```
ushort messageBoxRect( const TRect &r, const char *msg, ushort aOptions
                       );
ushort messageBoxRect( const TRect &r, ushort aOptions, const char *msg,
                       ... );
```

Action

The first form displays the given message in the given rectangle. The second form uses *msg* as a format string using the addition parameters that follow it. *aOptions* is set to one of the message box constants listed under the *mfXXXX* entry.

See also *mfXXXX*

mfXXXX constants — MSGBOX.H

Values The following message box constants are defined for use with **messageBox**:

Table 16.21 **messageBox** constants

Constant	Value	Meaning
mfWarning	0x0000	Display a Warning box
mfError	0x0001	Display a Error box
mfInformation	0x0002	Display an Information Box
mfConfirmation	0x0003	Display a Confirmation Box
mfYesButton	0x0100	Put a Yes button into the dialog
mfNoButton	0x0200	Put a No button into the dialog
mfOKButton	0x0400	Put an OK button into the dialog
mfCancelButton	0x0800	Put a Cancel button into the dialog

The standard "Yes, No, Cancel" dialog box is defined:

```
mfYesNoCancel  = mfYesButton | mfNoButton | mfCancelButton;
```

The standard "OK, Cancel" dialog box is defined:

```
mfOKCancel     = mfOKButton | mfCancelButton;
```

See also **messageBox**

moveBuf function — UTIL.H

Declaration `void moveBuf( void *dest, const void *source, ushort attr, ushort count);`

Action Moves text into a buffer. *dest* points to a user-created buffer; *source* should be an array of char. *count* bytes are moved from *source* into the low bytes of corresponding words in *dest*. The high bytes of the words in *dest* are set to *attr*, or remain unchanged if *attr* is zero.

See also **TDrawBuffer, moveChar, moveCStr, moveStr**

moveChar function — UTIL.H

Declaration `void moveChar( void *dest, char c, ushort attr, ushort count);`

Action Moves characters into a buffer. *dest* points to a user-created buffer. The low bytes of the first *count* words of *dest* are set to *c*, or remain unchanged if *c* is '\0'. The high bytes of the words are set to *attr*, or remain unchanged if *attr* is zero.

See also **TDrawBuffer, moveBuf, moveCStr, moveStr**

moveCStr function — UTIL.H

Declaration `void moveCStr( void *dest, const char *str, ushort attrs);`

Action Moves a two-colored string into a buffer. *dest* points to a user-created buffer. The characters in *str* are moved into the low bytes of corresponding words in *dest*. The high bytes of the words are set to *lo(attr)* or *hi(attr)*. Tilde characters (~) in the string are used to toggle between the two attribute bytes passed in the *attr* word.

See also **TDrawBuffer, moveChar, moveBuf, moveStr**

moveStr function — UTIL.H

Declaration `void moveStr( void *dest, const char *str, ushort attrs);`

Action Moves a string into a buffer. *dest* points to a user-created buffer. The characters in *str* are moved into the low bytes of corresponding words in *dest*. The high bytes of the words are set to *attr*, or remain unchanged if *attr* is zero.

See also **TDrawBuffer** class, **moveChar, moveCStr, moveBuf**

newStr function — UTIL.H

Declaration `char *newStr( const char *s);`

Action Dynamic string creation. If *s* is empty, **newStr** returns a 0 pointer; otherwise, `strlen(s)+ 1` bytes are allocated, containing a copy of *s* (with a terminating '\0'), and a pointer to the first byte is returned. You can use **delete** to dispose of such strings.

of*XXXX* constants — VIEWS.H

Action These mnemonics are used to refer to the bit positions of the **TView::options** data member. Setting a bit position to 1 indicates that the view has that particular attribute; clearing the bit position means that the attribute is off or disabled. For example,

```
myWindow.options = ofTileable | ofSelectable;
```

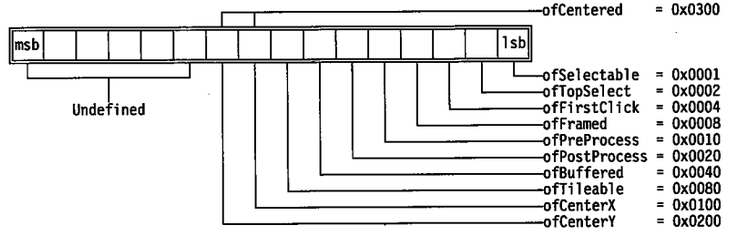
Values The following option flags are defined:

Table 16.22: Option flags

Constant	Meaning
ofSelectable	Set if the view should select itself automatically (see *sfSelected*); for example, by a mouse click in the view, or a *Tab* in a dialog box.
ofTopSelect	Set if the view should move in front of all other peer views when selected. When the *ofTopSelect* bit is set, a call to **TView::select** corresponds to a call to **TView::makeFirst**. Windows (**TWindow** and descendants) by default have the *ofTopSelect* bit set, which causes them to move in front of all other windows on the desktop when selected. See also **TView::select, TGroup::makeFirst**.
ofFirstClick	If clear, a mouse click that selects a view will have no further effect. If set, such a mouse click is processed as a normal mouse click after selecting the view. Has no effect unless *ofSelectable* is also set. See also **TView::handleEvent**, *sfSelect*, *ofSelectable*.
ofFramed	Set if the view should have a frame drawn around it. A **TWindow**, and any class derived from **TWindow**, has a **TFrame** as its last subview. When drawing itself, the **TFrame** will also draw a frame around any other subviews that have the *ofFramed* bit set. See also **TFrame, TWindow**.
ofPreProcess	Set if the view should receive focused events before they are sent to the focused view. Otherwise clear. See also *sfFocused*, *ofPostProcess*, *TGroup::phase*.
ofPostProcess	Set if the view should receive focused events whenever the focused view fails to handle them. Otherwise clear. See also *sfFocused*, *ofPreProcess*, *TGroup::phase*.
ofBuffered	Used for **TGroup** objects and classes derived from **TGroup** only: Set if a cache buffer should be allocated if sufficient memory is available. The group buffer holds a screen image of the whole group so that group redraws can be speeded up. In the absence of a buffer, **TGroup::draw** calls on each subview's **drawView** method. If subsequent memory allocation calls fail, group buffers will be deallocated to make memory available.
ofTileable	Set if the desktop can tile (or cascade) this view. Usually used only with **TWindow** objects.
ofCenterX	Set if the view should be centered on the *x*-axis of its owner when inserted in a group using **TGroup::insert**.
ofCenterY	Set if the view should be centered on the *y*-axis of its owner when inserted in a group using **TGroup::insert**.
ofCentered	Set if the view should be centered on both axes of its owner when inserted in a group using **TGroup::insert**.

The *options* bits are defined as follows:

Figure 16.4
options bit flags



See also **TView::options**

operator + — MENUS.H

Declaration
```
TSubMenu& operator + ( TSubMenu& s, TMenuItem& i );
TSubMenu& operator + ( TSubMenu& s1, TSubMenu& s2 );
TStatusDef& operator + ( TStatusDef& s1, TStatusItem& s2 );
TStatusDef& operator + ( TStatusDef& s1, TStatusDef& s2 );
```

Action These overloaded **+** operators are used with the **TSubMenu, TMenuItem, TStatusDef,** and **TStatusItem** constructors to create status lines and nested menus.

See also Chapter 2, pages 32 and 35

operator delete NEW.CPP

Declaration `void *operator delete( void *blk );`

Action Frees the allocation block from the heap. If the safety pool is exhausted, **TVMemMgr::resizeSafetyPool** is called. This frees the old safety pool and allocates a new, default safety pool (usually 4,096 bytes).

See also: **TVMemMgr::resizeSafetyPool**

operator new NEW.CPP

Declaration `void *operator new( size_t sz );`

Action Tries to allocate *sz* bytes on the heap. A pointer to the allocation is returned if **new** succeeds. If the allocation fails, cache buffers (if any) are freed one by one and the allocation attempt is retried. If this fails and the safety pool is "exhausted", **new** calls **abort**. Otherwise, allocation in the safety pool is attempted. This attempt, whether successful or not, sets the **TVMemMgr::safetyPool** pointer to 0 (indicating that the safety pool is "exhausted"). If **new** does allocate successfully from the safety pool, a pointer to the allocation is returned; otherwise, **abort** is called. Operator **new** is a friend function of **TBufListEntry**.

See also: **TVMemMgr::safetyPool**, **TVMemMgr::safetyPoolExhausted**, **TVMemMgr::resizeSafetyPool**

positionalEvents constant VIEWS.H

Declaration `positionalEvents = evMouse;`

Action Defines the event classes that are *positional events*. The *focusedEvents* and *positionalEvents* masks are used by **TGroup::handleEvent** to determine how to dispatch an event to the group's subviews. If an event class isn't contained in *focusedEvents* or *positionalEvents*, it is treated as a broadcast event.

See also **TGroup::handleEvent**, **TEvent**, *evXXXX*, *focusedEvents*

sb*XXXX* constants VIEWS.H

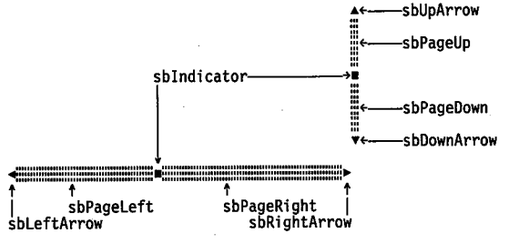
Action These constants define the different areas of a **TScrollBar** in which the mouse can be clicked.

The **TScrollBar::scrollStep** function converts these constants into actual scroll step values. Although defined, the *sbIndicator* constant is never passed to **TScrollBar::scrollStep**.

Table 16.23 Scroll bar part constants

Constant	Value	Meaning
sbLeftArrow	0	Left arrow of horizontal scroll bar
sbRightArrow	1	Right arrow of horizontal scroll bar
sbPageLeft	2	Left paging area of horizontal scroll bar
sbPageRight	3	Right paging area of horizontal scroll bar
sbUpArrow	4	Top arrow of vertical scroll bar
sbDownArrow	5	Bottom arrow of vertical scroll bar
sbPageUp	6	Upper paging area of vertical scroll bar
sbPageDown	7	Lower paging area of vertical scroll bar
sbIndicator	8	Position indicator on scroll bar

Figure 16.5 Scroll bar parts



The following values can be passed to **TWindow::standardScrollBar**:

Table 16.24 **standardScrollBar** constants

Constant	Value	Meaning
sbHorizontal	0x0000	Scroll bar is horizontal
sbVertical	0x0001	Scroll bar is vertical
sbHandleKeyboard	0x0002	Scroll bar responds to keyboard commands

See also **TScrollBar**, **TScrollBar::scrollStep**, **TWindow::standardScrollBar**

selectMode enumeration VIEWS.H

Declaration `selectMode = {normalSelect, enterSelect, leaveSelect};`

Action Used internally by Turbo Vision.

See also **TGroup::execView, TGroup::setCurrent**

sf*XXXX* constants VIEWS.H

Action These constants are used to access the corresponding bits in the *TView::state* data member. You must never modify *TView::state* directly; instead, use **TView::setState**.

Values The following state flags are defined:

Table 16.25: State flag constants

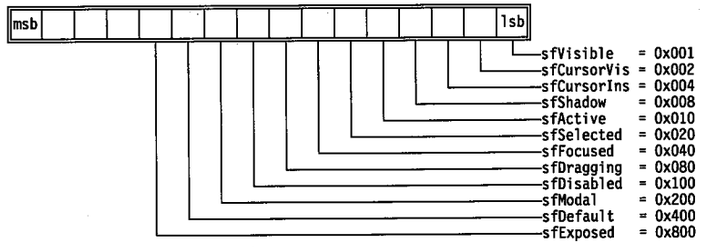
Constant	Meaning
sfVisible	Set if the view is visible on its owner; otherwise, clear. Views are by default *sfVisible*. You can use **TView::show** and **TView::hide** to modify *sfVisible*. An *sfVisible* view is not necessarily visible on the screen, since its owner might not be visible. To test for visibility on the screen, examine the *sfExposed* bit or call **TView::exposed**.
sfCursorVis	Set if a view's cursor is visible; otherwise, clear. Clear is the default. You can use **TView::showCursor** and **TView::hideCursor** to modify *sfCursorVis*.
sfCursorIns	Set if the view's cursor is a solid block; clear if the view's cursor is an underline (the default). You can use **TView::blockCursor** and **TView::normalCursor** to modify *sfCursorIns*.
sfShadow	Set if the view has a shadow; otherwise, clear.
sfActive	Set if the view is the active window or a subview in the active window.
sfSelected	Set if the view is the currently selected subview within its owner. Each **TGroup** object has a *current* data member that points to the currently selected subview (or is 0 if no subview is selected). There can be only one currently selected subview in a **TGroup**.
sfFocused	Set if the view is focused. A view is focused if it is selected and all owners above it are also selected; that is, if the view is on the chain that is formed by following each *current* pointer of all **TGroups** starting at *application* (the topmost view in the view hierarchy). The last view on the focused chain is the final target of all *focused events*.
sfDragging	Set if the view is being dragged; otherwise, clear.
sfDisabled	Set if the view is disabled; clear if the view is enabled. A disabled view will ignore all events sent to it.
sfModal	Set if the view is modal. There is always exactly *one* modal view in a running Turbo Vision application, usually a **TApplication** or **TDialog** object. When a view starts executing (through an **execView** call), that view becomes modal. The modal view

Table 16.25: State flag constants (continued)

	represents the apex (root) of the active event tree, getting and handling events until its **endModal** method is called. During this "local" event loop, events are passed down to lower subviews in the view tree. Events from these lower views pass back up the tree, but go no further than the modal view. See also *sfSelected*, *sfFocused*, **TView::setState**, **TView::handleEvent**, **TGroup::execView**.
sfDefault	This is a spare flag, available to specify some user-defined default state.
sfExposed	Set if the view is owned directly or indirectly by the *application* object, and therefore possibly visible on the screen. **TView::exposed** uses this flag in combination with further clipping calculations to determine whether any part of the view is actually visible on the screen. See also **TView::exposed**.

Values The *state* flag bits are defined as follows:

Figure 16.6 *state* flag bit mapping



See also *TView::state*

specialChars variable TTYPES.H

Declaration
```
extern const uchar specialChars[] =
{
   175, 174, 26, 27, ' ', ' '
};
```

Action Defines the indicator characters used to highlight the focused view in monochrome video mode. These characters are displayed if the *showMarkers* variable is *True*.

See also *showMarkers* variable

StreamableInit enumeration — TTYPES.H

Declaration `enum StreamableInit { streamableInit };`

Action Each streamable class has a special "builder" constructor that takes argument *streamableInit*. This is defined in this enumeration to provide a unique data type for the constructor argument.

See also `TView::TView( streamableInit );`

TScrollChars typedef — VIEWS.H

Declaration `typedef char TScrollChars[5];`

Action An array representing the characters used to draw a **TScrollBar**.

See also **TScrollBar**

uchar typedef — TTYPES.H

Declaration `typedef unsigned char uchar;`

Action Provides a synonym for **unsigned char**.

ushort typedef — TTYPES.H

Declaration `typedef unsigned short ushort;`

Action Provides a synonym for **unsigned short**.

wf*XXXX* constants — VIEWS.H

Action These mnemonics define bits in the *flags* data member of **TWindow** objects. If the bits are set, the window will have the corresponding attribute: The window can move, grow, close, or zoom.

Values The window flags are defined as follows:

Figure 16.7 Window flags

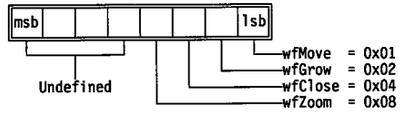


Table 16.26 Window flag constants

Constant	Value	Meaning
wfMove	0x01	Window can be moved.
wfGrow	0x02	Window can be resized and has a grow icon in the lower-right corner.
wfClose	0x04	Window frame has a close icon that can be mouse-clicked to close the window.
wfZoom	0x08	Window frame has a zoom icon that can be mouse-clicked to zoom the window.

If a particular bit is set (= 1), the corresponding property is enabled; otherwise, if clear (= 0), that property is disabled.

See also *TWindows::flags*

wnNoNumber constant — VIEWS.H

Declaration `const ushort wnNoNumber = 0;`

Action If the *TWindow::number* data member holds this constant, it indicates that the window is not to be numbered and cannot be selected via the *Alt+number* key. If *number* is between 1 and 9, the window number is displayed, and *Alt-number* selection is available.

See also *TWindow::number*

write_args structure — VIEWS.H

Declaration

```
struct write_args
{
   void far *self;
   void far *target;
   void far *buf;
   ushort offset;
};
```

Action Used internally by **TView::writeView**

wp*XXXX* constants VIEWS.H

Action These constants define the three standard color mapping assignments for windows. By default, a *TWindow::paletee* is *wpBlueWindow*. The default for **TDialog** objects is *wpGrayWindow*.

Values Three standard window palettes are defined:

Table 16.27 Standard window palettes

Constant	Value	Meaning
wpBlueWindow	0	Window text is yellow on blue
wpCyanWindow	1	Window text is blue on cyan
wpGrayWindow	2	Window text is black on gray

See also *TWindow::palette*, **TWindow::getPalette**

I N D E X

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

USER'S GUIDE

TURBO VISION® FOR C++

BORLAND

CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300.
OFFICES IN: AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, NEW ZEALAND, SINGAPORE, SWEDEN AND THE UNITED KINGDOM ■ PART # 14MN-TVN01-10 ■ BOR 2413