THINK C
Symantec C++

Visual Architect and THINK Class Library Guide

Credits

Documentation David Allcott, Bob Foster, Bonnie Hill, Jeff Mattson, Jeanne Munson, Steve Raphel, Susan Rona, and Cambridge Publications

Development Patrick Beard, Walter Bright, Thomas Emerson, Bob Foster, Greg Howe, Michael Kahl, Darrell LeBlanc, John Micco, Pat Nelson, Daniel Podwall, and Phil Shapiro

Quality Assurance Celso Barriga, Constantine Hantzopoulos, Kevin Irlen, Yuen Li, and Christopher Prinos

Technical Support Celso Barriga, Colen Garoutte-Carson, Rick Hartmann, and Scott Shurr

Project Management David Allcott, Constantine Hantzopoulos, and David Neal

Product Management Steve Levine and Peggy Liu

Copyright © 1989, 1993, 1994 Symantec Corporation.
All Rights Reserved. Printed in U.S.A.

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
(408) 253-9600

Symantec C++, THINK C, THINK Reference, and THINK Pascal are trademarks of Symantec Corporation. Other brands and their products are trademarks of their respective holders and should be noted as such.

The *Visual Architect and THINK Class Library Guide* is copyrighted and all rights are reserved. Information in this document is subject to change without notice and does not represent a commitment on the part of Symantec Corporation. The software described in this document is furnished under a license agreement. The document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Symantec Corporation. The *Visual Architect and THINK Class Library Guide* contains samples of names and addresses to illustrate features and capabilities of THINK and Symantec C++. Any similarities to names and addresses of actual individuals is purely coincidental.

SYMANTEC CORPORATION MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY, OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Contents

Part One: Getting Started

Part Two: Using the THINK Class Library

Part Four: THINK Class Library Reference

THINK C
Symantec C++
Getting Started

Part One

Welcome 1

This guide describes how to use the THINK Class Library and Visual Architect to program on the Macintosh with Symantec C++. The THINK Class Library is a collection of classes that define a generic Macintosh application. Visual Architect is a tool that can assist you in designing and assembling the user interface portion of an application using the THINK Class Library.

Contents

1 Welcome

What's in This Guide

This guide has four parts: "Getting Started," "Using the THINK Class Library," "Using Visual Architect," and "The THINK Class Library Reference."

Getting Started

This first section presents background information about the THINK Class Library and Visual Architect and the basic concepts of their usage. It also includes two introductory tutorials.

About the THINK Class Library and Visual Architect	This chapter provides a brief introduction to the THINK Class Library and Visual Architect and describes the benefits of using them in programming with Symantec C++.
THINK Class Library Basic Concepts	This chapter gives you an overview of the THINK Class Library and describes how the different parts of the THINK Class Library fit together.
Visual Architect Basic Concepts	This chapter gives you an overview of how Visual Architect works to assist in the design and construction of a user interface.
Tutorial: Beeper	This chapter presents a basic tutorial for Visual Architect.
Tutorial: Process Monitor	This chapter presents a more elaborate tutorial demonstrating many of the features and techniques involved in programming with Visual Architect and the THINK Project Manager.

Using the THINK Class Library

This section describes how to use the THINK Class Library to construct your application.

Programming with the THINK Class Library	This chapter provides detailed information on how to use the THINK Class Library.
Using Object I/O	This chapter describes Object I/O, a set of features built into the THINK Class Library that help you store and retrieve objects on disk as documents and resources.
Exception Handling and RTTI	This chapter describes the exception handling and Run-Time Type Identification (RTTI) mechanisms used in the THINK Class Library. The exception handling mechanism lets you handle errors in a consistent and graceful way. RTTI allows you to test the class of objects at run-time.

Using Visual Architect

This section describes how to use Visual Architect to design and construct your application's user interface.

Creating and Editing Views	This chapter explains how to create and edit views. Views are the visible objects that make up an application's user interface.
Creating and Editing Menus and Commands	This chapter describes how menus and commands are created and edited. Menus are the standard menus used in an application. Commands link user actions with user interface objects.

Editing Classes	This chapter explains how to create and define classes derived from the THINK Class Library.
Generating Code	This chapter tells how to generate the code with Visual Architect for a complete application as well as how to incorporate your own code.

The THINK Class Library Reference

This section describes each class in the THINK Class Library and the global variables and library routines used by these classes.

Chapter 14– Chapter 125	These chapters describe each class in the THINK Class Library.
Global Variables	This chapter lists all of the global variables that the THINK Class Library uses.
Library Routines	This chapter describes library routines used throughout the THINK Class Library.

How to Use This Guide

If you are new to programming using the THINK Class Library, you should read all the chapters in the “Getting Started” section and try the two tutorials. Then read the “Using the THINK Class Library” and “Using Visual Architect” sections to obtain more detailed information.

If you have already programmed using the THINK Class Library, you should still read the chapters on Visual Architect in the “Getting Started” section and try the two tutorials. To acquaint yourself with features new to programming with the THINK Class Library, read the “Using the THINK Class Library” and “Using Visual Architect” sections.

Visual Architect makes exploring the THINK Class Library easy and fun, so don’t be afraid to experiment while you are learning.

1 Welcome

About the THINK Class Library and Visual Architect

2

This chapter provides a brief introduction to the THINK Class Library and Visual Architect and describes the benefits of using them in writing applications for the Macintosh with Symantec C++.

Contents

2 About the THINK Class Library and Visual Architect

Introduction

This chapter introduces two of the most powerful components of Macintosh programming with Symantec C++: the THINK Class Library and Visual Architect. It describes what they are and what they're good for.

What you should know

Before reading any of the chapters following this one, you should be familiar with the fundamentals of Macintosh programming. In particular, you should understand QuickDraw, the Window Manager, and the Memory Manager. You should also be familiar with Symantec C++. As you go through subsequent chapters, remember that there is a lot of new material. Don't be discouraged if it takes a while to understand the relationships among all the components.

What Is the THINK Class Library?

The THINK Class Library is a collection of C++ classes that implement a standard Macintosh application. These classes take care of basic Macintosh functions, such as handling menu commands, updating windows, dispatching events, dealing with the operating system, managing memory, maintaining the Clipboard, printing, and so on. Using the THINK Class Library frees you from most of the responsibility for establishing the core operations common to Macintosh applications. Rather than developing your code from scratch with Macintosh Toolbox and Operating System calls, you can implement existing classes in the THINK Class Library in which these lower-level interfaces have already been established. This lets you concentrate on the essential pieces of your program. Of course, when using the THINK Class Library, you still have full access to all Macintosh routines.

What Is Visual Architect?

Visual Architect is a development tool you use in conjunction with the THINK Class Library to construct the user interface portion of your application's code. It lets you define classes derived from the THINK Class Library by using interactive, visual tools, rather than by writing C++ code. You can use Visual Architect at any stage during the development of your application: to begin your design, or to fine-tune the application's user interface when the project is nearly complete. Most likely, you will find it helpful to work with Visual Architect throughout the development of your application, and to

build your application incrementally by going back and forth between Visual Architect and the THINK Project Manager.

Visual Architect is a flexible and powerful tool. It can serve as:

- An interface prototyper
- A code generator
- A resource editor
- An explicator of the THINK Class Library

Visual Architect as an interface prototyper

Visual Architect helps you develop your application's user interface: windows, dialog boxes, buttons, menus, and so on. Using the familiar Macintosh interactive, graphical tools, you can easily create, modify, and test these elements completely separate from the THINK Project Manager.

Visual Architect as a code generator

By using its own macro files, Visual Architect creates from your interface design a set of source files that are automatically incorporated into your Symantec C++ project. These source files let you perform hand-coded modifications that remain in place through subsequent development cycles using Visual Architect. Furthermore, the macro files are written in a scripting language that allows easy customization.

Visual Architect as a resource editor

Like ResEdit™ and Resorcerer™, Visual Architect deals with resources and creates resource files for incorporation into your project. However, Visual Architect works at a higher level than do other resource editors, and lets you construct visual structures called views rather than just the individual elements such as buttons and scroll bars.

Visual Architect as an explicator of the THINK Class Library

By studying the commented code generated by Visual Architect, you can learn about the structure and implementation of classes in the THINK Class Library. Visual Architect uses the THINK Class Library in a standard way, and encourages you to experiment with the code it generates as a means of understanding the THINK Class Library.

Conclusion

Programming in Symantec C++ with the THINK Class Library and Visual Architect lets you implement the standard Macintosh core of your application much faster and more easily than if you had to code by hand and from scratch. Furthermore, Visual Architect can facilitate the design of your user interface by providing you instant feedback on the look and feel of your application's visual elements.

2 About the THINK Class Library and Visual Architect

THINK Class Library Basic Concepts 3

The THINK Class Library is a collection of C++ classes that implement a standard Macintosh application. It encapsulates many routine aspects of Macintosh programming, such as handling menu commands, updating windows, dispatching events, dealing with the operating system, maintaining the Clipboard, and so on.

Contents

3 THINK Class Library Basic Concepts

Introduction

This chapter introduces you to the THINK Class Library and the fundamental abstractions that it implements. It presents the conceptual shape of a Macintosh application written using the Library. In addition, it gives you a sense of how the THINK Class Library encapsulates the more routine aspects of Macintosh programming, allowing you to concentrate your development efforts on inventing new forms instead of reinventing old ones.

Overview

The THINK Class Library is organized into three distinct, interacting structures: the class hierarchy, the visual hierarchy, and the chain of command. The class hierarchy is the set of all the classes that make up the THINK Class Library. The visual hierarchy describes the organization of all visible entities, such as windows and buttons, in a given application. The chain of command specifies which objects in an application get to handle commands (such as menu item choices) and in what order. The class hierarchy is static; the relationship of derivation between classes is fixed for all applications and does not change during execution. In contrast, both the visual hierarchy and the chain of command are dynamic structures. They are composed of objects that exist only during the execution of individual programs.

The THINK Class Library converts Macintosh events into calls to member functions defined by the classes of the class hierarchy. Some member functions handle events that affect the visual hierarchy; these visual events include mouse clicks, activate events, and update events. Other member functions handle direct commands—requests that an object perform an action. Direct commands are usually the result of menu choices or their keyboard equivalents.

To convert Macintosh events into calls to appropriate member functions, the THINK Class Library uses an object called a switchboard. The switchboard calls the Macintosh Toolbox functions `WaitNextEvent` (or `GetNextEvent`) repeatedly and, depending on the kind of event, calls a member function of the appropriate object. Each application has only one instance of a switchboard object. Another object, called the bartender, converts menu choices into direct commands.

The Class Hierarchy

The class hierarchy describes the relationships among all the classes in the THINK Class Library. It can be thought of as a family tree. Indeed, the derivation relationship is referred to as inheritance. Each class inherits all of the behavior (member functions) and all of the attributes (data members) of its base classes. Thus the CButton class, being derived from the CControl class, is a control. It has all the attributes and exhibits all the behaviors characteristic of a control.

Some of the classes in the class hierarchy are abstract classes. The THINK Class Library never creates instances of these classes. Abstract classes are used to give common behaviors to a family of objects—that is, to their derived classes.

Note

The class hierarchy is not, strictly speaking, a tree; it does not have a unique root class from which all other classes in the library are directly or indirectly derived. In fact, because some classes in the THINK Class Library have more than one base class, it is not even a forest (a set of disjoint trees). It is a directed, acyclic graph.

The Visual Hierarchy

The visual hierarchy describes all the visible objects, or views, that the THINK Class Library contains. A view is an object of a class descended from the abstract class CView. The visual hierarchy is based on the concept of enclosure. Everything that you see on the screen belongs to—is enclosed by—another visual entity. When a view is created, its enclosure is specified as an argument to its constructor.

The desktop, windows, and panes

The class CView has three important derived classes: CDesktop, CWindow, and CPane.

At the top of the visual hierarchy is the desktop, an object of class CDesktop. Every THINK Class Library application has a unique desktop object. It is the only view that is not enclosed within another view.

The desktop encloses all of the windows in an application. In the THINK Class Library, a window is an object of class CWindow. This class encapsulates the properties and behavior of Macintosh windows.

Each window encloses one or more panes, and those panes may enclose other panes. A pane is an object of class CPane. Panes are the most important kinds of views. All drawing takes place in a pane. The THINK Class Library defines different kinds of panes for displaying various standard visual interface elements. Every pane has its own drawing environment (coordinate system, font, line width, and so on), so you can draw in a pane without worrying about where it is on the screen. The THINK Class Library has classes that make it easy to implement scrollable panes that have logical dimensions which exceed the size of the enclosures in which they are displayed.

Figure 3-1 illustrates a typical visual hierarchy. The desktop encloses windows, each of which encloses at least one pane; each pane, in turn, may enclose one or more nested panes; and so on, recursively. Although the three derived classes—CDesktop, CWindow, and CPane—are siblings in the class hierarchy, objects of these classes are typically *not* at the same level in the visual hierarchy.

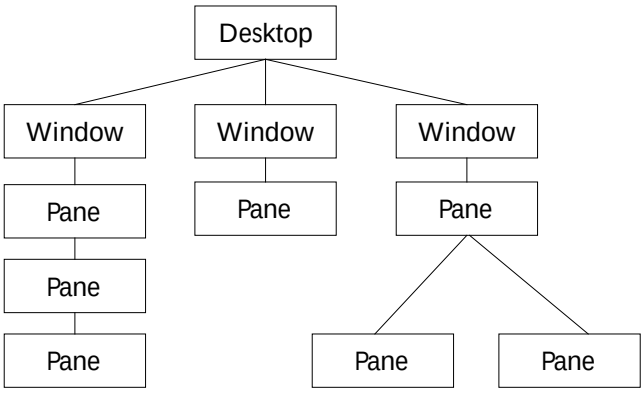


Figure 3-1 Typical visual hierarchy

Unlike the class hierarchy, the visual hierarchy is dynamic. It changes as your program runs. When you open a new document, you add another window to the desktop; and you add another set of panes to the new window.

The visual hierarchy and events

Visual events—mouse events, window activation and update events—work their way down the visual hierarchy. Since Macintosh update and activate events always have a window associated with them, the appropriate member function of a window object is always called directly to notify that window of such an event. Mouse clicks and cursor adjustment notifications always work their way down from the desktop to the active window and from there to the appropriate pane.

Because they are descendants of CBureaucrat (discussed next), views can be in the chain of command, and thus can respond to direct commands.

The Chain of Command

The chain of command specifies which objects handle direct commands, and in what order.

Bureaucrats, supervisors, and the application object

A bureaucrat is an object capable of handling direct commands. A bureaucrat gets to handle a command when its `DoCommand` member function is called with that command as its argument. It is said to handle the command if its `DoCommand` function responds by doing more than merely passing the command to another bureaucrat. Technically, a bureaucrat is an instance of a class derived from CBureaucrat. Derived classes of CBureaucrat can override `DoCommand` to handle commands specific to an application.

The supervisor of a bureaucrat is itself a bureaucrat. The supervisor of a bureaucrat is the object to which the bureaucrat passes direct commands that it doesn't handle itself. More precisely, when a bureaucrat does not know how to handle a particular command, it defers to its supervisor by calling the supervisor's `DoCommand` function with that command as its argument. The default `DoCommand` function, as implemented in class CBureaucrat, calls the `DoCommand` function of its supervisor. Thus a generic bureaucrat takes no responsibility for anything.

When a bureaucrat is constructed, its supervisor is specified as an argument to the CBureaucrat constructor.

Every THINK Class Library program has a unique application object, an instance of class CApplication. The application object is the only bureaucrat that does not have a supervisor. Every bureaucrat is either directly or indirectly supervised by the application object.

The supervision relation is, in fact, a tree—a hierarchy governing how objects communicate among themselves. At the root of the tree is the application object. The collection of bureaucrats, related by supervision, is like an organization chart. Every bureaucrat has at most one supervisor, and the head of the organization is the application object. Whereas visual events work their way down the visual hierarchy, direct commands work their way up through the organization.

The gopher and the chain of command

The first object that has the chance to handle a direct command is called the gopher. Your application is responsible for appointing the bureaucrats that act as the gopher. The identity of the gopher is dynamic, changing during execution in response to events.

Usually, the gopher is a pane in the active window. For instance, an edit pane must be made the gopher in order to receive the commands generated by keystrokes and by menu choices. Objects further up in the visual hierarchy are seldom the gopher. Windows are not made the gopher by THINK Class Library objects, and you are unlikely to find a reason to do so yourself. The application object is made the gopher only when no documents or windows are open.

If the gopher can't handle a command, it passes the command on to its supervisor. Its supervisor can handle the command and return, or defer to its own supervisor. Eventually, either some bureaucrat takes action, or each one defers to its supervisor—until ultimately the command reaches the application object. If the application object doesn't handle the command, no object does.

The chain of command is the list of bureaucrats starting with the gopher and ending with the application object. The chain of command contains all intermediary supervisors. It is a path in the supervision tree. By definition, every object in the chain of command is a bureaucrat.

Figure 3-2 shows a sample chain of command and visual hierarchy. The gopher is a pane. The gopher's supervisor is a document, which also supervises the other pane and the window. A document is a

kind of bureaucrat that manages the communication between windows, files, and menu commands. If the pane can't handle a direct command, it passes the command on to the document. If the document can't handle the command, it passes the command to the application. Note that the desktop's supervisor is the application; this is always the case in THINK Class Library programs.

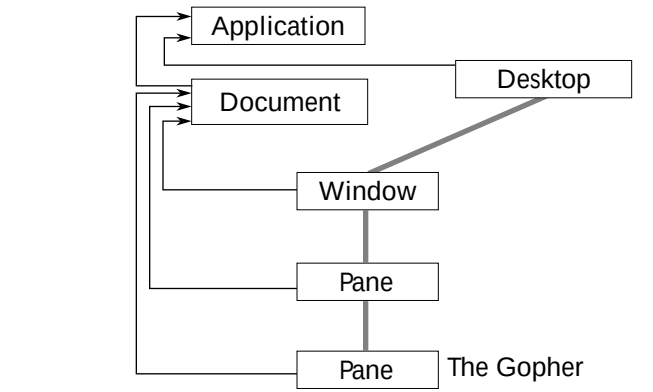


Figure 3-2 The gopher, chain of command (black lines), and visual hierarchy (gray line)

The enclosure relationships of the visual hierarchy are drawn in gray; the supervisory relationship is drawn in black. Notice that although the chain of command and the visual hierarchy overlap, they are nevertheless distinct. In the example, the panes and the window have the same supervisor—the document.

The Flow of Control

The chain of command and the visual hierarchy get their direction from the switchboard. The switchboard gets events from the Macintosh Event Manager and converts the events to member function calls affecting either the chain of command or the visual hierarchy.

Functions that affect the chain of command are usually called through the gopher, the first bureaucrat in the chain. Functions affecting the visual hierarchy are called by the active window or the desktop. Update and activate events affect the visual hierarchy. Mouse clicks can affect the visual hierarchy (if the click is in a

window) or the chain of command (if it's in the menu bar). Keyboard events are dispatched to the gopher as keystroke-related function calls, or in the case of keyboard equivalents for menu items, as commands.

Working with Menus

The THINK Class Library lets you work with your menu commands at a higher level than does the Macintosh Toolbox. Instead of identifying a menu command by its menu ID and item number, the THINK Class Library lets you assign unique command numbers to each item of the menu. With command numbers, you can reassign functions to different menu items without having to rebuild the application. Command numbers represent all direct commands; the `DoCommand` member function of class CBureaucrat takes a command number as its argument.

The THINK Class Library reserves command numbers for the most common Macintosh application commands, such as **New**, **Open**, **Save**, **Quit**, **Page Setup**, and others.

When you choose a command from a menu, the desktop uses the bartender to translate the menu choice into a direct command, which it then passes to the gopher.

Menus need not be, and sometimes cannot be, completely specified when you build your application. For instance, menu items like **Cut** and **Copy** must be enabled and disabled at run-time depending upon whether there is or is not a selection. The THINK Class Library makes it easy to do this. You also can check and uncheck items to reflect whether or not an item is in force. For instance, some items represent toggles; for example, **Bold** might be checked only when newly entered text is in boldface. Or a set of menu items can form an exhaustive and mutually exclusive group; for example, the font in use at the insertion point might be checked on a menu.

A better example is the **Font** menu. It consists of a set of items whose totality cannot be determined at the time that the application is designed. The fonts listed in the menu should be those installed on the machine running the application. Using the THINK Class Library, you can create menus while your program is running to reflect the environment. Choices from menus built at run-time are also converted into direct commands.

The THINK Class Library lets you add menu items to existing menus. For instance, you might want to add the **Font** menu to a general text-handling menu, or you might want to append it to a menu with the names of all the documents your application has opened.

Creating an Application

To create an application that uses the THINK Class Library, you derive new classes from existing classes that the Library provides. Your new classes implement the unique parts of your application; generic application behavior is already taken care of by the base classes in the Library. Generally, you need to derive the following: one class for your unique application object; one or more classes for documents—one for each type of document that your application deals with; and various pane classes, which provide the user interface.

In addition, you need to define your menu structure containing the commands that your application implements, and the linkage between menu items and commands (that is, which command is performed in response to which item). Finally, you may need to define other resources, such as strings of various sorts (for changeable menu item text, error messages, and so on).

Visual Architect streamlines the process of creating, editing, and connecting the classes, menus, commands, and other resources needed by your application. The next chapter introduces you to this powerful tool.

Visual Architect Basic Concepts 4

This chapter explains the basic concepts of Visual Architect. It helps you understand how Visual Architect works and how it can aid in the development of your application's user interface. Part Three in this guide, "Using Visual Architect," discusses Visual Architect in greater detail. If you are not familiar with the THINK Class Library, read the previous chapter, "THINK Class Library Basic Concepts," before proceeding with this chapter.

Contents

4 Visual Architect Basic Concepts

How Visual Architect and the THINK Project Manager Cooperate

Visual Architect is designed for use with the THINK Project Manager. Although it is possible to run Visual Architect by itself, usually you launch it from the THINK Project Manager and alternate between the two while constructing your application's user interface. Also, if you are running System 7 (recommended), you can control some of the THINK Project Manager's functions from Visual Architect using built-in Apple events.

Visual Architect generates source code files that are added automatically to your Symantec C++ project. It also edits a single resource file, which is a part of your project as well.

Source code files created by Visual Architect

The source code files generated by Visual Architect contain definitions and declarations for the classes created while designing your user interface. These files are standard C++ `.cp` and `.h` files and can be opened and edited using the THINK Project Manager. In fact, you usually modify a subset of these files during the development cycle, as discussed later in this chapter. The Visual Architect source code files reside in the Application segment of your project and are compiled and linked with the other files in your project, such as the THINK Class Library files and the files you write yourself.

Note

By inspecting the source code files Visual Architect creates, you can learn the function and implementation of many of the THINK Class Library classes. Visual Architect comments the code it generates to aid in interpretation.

Visual Architect uses special files called macro files to generate source code. A macro file is an ordinary text file that contains C++ source code and macro expressions, which Visual Architect interprets to produce one or more source code files as output. The macro files supplied with Visual Architect can generate the source code for a complete, running THINK Class Library application. Because macro files are ordinary text files, you can modify them to suit your programming style or extend them with new capabilities.

Note

One particular macro file, `GenerateTCLApp`, contains only macro statements that control the other macro files that are used by Visual Architect.

The resource file edited by Visual Architect

In addition to generating source code files, Visual Architect edits a resource file named `VisualArchitect.rsrc`. This file contains many of the standard user-interface resource types, such as `'WIND'` and `'CNTL'`, as well as some of its own types.

Note

You do not need to modify the resources created by `VisualArchitect.rsrc` using any other resource editors.

Visual Architect defines one particular resource type, `'CVue'`, that is used to describe each view or subview, and all the subordinate objects pointed to by the view and subview, such as borders and controls. You will learn more about views later in this chapter.

The source code files generated by Visual Architect, along with `VisualArchitect.rsrc`, are incorporated automatically into your Symantec C++ project, if the THINK Project Manager is running concurrently.

Creating and Modifying Classes

One of Visual Architect's most powerful features is its interactive, graphical environment for deriving classes from the THINK Class Library and defining their data members and member functions.

Although the classes you create are generally derived from the set of base THINK Class Library classes that Visual Architect knows about, you can specify that your classes be derived from other classes you previously created. This allows you greater flexibility and control.

In general, the classes you construct fall into one of three categories of user interface design:

- Views
- Commands
- Menus

Working with Views

Visual Architect provides several predefined views for implementing common graphical representations, such as document windows and floating tool palettes. Each view is implemented by means of a particular class. When you add a view to your project, Visual Architect generates the class necessary for proper functioning of the view, as well as the associated resources. You need not be concerned with the details of the class hierarchy. This frees you to concentrate on developing your application's user interface and visual hierarchy.

The most important view defined by Visual Architect is the main window view. Main window views typically are displayed in an application when the user chooses **New** or **Open** from the **File** menu. They display the contents of an associated file.

Visual Architect also defines views for implementing, for example, dialog boxes, alerts, splash screens, and floating windows, as shown in Figure 4-1.

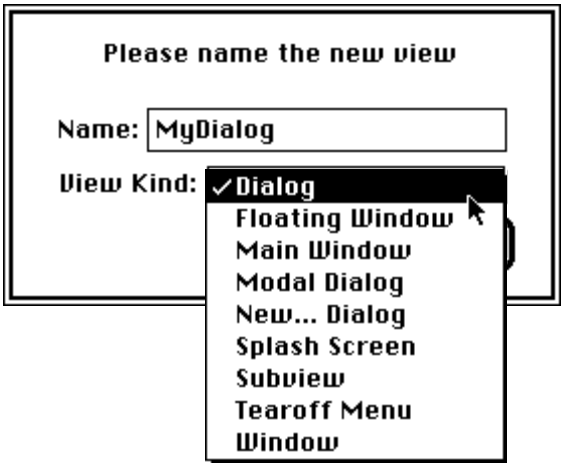


Figure 4-1 The New View dialog box, showing predefined view types

Visual Architect lets you change the attributes of a view, such as whether it has a scroll pane and whether its window has a close box. Instead of writing the code by hand, you can change these attributes through dialog boxes in Visual Architect.

Adding graphical elements to views is also easy. Visual Architect's Tool Palette, shown in Figure 4-2, is used in much the same way as Resedit is used: You can add view elements—buttons, check boxes, static and editable text fields, scrollable text fields, and pictures—as well as more complex and user-definable elements. Visual Architect then generates the necessary classes.

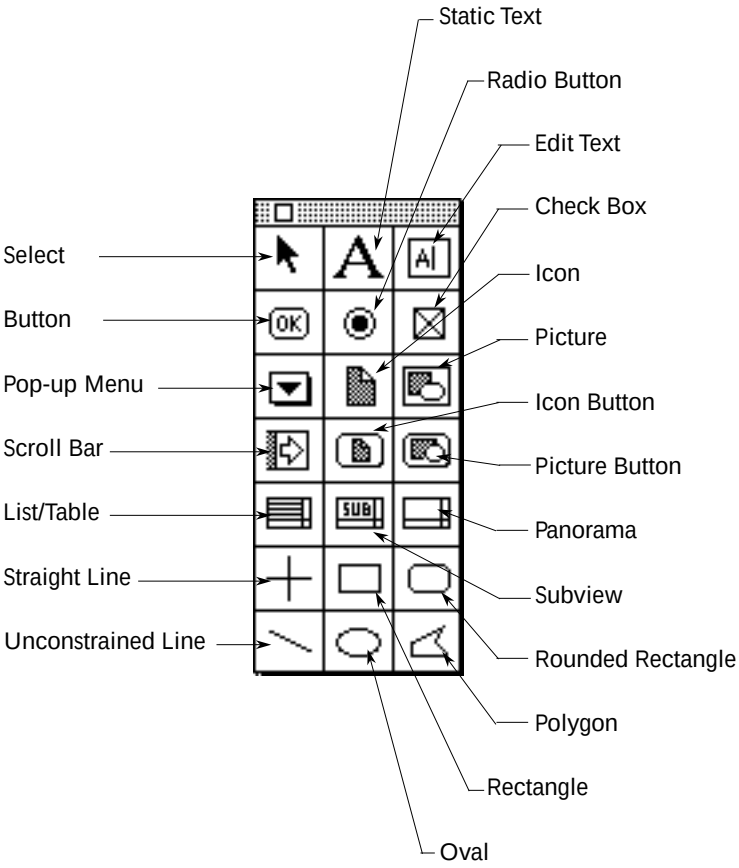


Figure 4-2 The Tool Palette

Visual Architect provides a convenient way to change important data member values in each class in an object's class hierarchy through CView. For example, if you are constructing a dialog box and want to define its OK button, you can access data members in CButton, CControl, CPane, and CView.

In Figure 4-3, the left window shows the dialog box **MyDialog** under construction. The right window shows access to data members for classes in the OK button's class hierarchy.

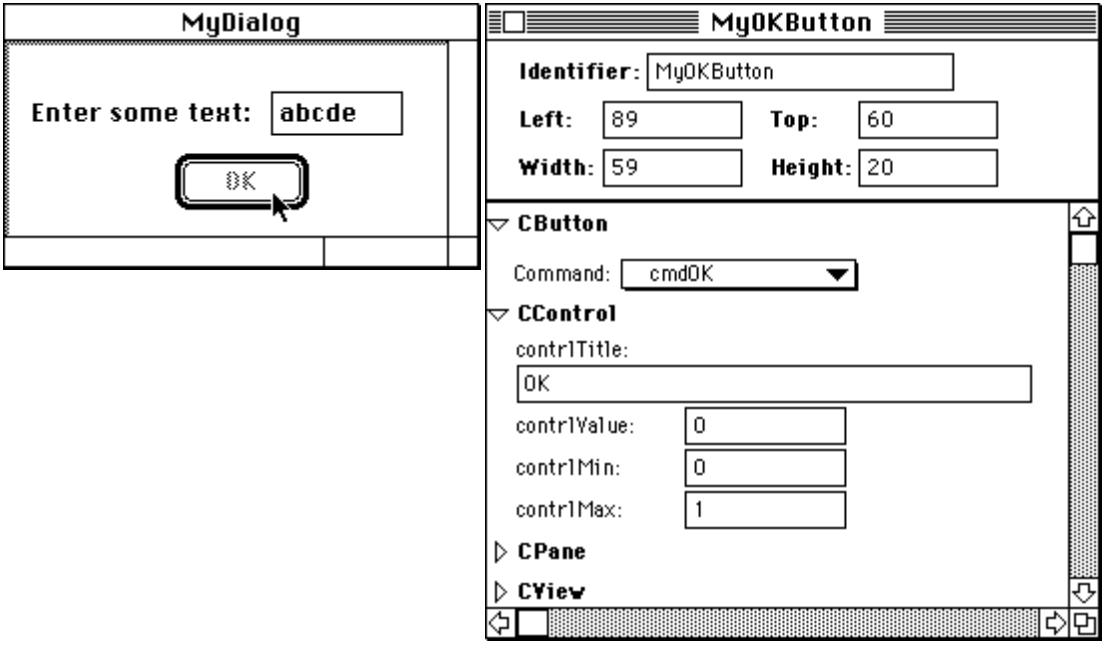


Figure 4-3 Accessing the OK button's class hierarchy

Defining the Commands

Commands link user actions with classes' member functions. Through commands, objects communicate with each other the action a user desires. The THINK Class Library predefines many such commands, such as closing a window, changing text attributes, saving a file, and quitting the application. These pre-existing commands significantly reduce the amount of coding you need to do. However, you have to define commands for those functions unique to your program.

When you define a new command using Visual Architect, you indicate those classes you would like to respond to the command. Visual Architect generates skeleton code for the appropriate classes, into which you later insert your code to handle the command. Because one of the most common results of a user action is the opening of a window, you can specify that a command cause a window to be opened, and Visual Architect generates the necessary code.

For example, in Figure 4-4 you want to add a new command, `cmdOpenMyDialog`, to class CMain. When `cmdOpenMyDialog` is sent to CMain, CMain opens a view as defined by the class CMyDialog. Notice some of the predefined commands available in the scrolling list.

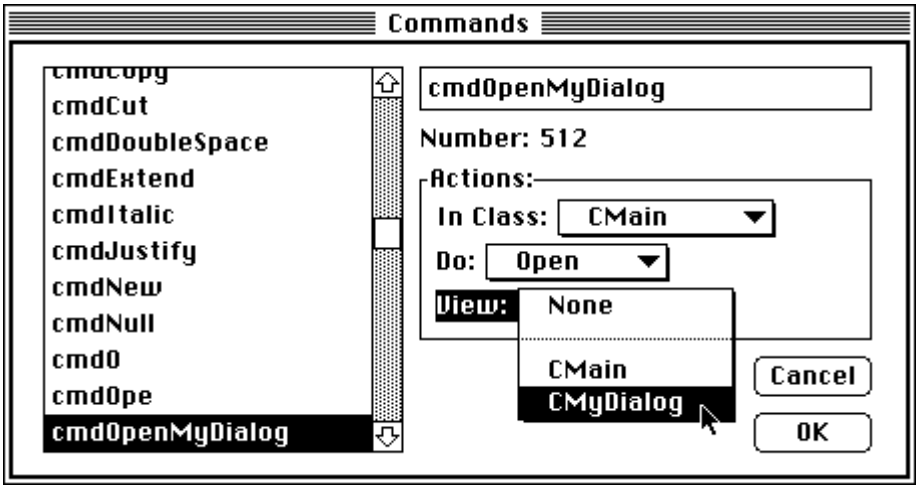


Figure 4-4 Adding a command using the Commands dialog box

Visual Architect also lets you associate commands with the elements that send them, such as buttons and menu items.

Constructing Menus

Visual Architect lets you assemble menus and link menu items with their associated commands. The interface with which you build menus is similar to that of ResEdit. In addition to constructing the menu resources, Visual Architect adds the command number that each menu item sends when the user selects the item. All you have

to do is specify the command from the list of available commands, as shown in Figure 4-5.

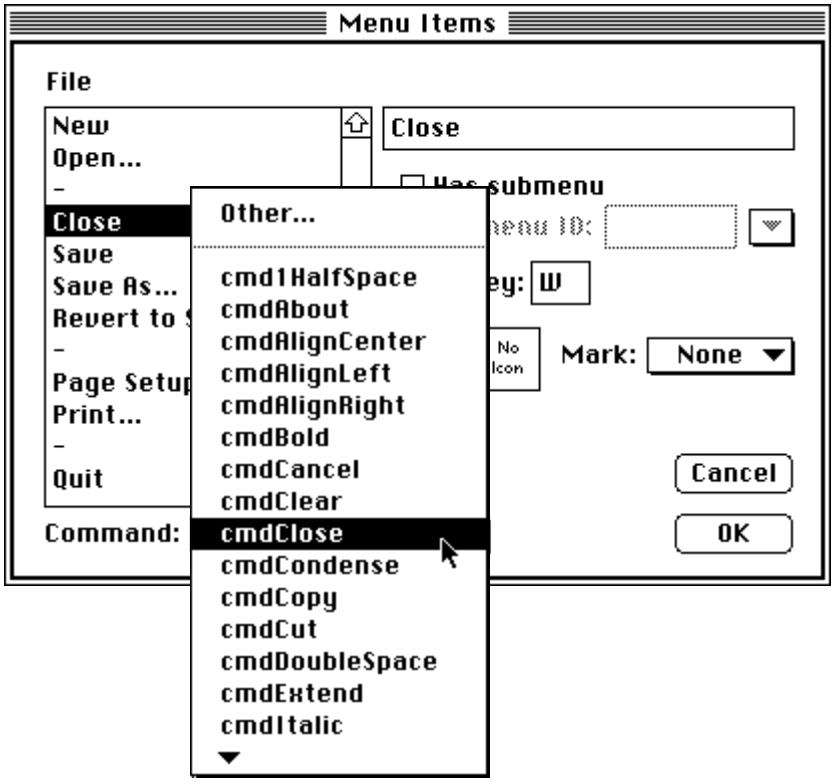


Figure 4-5 Linking commands via the Menu Items dialog box

Visual Architect's menu editor supports menu item marks and icons, as well as hierarchical, pop-up, and tear-off menus. In addition, you can specify the menus you want to appear in the menu bar.

Adding Balloon Help

Balloon Help can be added to panes, menus, and menu items with a simple menu choice, followed by typing the text you want to appear in the balloon. Because user interface elements can be in different states—such as enabled and disabled—when the application is running, you can define up to four different balloons for each element.

Previewing Your Views

Visual Architect lets you see how your view works before you complete a development cycle of generating source code, compiling it, and running the application. When you preview, or try out, a view, it appears exactly as it would in your running application. Furthermore, you can interact with the view just as if the application were running. For example, you can scroll, resize, and reposition the view's windows. All the view's elements—such as pop-up and tear-off menus, edit text fields with type constraints, scrolling edit text, custom buttons, and button groupings—are active as well.

Previewing your views lets you glimpse the final product of your work quickly and conveniently. This enables the design process to proceed more rapidly.

Modifying the Code Generated by Visual Architect

Programming is rarely accomplished in one step. Most often, you design some user interface elements in Visual Architect, do some hand coding in the THINK Project Manager, compile, run, and inspect your project, then return to Visual Architect to make changes and start the cycle again. Therefore, Visual Architect does not force you to live with the code it generates "as is." Much of the standard, commented C++ code it generates is skeleton code, a user-interface framework to which you add the source code that makes your application unique. Furthermore, due to the structure of the source code files, Visual Architect does not, in subsequent code-generation steps, overwrite changes you make to your code by hand.

Split-level classes

Visual Architect facilitates and protects hand-coding with a technique known as split-level classes. Most classes defined in Visual Architect are implemented as two classes: a lower-level class, reserved for Visual Architect, and a higher-level class, reserved for your custom programming. The first time Visual Architect generates source code for a graphical element, it generates both classes, each in a separate file.

The lower-level class contains code that Visual Architect generates from scratch each time the element it defines is modified. Most of Visual Architect's generated code appears here. Generally, there is no reason for you to modify this code.

The upper-level class is derived from the lower-level class. It contains code, such as added and deleted member functions, that you hand-write on top of the skeleton functions generated by Visual Architect. The upper-level class lets you override lower-level class member functions and add additional data members. Visual Architect writes to this file only once, when it generates the class files after you first define the class. After that, you are responsible for maintaining the upper-level class file.

Visual Architect employs a specific naming convention to distinguish levels. If you create a class named `MyDialog` in Visual Architect, Visual Architect creates four files: `x_MyDialog.cp`, `MyDialog.cp`, `x_MyDialog.h`, and `MyDialog.h`. Files with the `x_` prefix contain the lower-level class; those without the prefix contain the higher-level class.

Where to Go Next

You should now have a basic understanding of both the THINK Class Library and Visual Architect. The THINK Class Library gives you the classes necessary to specify fully and organize the visual entities that make up your application. Visual Architect is a powerful environment for developing your user interface with the THINK Class Library.

The two tutorials in Chapters 5 and 6 give you the opportunity to try out the THINK Class Library and Visual Architect. The first demonstrates the concepts you've just learned; the second reinforces them and presents some additional features and techniques. After performing the tutorials, read Part Two, "Using the THINK Class Library," and Part Three, "Using Visual Architect," for more detailed information.

4 Visual Architect Basic Concepts

Tutorial: Beeper 5

This tutorial demonstrates some of the basic properties of Visual Architect. It shows you how to create a simple application called Beeper using the THINK Class Library. Visual Architect helps you build the user interface portion of Beeper.

5 Tutorial: Beeper

About the Tutorial

The tutorial gets you started with Visual Architect and builds your confidence using it as a development tool. The Beeper application you will build is simple. It allows you to call up a dialog box containing an edit text field into which you enter a number representing the number of times your Macintosh will beep. The dialog box also contains a push button that starts the beeping.

This tutorial contains a lot of information. You should plan to spend about one hour completing it.

Getting Started

Creating the Beeper project

Because you are creating this application from scratch, you first need to create a Beeper project:

1. From the Finder, launch THINK Project Manager, which resides in the `Symantec C++ for Macintosh` folder.

   A **File Open** dialog box appears.

2. Choose **New** from the **File Open** dialog box.

   The **New Project** dialog box appears, similar to the one shown in Figure 5-1.

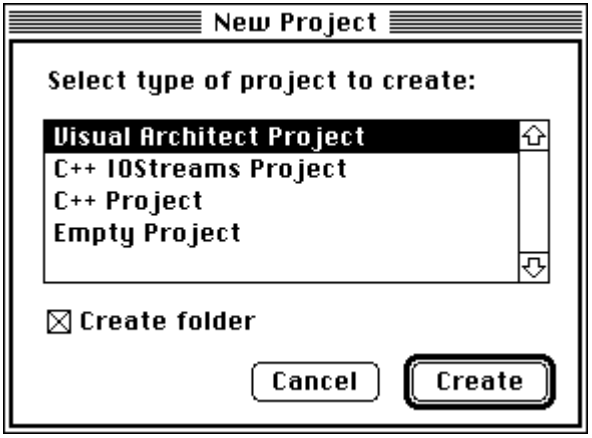


Figure 5-1 The New Project dialog box

3. Select Visual Architect Project from the list, make sure Create folder is checked, and click Create.

   A **File Save** dialog box appears.

4. In the **File Save** dialog box, name the new project `beeper`, place it somewhere outside the `Symantec C++ for Macintosh` folder, and click Save.

A folder named `beeper ƒ` is created automatically, and the `beeper.π` project is created inside it. A window titled beeper.π appears.

Designing the User Interface

You now run Visual Architect to design Beeper's user interface. This process involves creating dialog boxes, push buttons, and menus.

Starting Visual Architect

Visual Architect is launched directly from the THINK Project Manager:

1. In the `beeper.π` project window, shown in Figure 5-2, locate the segment titled Visual Architect.

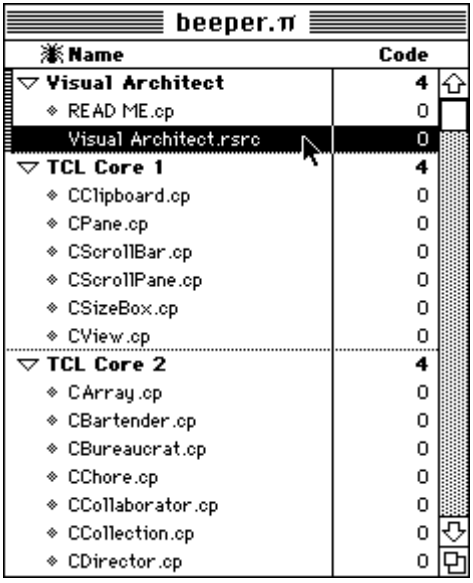


Figure 5-2 The beeper.π project window

2. Double-click the `Visual Architect.rsrc` file to launch Visual Architect with this file.

The Views List window appears, showing a list of defined views in `Visual Architect.rsrc`. See Figure 5-3.

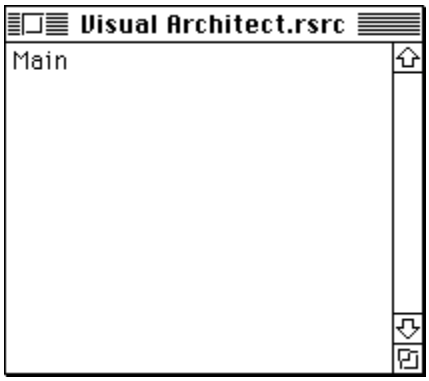


Figure 5-3 The Visual Architect.rsrc Views List window

Creating the view for the Beeper dialog box

At present, Main is the only view defined in `Visual Architect.rsrc`. To add a new view, the **Beeper** dialog box:

1. Choose **New View** from the **View** menu.

   A **New View** dialog box appears, in which you specify a view's name and kind, as shown in Figure 5-4 .

Figure 5-4 The New View dialog box

2. In the **New View** dialog box, type `BeeperDialog` in the field Name, leave Dialog chosen for View Kind, and click OK.

A new window titled BeeperDialog appears. This is the Beeper Dialog view edit window, in which you construct the BeeperDialog view.

Adding pane elements to the dialog box

To add pane elements to the **Beeper** dialog box, first add a static text pane:

1. Click the **Tools** menu and tear off the Tool Palette shown in Figure 5-5 by dragging beyond the edge of the Palette. Put the Tool Palette off to the side of the screen and out of the way.

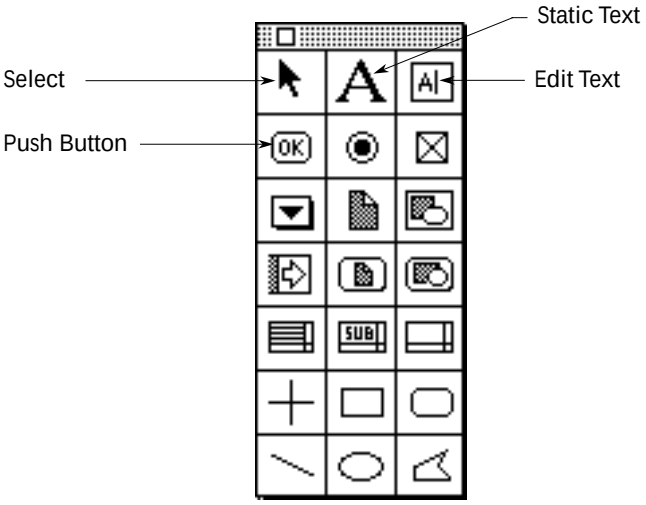


Figure 5-5 The Tool Palette

Note

At this point, you may want to look ahead to Figure 5-10, which shows how the finished dialog box appears.

2. Click the Static Text tool on the Tool Palette.

   Note that the cursor changes to an I-beam when it is over the BeeperDialog window.

3. Click the cursor near the left side of the dialog box to position the static text pane.

   A blinking insertion point appears, indicating that you should enter the text for the item.

4. Type `Number of beeps`: and either click elsewhere in the window or press Enter to end the typing task.

Note

You can reposition any panes you add to a view by dragging them within the window.

Next, add an edit text pane:

1. Choose the Edit Text tool from the Tool Palette. Note that the cursor changes to a cross hair when it is over the dialog box. Click the cursor to the right of the static text pane you just added, to position the edit text pane.

   An edit text box appears. You can constrain the type of text the user is allowed to enter into the edit text pane by selecting a special THINK Class Library class, as shown in the next step. The edit text pane then becomes an instance of this class.

2. Choose **Class** from the **Pane** menu and choose **CIntegerText** from the pop-up menu, as shown in Figure 5-6.

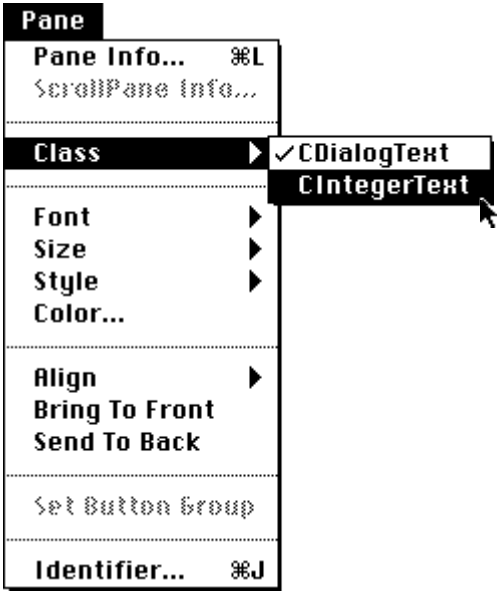


Figure 5-6 Setting the class of the edit text pane

The next item to add to the dialog box is an OK button:

1. Choose the Push Button tool from the Tool Palette.
2. Click below the edit text pane you just added.

   A push button named OK appears. By default, the first push button added to a view is named OK.

The last item to add to the dialog box is the button that calls the beep function:

1. Once again, choose the Push Button tool from the Tool Palette.

2. Click to the right of the edit text pane you added previously.

   A push button named Cancel appears. By default, the second push button added to a view is named Cancel. In this dialog, you must rename it.

3. Type `Beep` to rename it.

Creating the command to execute the beep function

By default, a `cmdOK` command is sent by the OK button when the button is clicked. The `cmdOK` command is a predefined THINK Class Library command for closing dialog boxes in response to clicks on the OK button. However, the command that the Beep button sends when it is clicked has not yet been defined. This command will call the beep function, and you must define it yourself:

1. Choose **Commands** from the **Edit** menu.

   The **Commands** dialog box opens, similar to the one shown in Figure 5-7. The **Commands** dialog box lets you add your own commands to those already defined in the THINK Class Library.

2. Choose **New Command** from the **Edit** Menu (Command-K) or press Return to activate the edit text field. Type `cmdBeep` to name the command.

   This step adds a new command item named `cmdBeep`.

3. Choose CBeeperDialog from the In Class pop-up menu, as shown in Figure 5-7, to indicate that the `cmdBeep` command should be handled by the class CBeeperDialog.

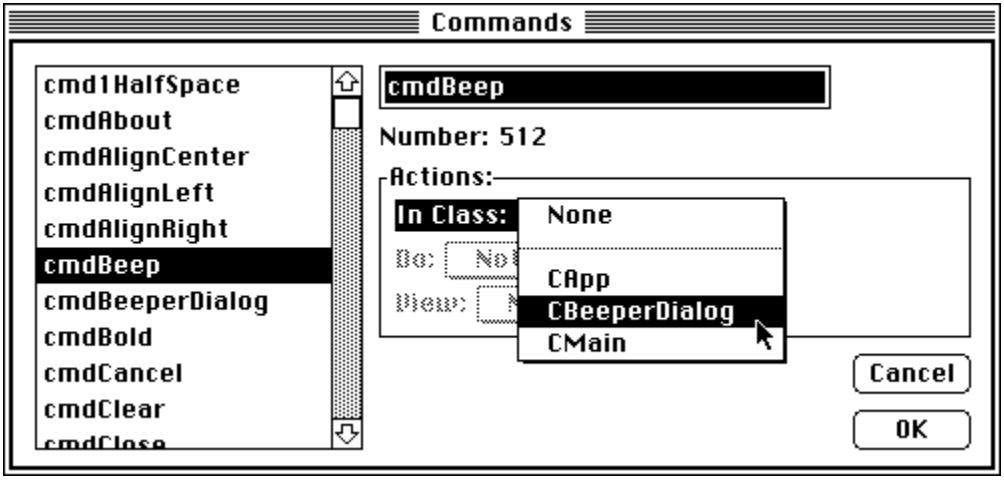


Figure 5-7 Using the Commands dialog box to choose classes that handle a command

4. Choose Call from the Do pop-up menu to indicate that `cmdBeep` calls a function, namely the one that causes the beeps.

Visual Architect later generates skeleton code for the member function, into which you hand-code your CBeeperDialog class. Note that a command number (`512`, in this case) is generated automatically for this command. This is the number that THINK Class Library routines use to identify the command.

5. Click OK to close the **Commands** dialog box.

Associating the cmdBeep command with the Beep button

To have the Beep button send a `cmdBeep` command when it is clicked:

1. Choose **Pane Info** from the **Pane** menu (Command-L).

The Pane Info window for the push button opens, which allows you to manipulate the values of many of the class variables in the selected pane's class and in its base classes up through `CView`.

2. Click the small triangle next to CButton to access the data members of the CButton class, as shown in Figure 5-8.

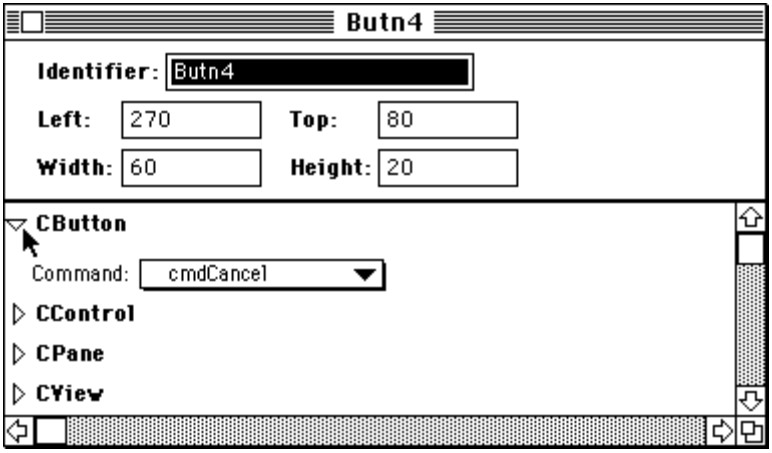


Figure 5-8 Opening the CButton class in the Pane Info window

3. Choose cmdBeep from the Command pop-up menu, which lists available commands.

4. Close the Pane Info window by clicking its close box.

   Now a `cmdBeep` command is sent when the Beep button is clicked.

Setting the default command

Next, you should specify that `cmdBeep` is the default command for the **Beeper** dialog box. The default command is the one sent when the user presses Return or Enter.

1. Choose **Set Default Command** from the **View** menu.

This brings up a dialog box titled **Default Command**, as shown in Figure 5-9. As you can see, `cmdOK` is currently defined as the default command.

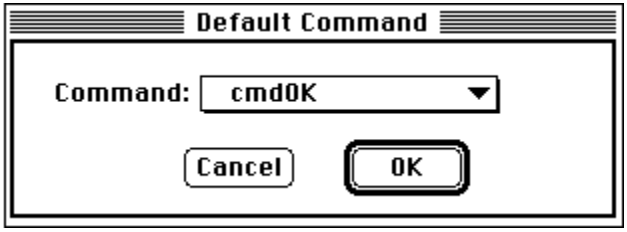


Figure 5-9 The Default Command dialog box

2. In the **Default Command** dialog box, choose cmdBeep from the Command pop-up menu and click OK.

   Note that the Beep button has now been outlined in the BeeperDialog view edit window and that the OK button has lost its outline. The **Beeper** dialog box should resemble the one shown in Figure 5-10.

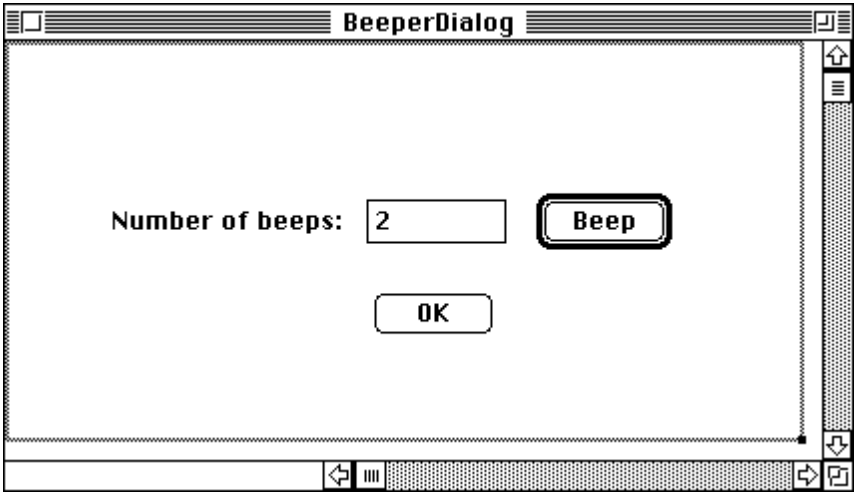


Figure 5-10 The completed Beeper dialog box

3. Close the BeeperDialog view edit window by clicking its close box; you have now finished constructing the Beeper dialog box.

Adding a push button to the Main view to open the Beeper dialog box

The `Visual Architect.rsrc` file has a default view defined called Main, as you saw when you first started Visual Architect. To add a push button to this view that, when clicked, brings up the **Beeper** dialog box:

1. Double-click Main in the Views List window.

   The Main view edit window opens. The Main view already contains two panes, a picture of class CPicture (the group of circles) and a static text panes of class CStaticText ("hello, world!"). You can better visualize these items by displaying their item numbers, as shown in the next step.

2. Choose **Show Item Numbers** from the **Options** menu to display the item numbers in the Main view edit window.

   Right now, the Main window has a default size too small to display a new button without the user having to scroll. Increase the size of the window with the next step.

3. Using the size box in the Main view editing window, extend the window's size downward by approximately one inch.

4. Resize the gray rectangle (the window's `portRect`) to fit snugly within the window using the sizing handle at its lower right corner, as shown in Figure 5-11.

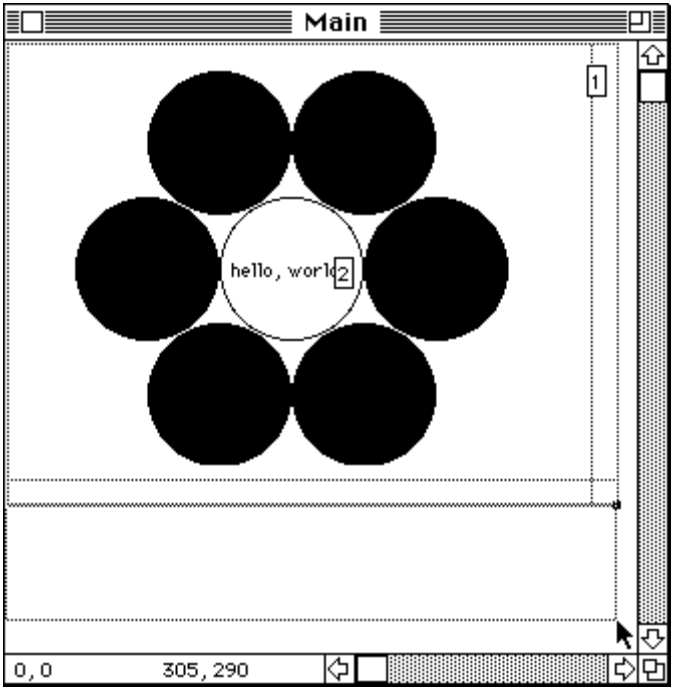


Figure 5-11 Resizing the Main view window's portRect

5. Now add the push button that brings up the **Beeper** dialog box. Choose the Push Button tool from the Tool Palette.

6. Click just below the bottom of the circles to position the button and type `Beeper…` to name it, as shown in Figure 5-12.

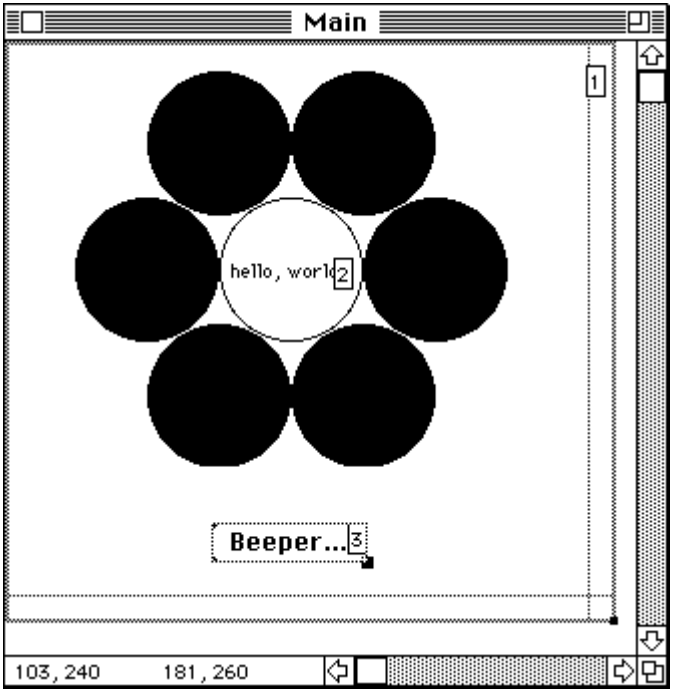


Figure 5-12 Creating the Beeper button in the Main view edit window

Creating the command to call up the Beeper dialog box

As with the Beep button in the **Beeper** dialog box, you need to create a command to call up the **Beeper** dialog box. This time, you do it in a slightly different way.

1. Choose **Pane Info** from the **Pane** menu (Command-L) to bring up the Pane Info window for the push button.

2. Click the small triangle next to `CButton` to access its data members.

3. Choose Other from the top of the Commands pop-up menu.

   The **Commands** dialog box opens, in which you can now create a new command.

4. Choose **New Command** from the **Edit** Menu (Command-K) or press Return to activate the edit text field. Type `cmdBeep`erDialog to name the field.

5. Choose CMain from the In class: pop-up menu to indicate that this is the class that will handle the command.

6. Choose Open from the Do pop-up menu to indicate that the command should open a view.

   Now indicate that you want this command to open the CBeeperDialog view.

7. Choose CBeeperDialog from the **View** pop-up menu, as shown in Figure 5-13, to indicate that the command should open a view of class CBeeperDialog (in other words, the **Beeper** dialog). Visual Architect will later generate the code to open the **Beeper** dialog.

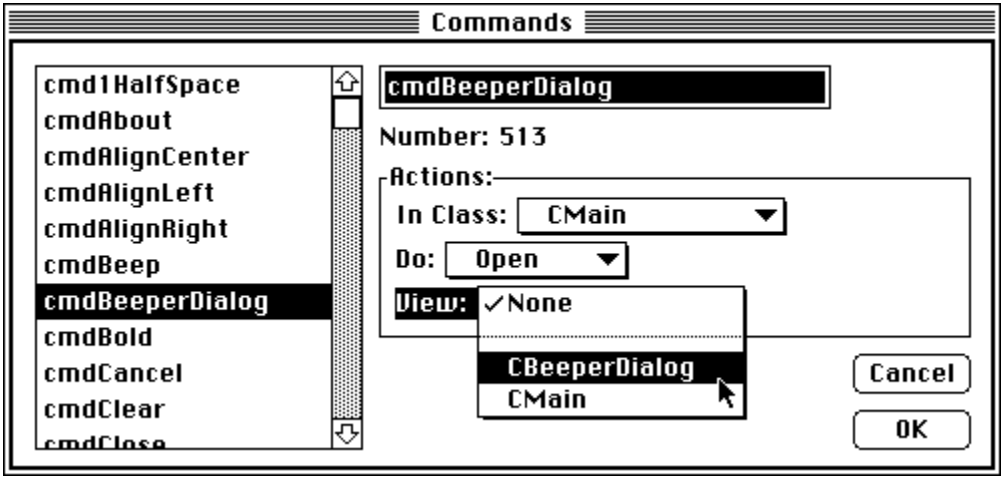


Figure 5-13 Setting the view that the cmdBeeperDialog command will open

8. Close the **Commands** dialog by clicking OK.

   As you can see in the Pane Info window, `cmdBeeperDialog` is now set as the command the Beeper button sends when it is clicked.

9. Close the Pane Info window.

You are now finished designing the user interface using Visual Architect.

Previewing your view

Visual Architect lets you try out how the view will look and feel in your application. All the graphical elements in a preview are active.

1. Choose **Try Out** from the **View** menu (Command-Y).

   The Main view window opens exactly as it opens in the running application.

2. Verify that the Beeper button is highlighted when clicked, and that the scroll bars and size box function properly.

3. Close the Main preview window by clicking in its close box.

You should now save your work.

- Choose **Save** from the **File** menu (Command-S).

Generating the Code and Updating the Symantec C++ Project

Visual Architect must now generate the code to implement all the classes associated with the user interface you designed. To have it do this:

1. Choose **Generate All** from the THINK Project Manager menu, which looks like the THINK Project Manager application icon, as shown in Figure 5-14.

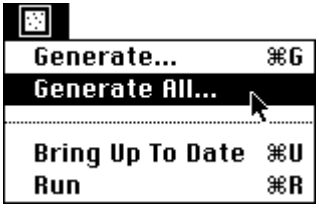


Figure 5-14 Choosing Generate All from the THINK Project Manager menu

Note

Because you're using Visual Architect for the first time to generate code for this project, you need to choose **Generate All**. For any subsequent code generation for this project, you can choose **Generate**, so code is generated only for those classes that have changed since the last code generation.

A dialog opens, showing the progress of the code generation. See Figure 5-15.

Figure 5-15 The code generation progress dialog

By default, Visual Architect uses the macro file `GenerateTCLApp` to generate code to the `Source` folder, which is in the `Beeper ƒ` folder. After Visual Architect generates the files, it automatically adds them to your project.

2. Switch to THINK Project Manager and scroll to the end of the `beeper.π` project window.

Visual Architect has added the new source files to your project in the Application segment, as shown in Figure 5-16.

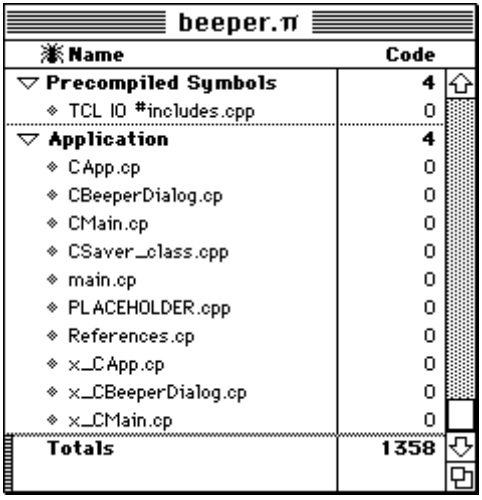


Figure 5-16 The beeper.π project window showing the Application segment

Modifying the Generated Code

To modify code generated by Visual Architect so you can write the beep function:

1. Open the `x_CBeeperDialog`.cp file in the Application segment by double-clicking its name.

Note

Because the `x_CBeeperDialog`.cp file has an `x_` prefix, it is identifiable as containing the lower-level class description for the Beeper dialog view. This file has an associated header file `x_CBeeperDialog.h`. The lower-level files are regenerated from scratch each time you make changes to the BeeperDialog view and generate code in Visual Architect.

---

2. Find the member function `x_CBeeperDialog::DoCmdBeep()` and copy it in its entirety to the Clipboard.

3. Open the file `CBeeperDialog.cp`.

Note

Because the `CBeeperDialog.cp` file has no prefix, it is identifiable as containing the upper-level class description for the BeeperDialog view. This file has an associated header file `CBeeperDialog.h`. Because Visual Architect does not generate these two files again, your hand-coding will not be overwritten.

4. Paste the copied function at the end of `CBeeperDialog.cp` and rename it `CBeeperDialog::DoCmdBeep()`.

   Because the class `CBeeperDialog` is derived from `x_CBeeperDialog`, the member function `DoCmdBeep()` in `CBeeperDialog.cp` overrides the one in `x_CBeeperDialog.cp`.

   As is, `CBeeperDialog::DoCmdBeep()` doesn't do anything, so you need to modify it to look like this:

```
/*****************************************
   DoCmdBeep {OVERRIDE}

   Respond to cmdBeep command.
*****************************************/

   void CBeeperDialog::DoCmdBeep()

   {
      long n;
      n = fBeeperDialog_Edit2->GetIntValue();
      for (long i = 0; i < n; i++)
         SysBeep(1);
   }
```

   The expression:

```
fBeeperDialog_Edit2->GetIntValue()
```

   calls the function `GetIntValue()`, which is a member of the class CIntegerText from which `fBeeperDialog` is derived. The function returns the value in the edit text field as an integer.

5. Near the top of the `CBeeperDialog.cp` file, remove the `//` comment characters from the line:

   ```
   // #include "AppCommands.h"
   ```

   This step is necessary because `cmdBeep` is defined in `AppCommands.h`.

Note

By default, the inclusion of the `AppCommands.h` header file is commented out to prevent unnecessary compilation of files whenever the `AppCommands.h` file is changed.

6. Open the file `CBeeperDialog.h` by selecting its name in `CBeeperDialog.c` and choosing **Open Selection** from the **File** menu (Command-D). As a public member function of the `CBeeperDialog` class, add the definition:

   ```
   void DoCmdBeep();
   ```

You have now completed the necessary code modifications.

Updating and Running the Application

With the design and coding of the Beeper application complete, you can compile and link the `beeper.π` project:

1. Choose **Bring Up To Date** from the **Project** menu (Command-U) in THINK Project Manager.

   Allow time for the compilation to occur, because the project contains many files.

   If you receive any compile or link errors, be sure that the code was entered exactly as written in the previous section.

2. Choose **Run** from the **Project** menu.

When Beeper starts, the Main view window opens with the Beeper button. Clicking this button makes the **Beeper** dialog box appear. You can enter an integer value into the edit text field and click Beep, and your Macintosh beeps that number of times.

Where to Go Next

This tutorial demonstrated some of the basic techniques used in working with Visual Architect. Now that you have a good idea of what the THINK Class Library and Visual Architect can do for you, you should proceed with the tutorial in the next chapter. The Process Monitor tutorial covers Visual Architect and the THINK Class Library in greater depth. You should also read Part Two, “Using the THINK Class Library,” and Part Three, “Using Visual Architect” for more information.

For now, you can try adding more items to the **Beeper** dialog box and more views to the application. Doing so will help you get a feel for how Visual Architect’s code generation scheme works in incremental development.

5 Tutorial: Beeper

Tutorial: Process Monitor 6

In this tutorial, you graphically create the user interface for a simple application with Visual Architect and generate code for it. The application contains many of the basic user interface elements: two main windows, a subview, a **New... Dialog**, push buttons, check boxes, scroll bars, text fields, and menus.

6 Tutorial: Process Monitor

About the Tutorial

The application you will build is called Process Monitor. It displays a list of all processes currently running on your Macintosh and lets you arrange the list by name, process serial number, or signature. The application presents two views of this information: one shows only the list of processes; the other not only shows the list of processes but also displays additional related information, including controls that may be manipulated by the user. Three push buttons are displayed that let the user:

- Enter the debugger
- Kill the selected process
- Bring the selected process to the front

In addition, a group of check boxes is displayed that show the current settings of the `'SIZE'` resource flags for the selected process.

The application has a **New... Dialog**, which comes up when **New** command is chosen from the **File** menu and lets the users choose the view they want to look at. The choices are Process List and Information, and Process Only.

You may have noticed that the key piece of information displayed by these views is the same, that is, the list of running processes. Rather than create this component twice, you need to set it up only once and use the same component in both views. The creation of a subview is what gives you the ability to perform this time-saving action.

A considerable amount of information is contained in this tutorial, and you shouldn't worry if you don't understand everything. Most of it will become clear to you as you proceed through the tutorial. All will become apparent as you use and become more familiar with Visual Architect. You should spend a few hours completing the tutorial.

Getting Started

What's special about Process Monitor.π

The `Process Monitor.π` project is the same as a project created with the Visual Architect project model, with the following additions:

- Added to the `Visual Architect.rsrc` file were icon resources that you will use later for custom buttons.
- Added to the project were the source files from the folder `Source`, that are pre-edited versions of files created by Visual Architect.
- Added to the project were the source files from the folder `Extra Sources`.

The use of these files will become apparent as you progress through the tutorial.

Starting with the Process Monitor.π project

Normally when you start a Visual Architect project, you use the Visual Architect project model supplied with Symantec C++. But a project has been created to make working with this tutorial easier. To get started, open the project.

- Double-click the `Process Monitor.π` project, which can be found in `Process Monitor ƒ`, inside the `TCL Demos` folder.

At this point, the THINK Project Manager is launched, and the `Process Monitor.π` project window appears on the right side of your screen, as shown in Figure 6-1.

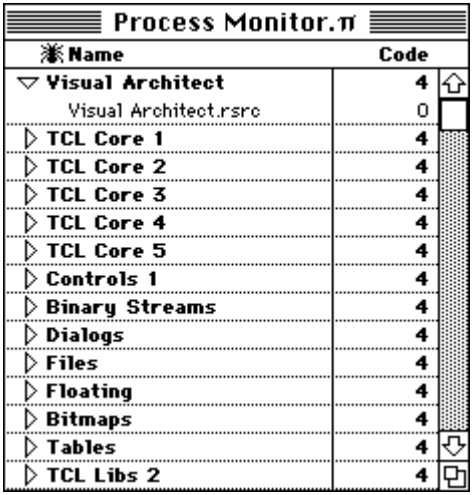


Figure 6-1 The Process Monitor.π project window

You now need to start Visual Architect.

- Double-click the `Visual Architect.rsrc` file at the top of the project window.

You see a Visual Architect Views List window like the one shown in Figure 6-2. As you create new views in the project, their names are displayed in this window. A view can be a dialog, a window, or a subview. Every Visual Architect project you create with the Think Project Manager has its own folder and unique version of the

`Visual Architect.rsrc` file within that folder, to maintain the special view resources for the project.

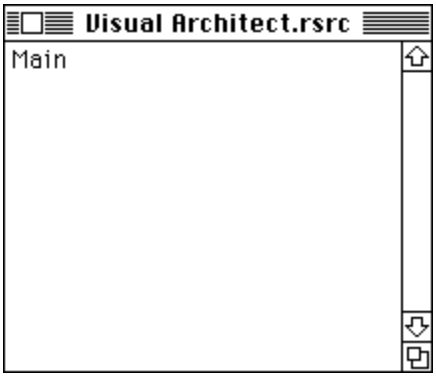


Figure 6-2 The Visual Architect project window

The Views List window contains a sample view called Main, which displays a “Hello, World!” message. You will change the characteristics of this view to create the Process Monitor application.

A note about resources in the Visual Architect.rsrc file

Some of the user interface elements that Visual Architect helps you create are icons and icon buttons. To be available for your selection, however, the resources for the icons you want to use must be in the form of compiled `'cicn'`s and `'ICON'`s resident in the `Visual Architect.rsrc` file. In general, you can use ResEdit or Resorcerer to cut and paste icons from a resource file into any `Visual Architect.rsrc` file.

This tutorial uses special icons for some of its controls. To save time, a `Visual Architect.rsrc` file is provided, with these resources already copied into it.

Setting the application information

You should now specify the top-level characteristics of your application.

1. Choose **Application** from the **Edit** menu to bring up the **Application Info** dialog box similar to the one shown in Figure 6-3.

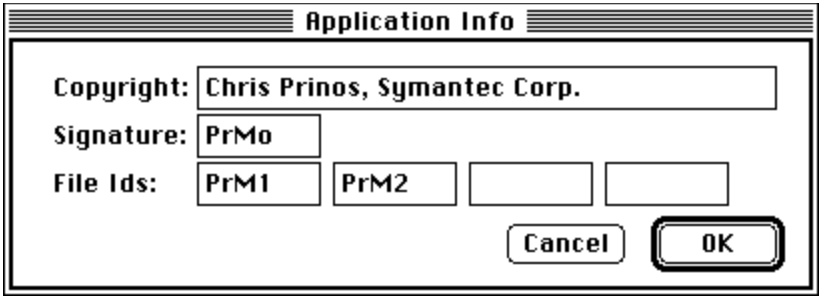


Figure 6-3 The Application Info dialog box

2. Type your name and company, or fictional ones, in the Copyright field.

3. Type a four-letter signature, `PrMo`, in the signature field.

4. Type `PrM1` and `PrM2` for the file IDs. File IDs are used by the Finder to associate your application with its particular document types.

5. Click OK to close the **Application Info** dialog box.

Don't change the application information after starting your project, unless you don't mind some of your files being regenerated.

Building the User Interface

Creating the main window

You're ready to create your main window. Start by opening the default Main view.

1. Double-click Main in the Views List window.

   The Main view edit window appears. You need to empty it and prepare it for your own additions.

2. Choose **Select All** (Command-A) followed by **Clear** from the **Edit** menu.

   You need to change the characteristics of the main window.

3. Choose **View Info** from the **View** menu.

   A **View Info** dialog box such as the one shown in Figure 6-4 appears.

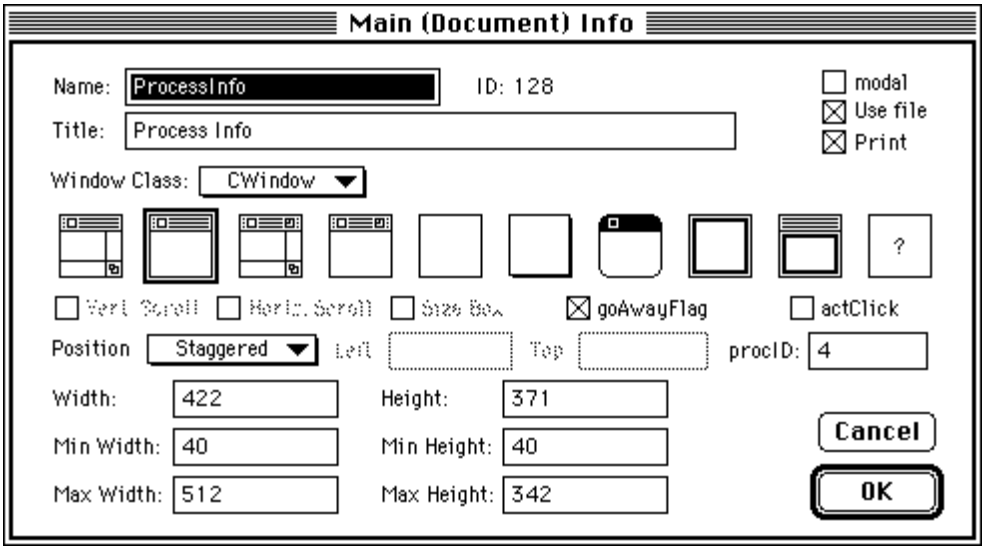


Figure 6-4 The Main View Info dialog box

4. Type `ProcessInfo` (no space) into Name and `Process Info` into Title.

5. Click the second window type icon from the left to make the view's window a `noGrowDocProc` type (`procID 4`) (with a close box but no scroll bars).

   You can leave the rest of the default values as they are.

6. Click OK to close the **View Info** dialog box.

Note

Throughout this tutorial, you are asked to give names (identifiers) to various visual elements. It is extremely important that the names you type match, in spelling and case, those given in the tutorial. If any of the names are not the same, the pre-edited source files provided for you will not compile properly.

You can now see how your window will look when running the application.

7. Choose **Try Out** from the **View** menu (Command-Y). The window should look similiar to the one in Figure 6-5.

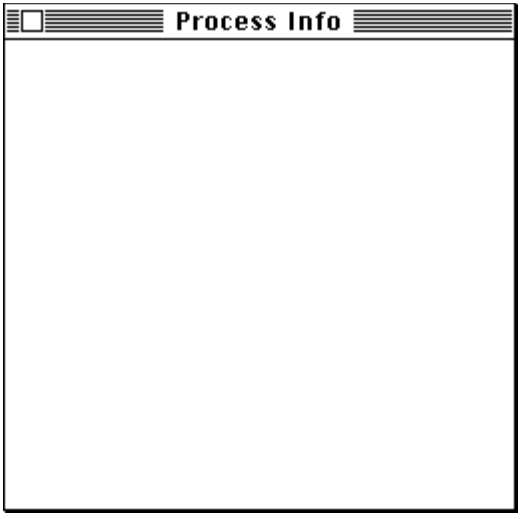


Figure 6-5 Previewing the Process Info view window

8. Close the window after examining it to see if it is correct, by either clicking its close box or choosing **Close** from the **File** menu (Command-W).

   You should now set the size of the Process Info window.

9. Use the size box in the Main view edit window to extend your window's size to approximately a six-inch square.

10. Resize the gray rectangle (the window's `portRect`) to fit snugly within the window using the sizing handle at its lower-right corner. If you drag the size box to the edges of the window, the view scrolls automatically.

Drawing rectangles

It will be more convenient to have the Tool Palette handy throughout the tutorial, so you should tear it off and place it next to the Main view edit window.

1. Click the **Tools** menu and drag the mouse beyond the edge of the Tool Palette. You can now place the menu's outline wherever you want it to be.

   Figure 6-6 shows the Tool Palette identifying callouts for the tools you will use in this tutorial.

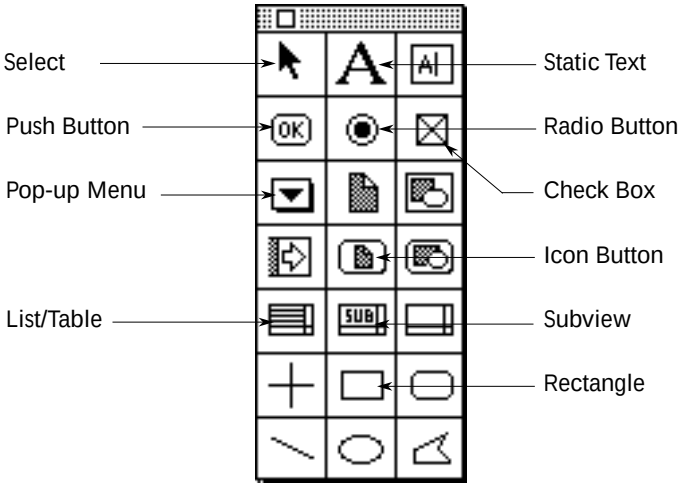


Figure 6-6 The Tool Palette with callouts for tools used in this tutorial

   The six tools at the bottom of the Tool Palette look and act like the standard drawing tools found in Macintosh draw programs. Using the Rectangle tool, draw three rectangles in your main window.

2. Choose the Rectangle tool from the Tool Palette. Double-clicking the tool causes it to stay selected until you choose a different tool.

3. Create three rectangles in your Main view edit window by clicking where you want to place the upper-left corner of each rectangle and then dragging until it's the right size.

4. Size and position your rectangles so the Main view edit window resembles the one in Figure 6-7.

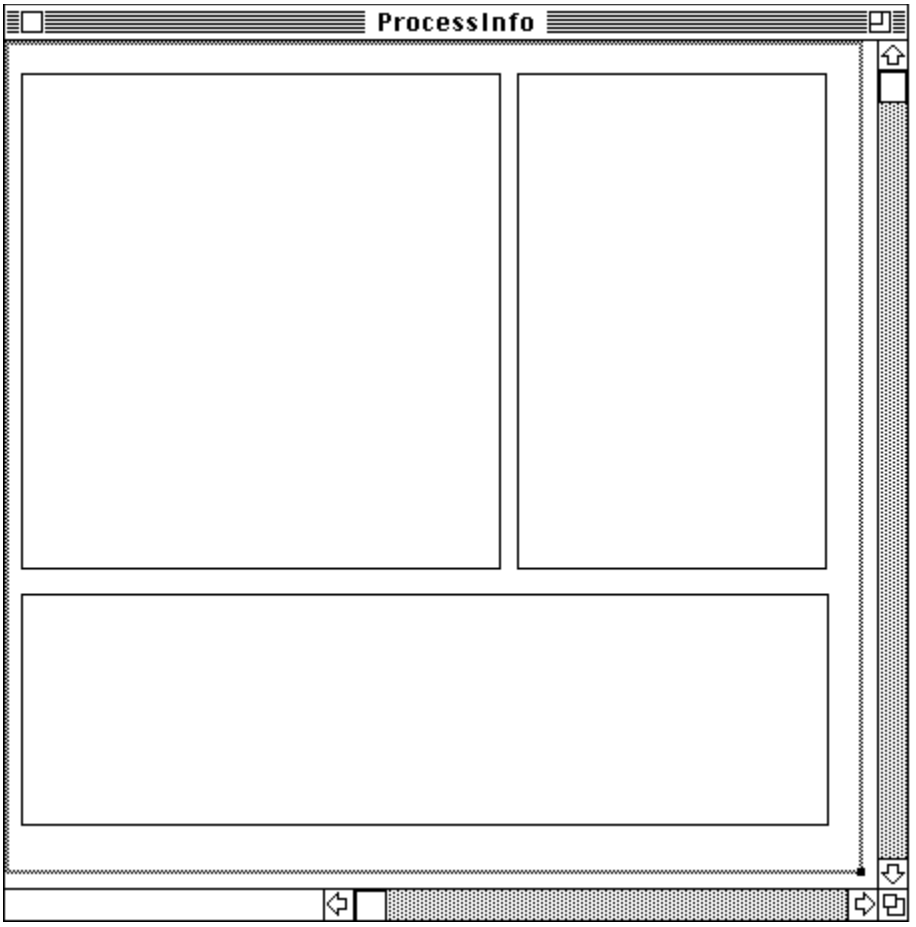


Figure 6-7 The Main view edit window

The rectangles serve as decorative elements and delineate functional areas in the main window. They can also be activated to serve as buttons or used to mask off certain active controls. You will explore the latter concept later.

Creating static text items

Note

As you work through this part of the tutorial, you may want to look ahead to Figure 6-23, which shows you how the finished window appears.

Give the upper-right rectangle a title.

1. Select the Static Text tool from the Tool Palette, click near the top of the upper-right rectangle, and type in the words `Process Control`.

2. Click in the window to end the typing task.

Note

If you need to edit the text after ending the typing task, select the text and press Return.

3. Select the text and move it to the location where it looks best.

Note

Rather than resize or move the text graphically, you can choose **Pane Info** from the **Pane** menu box and edit the values at the top of the dialog (Command-L).

Creating push buttons

The top-right rectangle contains three push buttons of the CIconButton class. This button type supports multiple activation states. The first push button causes the program to drop into the debugger.

1. Select the Icon Button tool from the Tool Palette and click toward the upper-left edge of the upper-right rectangle.

   The button appears with a default icon; however, some more interesting ones have been copied into the resource file. You can select one of these icons instead of the default one.

2. Choose **Pane Info** from the **Pane** menu (Command-L), or double-click the bottom pane.

   The Pane Info window, similar to the one shown in Figure 6-8, appears. Note that the value in the Identifier field may vary.

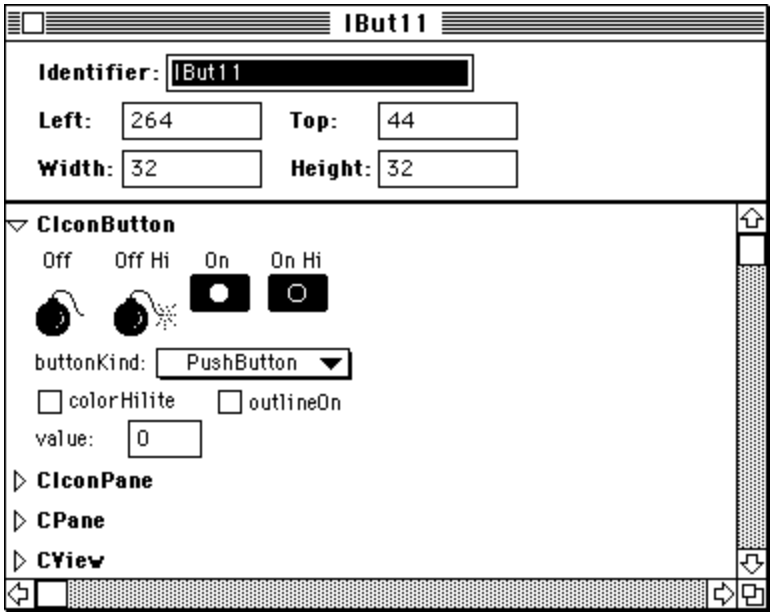


Figure 6-8 The Pane Info window for the Debugger button

3. Open the CIconButton class by clicking the triangle to the left of the class name.

   As you can see, there are four possible states for a CIconButton: Off, Off Hi, On, and On Hi, and each of these can have its own icon.

4. Click the icon representing the Off position.

An **Icon Pick** dialog box comes up, showing all the icons in the `Visual Architect.rsrc` file and thus are available for your use, as shown in Figure 6-9.

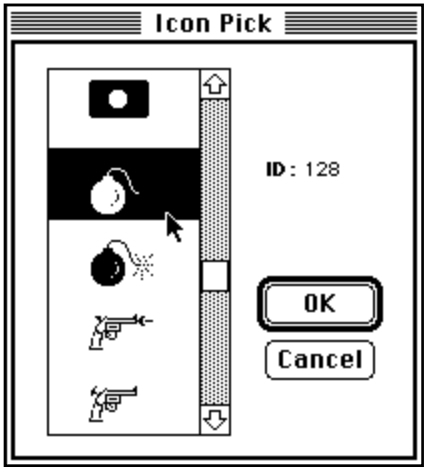


Figure 6-9 The Icon Pick dialog box

5. Click the bomb symbol with the fuse unlit (ID #128) and click OK.

6. Repeat the previous two steps for the Off Hi button state using the icon of the bomb with the lit fuse (ID #129). The On and On Hi states are not used for this type of push button.

   Define the behavior of the button by setting the value of the `buttonKind` data member.

7. Choose PushButton from the buttonKind pop-up menu.

Now it's time to make the button actually do something.

Associating commands with buttons

First, set the current command associated with the Debugger button.

1. Open the CIconPane class by clicking the triangle to the left of the class name. Note that the Command field is currently set to `cmdNull`.

   You need the Debugger button to perform a command that doesn't yet exist in the list. It will be a custom

command called `cmdEnterDebugger` that invokes the debugger.

Later, code will be provided to perform the actual command. For now, however, the command name can be added to the project so a framework can be generated for the code.

2. Choose Other from the Command pop-up menu (at the top), as shown in Figure 6-10.

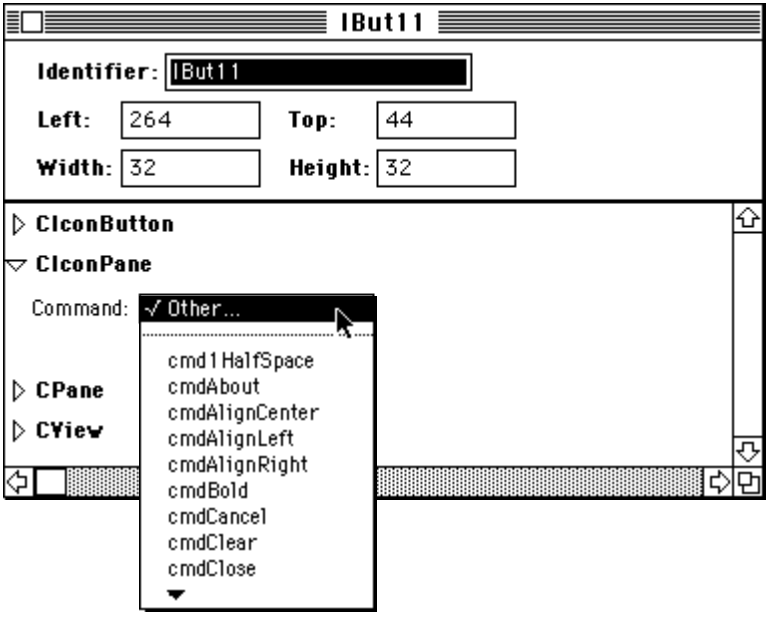


Figure 6-10 Choosing Other from the Command pop-up menu in the Pane Info window

The **Commands** dialog box appears, similar to the one shown in Figure 6-11.

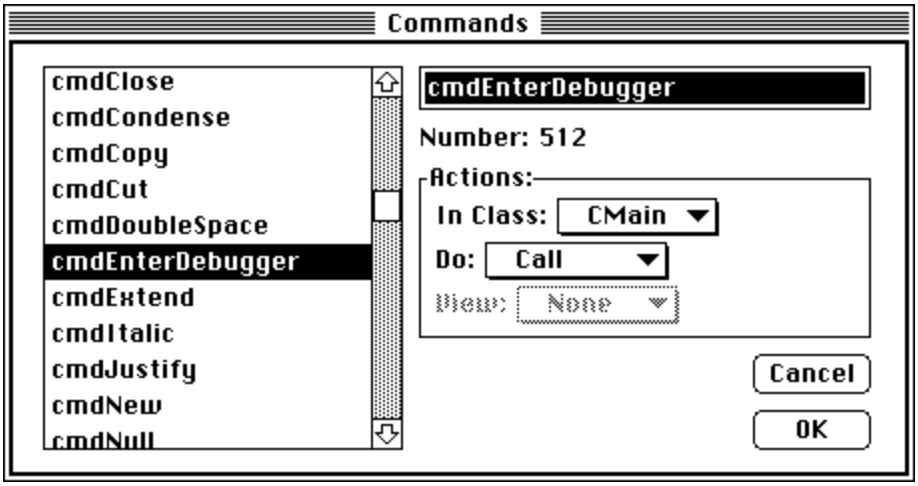


Figure 6-11 The Commands dialog box

3. Choose **New Command** from the **Edit** menu (Command-K) or press Return to activate the edit text field, then type in your new command name, `cmdEnterDebugger`.

   You now need to tell Visual Architect that you want a command handler to be generated in the class CMain for the `cmdEnterDebugger` command.

4. Choose CMain from the In Class pop-up menu.

5. Choose Call from the Do pop-up menu.

6. Click OK to close the **Commands** dialog box.

   You are back in the Pane Info window for the Debugger button. Verify that the Command pop-up in CIconPane is set to `cmdEnterDebugger`.

7. Click the close box to leave the Pane Info window.

Assigning identifiers

Visual Architect gives all the user interface elements you create a unique name, but it is easier to keep track of them if you give them more meaningful names. For example, name the button you just created DebuggerButton.

1. Choose **Identifier** from the **Pane** menu (Command-J).

   The **Identifier** dialog box appears, as shown in Figure 6-12.

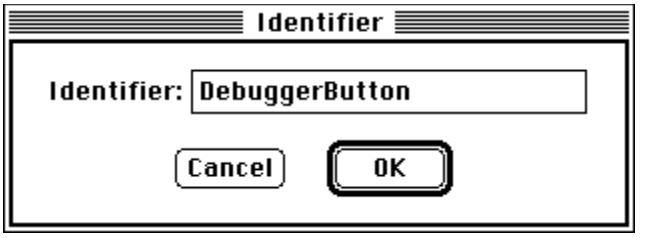


Figure 6-12 The Identifier dialog box

2. Type `DebuggerButton` in the Identifier field and click OK.

Note

You can also set the identifier of a pane directly in the **Pane Info** dialog box.

You also need to do this with the rest of the elements you create, with the exception of simple components such as decorative rectangles and static text items. This way, names in the code that you generate match those in the source files provided in the tutorial folder.

The second push button in the group is for killing the currently selected process in the list. To create this button, repeat the steps in "Creating push buttons," "Associating commands with buttons," and "Assigning identifiers," with the following departures:

- Creating push buttons, step 1: Position this button just under the first button.
- Creating push buttons, step 5: Click the resting gun icon (ID #131) for the Off state.

- Creating push buttons, step 6: Click the shooting gun icon (ID #130) for the Off Hi state.
- Associating commands with buttons, step 3: Name the new command `cmdKillProcess`.
- Assigning identifiers, step 2: Set the identifier to `KillButton`.

The third push button in the group brings the selected process window to the front. To create this button, you again need to follow the steps for creating the Debugger button, with the following departures:

- Creating push buttons, step 1: Position this button just under the second button.
- Creating push buttons, step 5: Click the smiling sun face behind the square icon (ID #132) for the Off state.
- Creating push buttons, step 6: Click the smiling sun face in front of the square icon (ID #133) for the Off Hi state.
- Associating commands with buttons, step 3: Name the new command `cmdBringToFront`.
- Associating commands with buttons, step 4: Choose CApp (not CMain) from the In Class pop-up menu.
- Assigning identifiers, step 2: Set the identifier to `BringToFrontButton`.

You need the `cmdBringtoFront` command to be handled by the application (CApp), because windows other than CMain (Process Info) support `cmdBringToFront`. (This is not true of the first two commands.)

This CIconButton class has no text associated with it, as a check box or radio button does, so you should create a static text element that goes next to each of the icon buttons.

1. Using the Static Text tool, create a static text pane next to each of the three buttons, and type in the names `Enter Debugger`, `Kill Process`, and `Bring to Front`, respectively.

You have completed your three push buttons and set up commands (and handlers) for each of them. You can now try out the buttons.

2. Choose **Try Out** (Command-Y) from the **View** menu.

   A preview window appears as before, but now there are active buttons. When you're finished previewing, close the window:

- Either click the window's close box or choose **Close** (Command-W) from the **File** menu.

Positioning objects

You now need to properly position the user interface elements you created within the Process Info view. A constraint grid has been activated by default to help you align your objects in straight rows and columns.

1. Verify that **Honor Grid** in the **Options** menu is checked.

   Position each of the three buttons and static text objects in turn.

2. Select the object using the Select tool.

3. Drag the button to its new position.

   There are two additional methods for positioning elements. You can do either of the following:

   - Move the selected element one grid step in any direction by pressing the appropriate arrow key (up, down, left, or right).

   - Position and resize the selected element by editing its data members directly in the Pane Info window, which is opened by choosing **Pane Info** from the **Pane** menu (Command-L). The left, top, width, and height values control the position and size of the element within its enclosing pane.

   Changes made graphically are reflected in the values of the data members, and vice versa.

As a further refinement of the objects' placements, you can have them aligned automatically.

4. Select the three buttons using the Select tool by clicking one button, then shift-clicking the other two.

5. Choose **Left** from the **Align** hierarchical menu in the **Pane** menu.

Repeat steps 4 and 5 for the static text objects.

Creating check boxes

In the bottom rectangular section of the main window, you will create a graphical representation of the flag settings in the `'SIZE'` resource of the process currently selected in the process list. First give a title to the bottom rectangle, as you did for the upper-right rectangle.

1. Using the Static Text tool, create a static text object at the top-left corner of the bottom rectangle and type `Size Resource` to name it.

   You need to create the check boxes that display the state of each flag. You should create 11 check boxes in two columns, as shown in Figure 6-13.

Figure 6-13 The Size Resource check boxes

2. Double-click the check box tool so the tool remains selected continuously.

   Repeat the following three steps for each check box:

3. Click the cursor approximately where you want the check box positioned.

The text field for the button is opened automatically when you place the check box.

4. Type in the appropriate label for the check box from Table 6-1.

5. Set the identifier for the check box by choosing **Identifier** from the **Pane** menu (Command-J), typing in the appropriate identifier from Table 6-1, and clicking OK. Remember that spelling is important.

Check box label	Identifier
TextEdit Services	TEServices
Stationery Aware	Stationery
Local & Remote HL Event	LocalRemoteHL
32 Bit Compatible	Is32Bit
Child-Died Events	ChildDied
High Level Event Aware	HighLevelAware
Get Front Clicks	FrontClicks
Background Only	BOnly
Multifinder Aware	Multifinder
Runs in Background	RunsInBackground
Suspend/Resume Events	SuspendResume

Table 6-1 Check box labels and identifiers

Here's an example of how the decorative rectangles discussed earlier can be given a more active role in your user interface. By default, the `wantsClicks` attribute (in the CView part of the class) is set to `TRUE`. However, these particular check boxes are for display only—you don't want the user to be able to click them and change the state of the check box. One way to prevent this is to go into the Pane Info window for each of the 11 check box items and change `wantsClicks` to `FALSE`. A quicker way is to set the rectangle surrounding the check boxes to ignore mouse clicks, and place it *on top of* all the check boxes.

1. Select the rectangle surrounding the check boxes using the Select tool.

2. Choose **Bring To Front** from the **Pane** menu.

3. Choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window.

4. Open the CView class and verify that the check box for the data member `wantsClicks` is not checked.

5. Close the Pane Info window.

   Now the rectangle acts like a transparent sheet of glass over the check boxes, allowing them to be seen but not touched.

   As a final step, bring the rectangle's label in front of the rectangle.

6. Select the static text item "Size Resource."

7. Choose **Bring To Front** from the **Pane** menu.

Creating a subview

The remaining rectangular area in the main window is going to contain a subview. First, give it a title.

1. Using the Static Text tool, create a static text object at the top-left corner of the upper-left rectangle and type `Process List` to name it.

   Next, create the subview.

2. Select the Subview tool and create a subview just inside the rectangle's border, as shown in Figure 6-14.

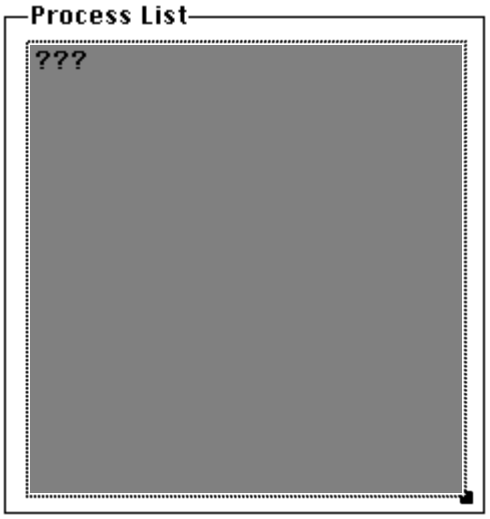


Figure 6-14 Creating the Process List subview

Note the shading of the subview area; it's clearly not an ordinary rectangle. You can verify this by examining the Pane Info window for the subview.

3. Choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window and open the CSubviewDisplayer class.

   The CSubviewDisplayer's Subview contains question marks (???) because no subview has yet been instantiated. A subview is an entirely new view, so you must first create the view.

4. Close the Pane Info window.

5. Choose **New View** from the **View** menu to open the **New View** dialog box.

6. Choose Subview from the View Kind pop-up menu and name the subview `ProcessListSubview`, as shown in Figure 6-15.

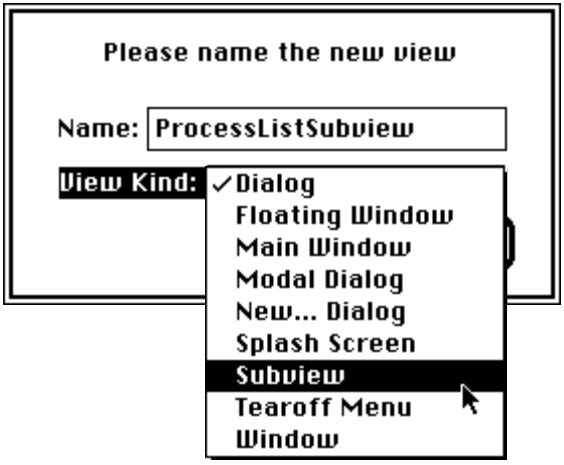


Figure 6-15 Creating a subview using the New View dialog box

7. Click OK to close the **New View** dialog box.

   A new editing window opens, into which you can add the subview's elements. First, however, you'll want to have CMain's subview pane refer to this new subview.

8. Choose **ProcessInfo** from the **Windows** menu.

9. Select the unnamed subview in the Process List area using the Select tool.

10. Choose **Identifier** from the **Pane** menu (Command-J), name the subview `ProcListSubview`, and click OK.

11. Once again, choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window and open the CSubViewDisplayer class.

12. Choose ProcessListSubview from the Subview pop-up menu and close the Pane Info window.

The subview is now properly incorporated into the Main view.

Creating a scroll table

This subview contains a list of processes in a scrolling table, with a pop-up menu, to allow a user to choose how to display the processes. First return to the subview's own editing window. To begin editing the process list subview, bring its window to the front.

1. Choose **ProcessListSubview** from the **Windows** menu.

2. Select the List/Table tool and click-drag in the left side of the Subview window to create a scroll table pane, as shown in Figure 6-16.

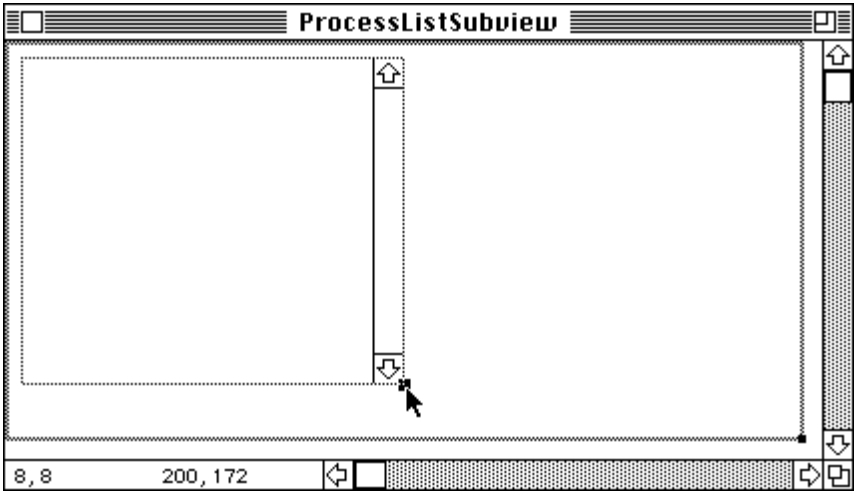


Figure 6-16 Creating a scroll table in the ProcessListSubview view

3. Choose **Identifier** from the **Pane** menu (Command-J) and type `ProcessTable`.

   When the table pane is selected, the **Scrollpane Info** item in the **Pane** menu becomes enabled. Although you won't need to change any of the values in CScrollPane now, you can verify that the **Scrollpane** item is enabled. To do this:

4. Select **ScrollPane** from the **Pane** menu to open the **Scrollpane Info** dialog box, and open the CScrollPane class.

5. Close the **Scrollpane Info** dialog box.

Setting the table command

The **Pane Info** window for the scrollpane brings you to the data members for the CTable item contained within the scrollpane, not to the scrollpane itself.

1. Choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window, and open the CTable class.

   You can set the double-click command for the table; this is the command that executes whenever a user double-clicks an item in the table.

2. Scroll down to the Command pop-up menu and choose cmdBringToFront from the list, as shown in Figure 6-17.

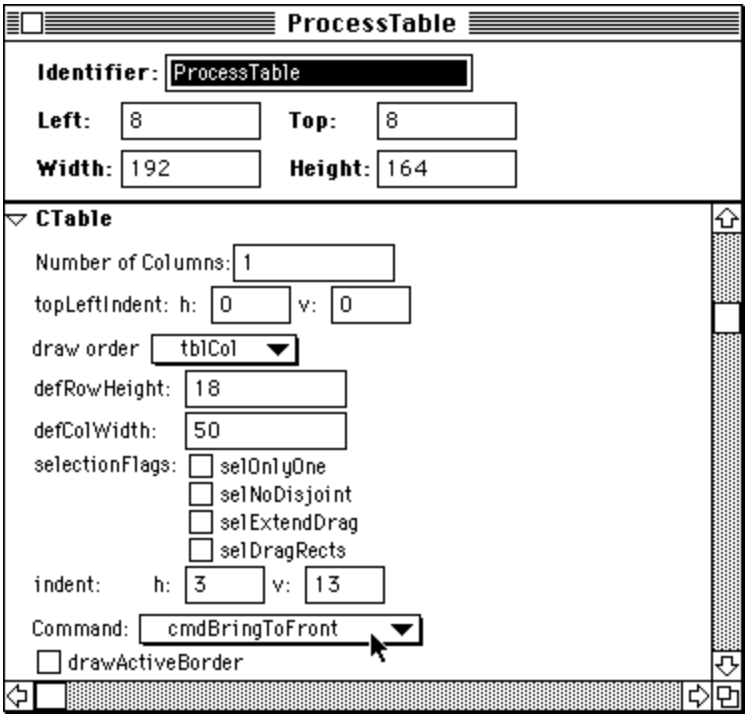


Figure 6-17 The Pane Info window for the ProcessTable scrollpane

3. Close the Pane Info window.

When the user double-clicks a process in the process list, that process becomes the foreground application.

Creating a derived class

Because you want to give your table pane the functionality to display a list of processes, you need to make it a derived class of CArrayPane.

1. Choose **Classes** from the **Edit** menu to open the **Classes** dialog box.

2. Enter a new class either by choosing **New Class** from the **Edit** menu (Command-K) or by clicking under the list of existing classes and typing `CProcessArrayPane` as the name of the class, as shown in Figure 6-18.

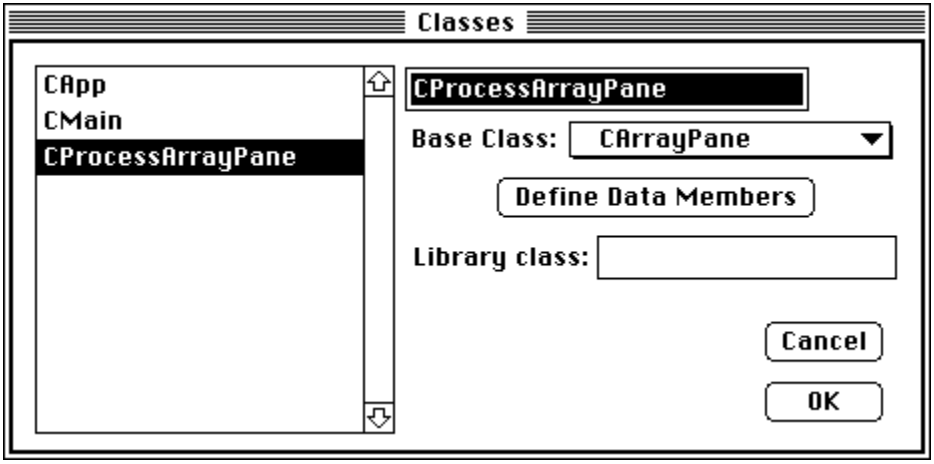


Figure 6-18 The Classes dialog box

3. Choose CArrayPane from the Base Class pop-up and click OK to close the **Classes** dialog box.

   You now need to set the CProcessArrayPane class as the one of which the scroll table pane is an instance.

4. Choose **CProcessArrayPane** from the **Class** hierarchical menu in the **Pane** menu, as shown in Figure 6-19.

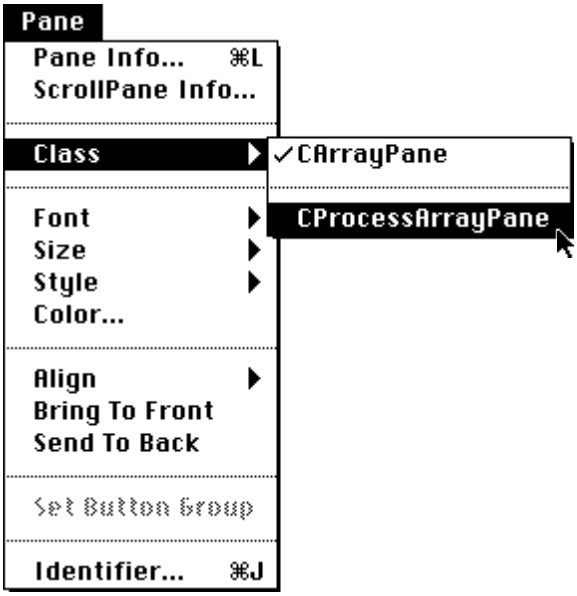


Figure 6-19 Setting the class of the scroll table pane

Now when you open the Pane Info window for the table pane, CProcessArrayPane appears in the class hierarchy.

You also may want to extend the functionality of the whole subview (to handle certain commands from a pop-up menu). To do this, you must create a new subview class derived from CPanorama.

5. Choose **Classes** from the **Edit** menu.

6. Enter a new class either by choosing **New Class** from the **Edit** menu (Command-K) or by clicking under the list of existing classes, and typing `CProcListSubviewPanorama` as its name.

7. Choose CPanorama from the Base Class pop-up and click OK to close the **Classes** dialog box.

8. Click somewhere on the ProcessListSubview view that doesn't contain the scrollpane.

9. Choose **CProcListSubviewPanorama** from the **Class** hierarchical menu in the **Pane** menu.

Now the entire subview is actually a CProcListSuviewPanorama into which you can add functionality.

Creating a pop-up menu

You will now create a pop-up menu in the subview.

1. Select the Pop-up Menu tool and click beneath the table pane, on the left side.

   Note that the pop-up menu already has default values entered in it. You need to put your own menu values in it.

2. Choose **Menus** from the **Edit** menu to bring up the **Menus** dialog box, similar to the one shown in Figure 6-20.

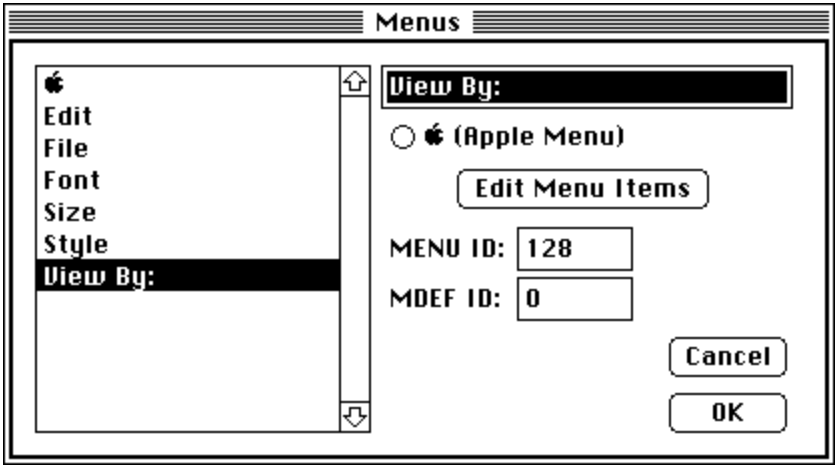


Figure 6-20 The Menus dialog box

   Note that the normal default menu types (for example, File and Edit) already exist, as does the pop-up menu you just added, generically titled Popup Menu. You should now name this menu, as well as the items within it.

3. Click the Popup Menu item and type `View By:` into the text edit field as the name of your menu.

4. Click the Edit Menu Items button to bring up the **Menu Items** dialog box.

5. Change the three item names to `Name`, `Serial Number`, and `Signature` by clicking them and then typing the text.

6. Select Name from the menu item list and choose the check mark item in the Mark pop-up menu, as shown in Figure 6-21.

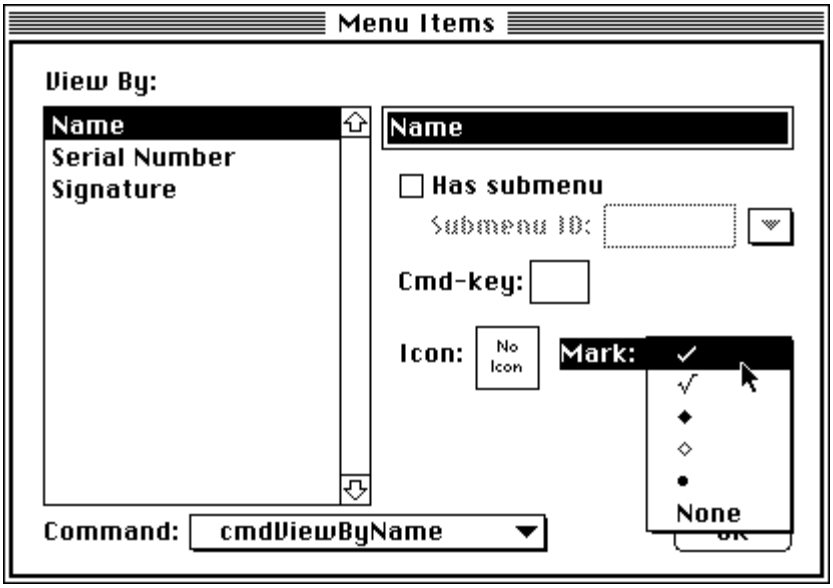


Figure 6-21 Choosing the check mark in the Menu Edit dialog box

This last step ensures that **Name** appears as the default choice in the View By pop-up menu.

Associating commands with a menu

To associate commands with each of the pop-up menu choices, first these commands have to be created, as did the icon buttons earlier.

1. Click the **Name** item to select it.

2. Choose Other from the Commands pop-up menu to bring up the **Commands** dialog.

3. Press Return to create a new command and type `cmdViewByName` to name it. Choose CProcListSubviewPanorama from the In Class pop-up menu, and choose Call from the Do pop-up menu.

4. Click OK to close the **Commands** dialog box.

5. Repeat steps 1 through 4 for the Serial Number and Signature items, naming their commands `cmdViewByPSN` and `cmdViewBySignature`, respectively.

6. Click OK to close the **Menu Items** dialog box.

7. Click OK to close the **Menus** dialog box.

Now the CProcListSubviewPanorama can handle any of the three menu selections.

Finishing the pop-up menu

You're almost finished with the pop-up menu. You now need to ensure that only one of the three menu items can be selected at a time.

1. Choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window, and open the CPopUpPane class.

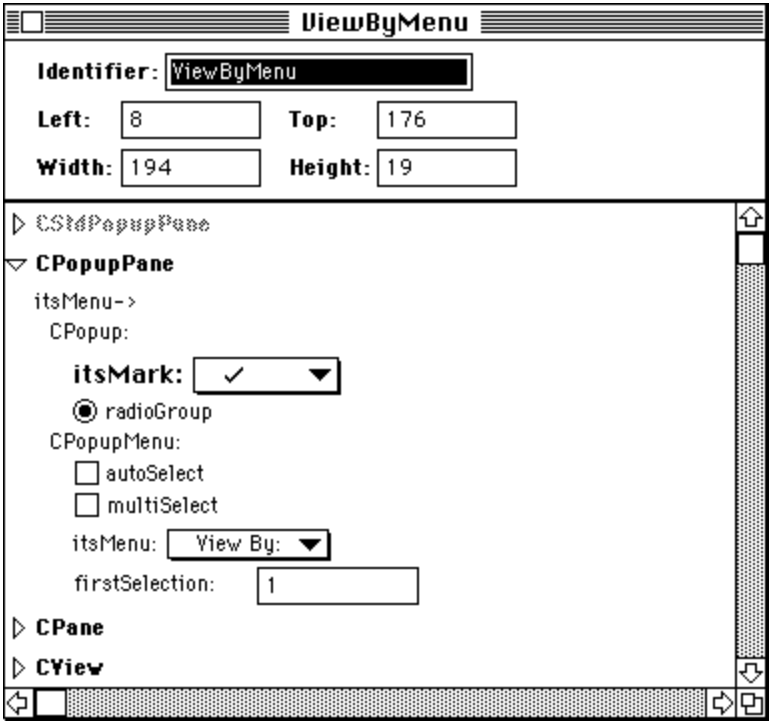


Figure 6-22 Clicking radioGroup in the Pane Info window

2. Click the Radio Group radio button, as shown in Figure 6-22, and close the Pane Info window.

3. In the Identifier field at the top of the window, type `ViewByMenu` and close the Pane Info window.

4. Close the ProcessListSubview edit window.

Trying out the completed main window

Your Main view window should now appear as shown in Figure 6-23.

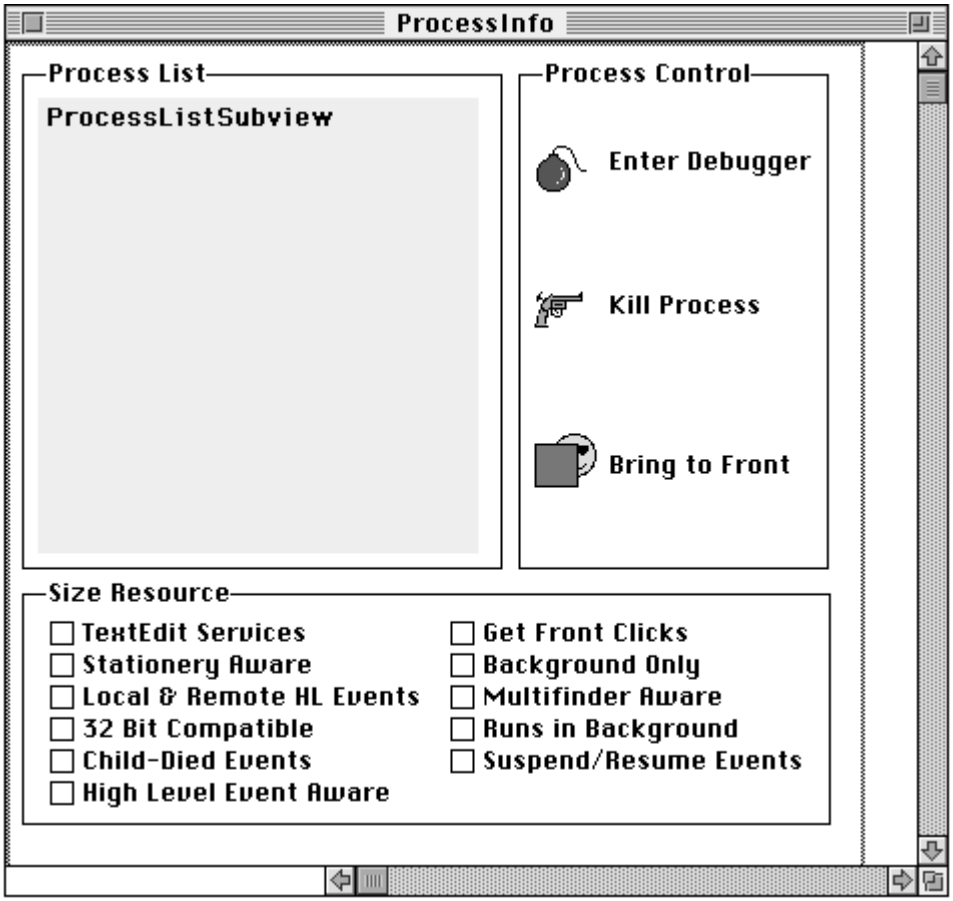


Figure 6-23 The completed Main view window

You can now preview how the main window will look in the running application.

1. Choose **Try Out** from the **View** menu (Command-Y).

The main window comes up as it will in the running application and lets you manipulate the controls, as in Figure 6-24.

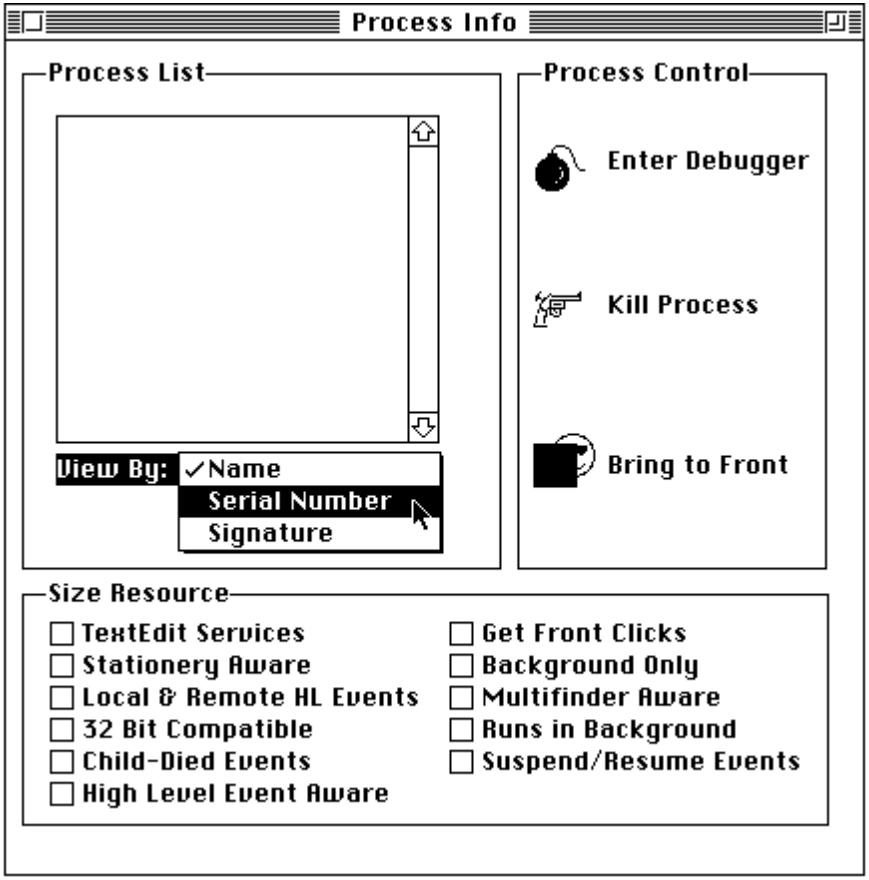


Figure 6-24 Previewing the Main view window

2. Close the preview window after verifying that the controls function properly.

Creating the alternative main window

The main window you just finished shows several kinds of process information, in addition to a subview containing a list of processes. The next view you create is an alternative window containing only the process list. It will be another main window.

1. Choose **New View** from the **View** menu to open the **New View** dialog box.

2. Choose Main Window from the View Kind pop-up menu and type `ProcessListOnly` as the view name, as shown in Figure 6-25.

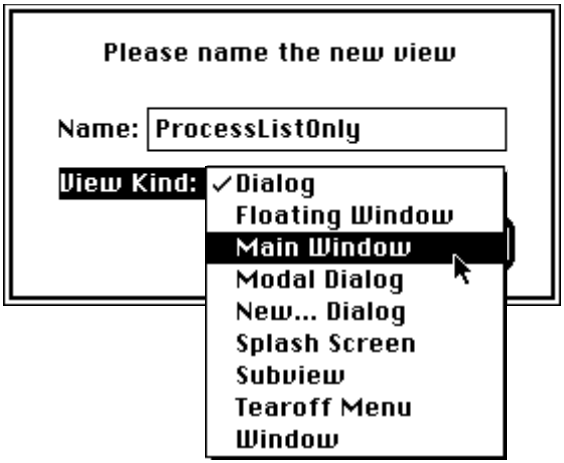


Figure 6-25 Creating a Main view using the New View dialog box

3. Click OK to close the **New View** dialog box.

   The ProcessListOnly view edit window appears.

4. Choose **View Info** from the **View** menu to bring up the View Info dialog box.

5. Type `Process List` as the view window title. (Leave the view's name as is.)

6. Make the view's window the same kind of window as the ProcessInfo window, that is, a window with no scroll bars or size box (the second window type from the left), and click OK.

   This view is going to have the same subview within it that the main window has.

   As before, you should create a subview in the window.

7. Select the Subview tool and create a subview just inside the gray-outlined rectangle within the window, as shown in Figure 6-26.

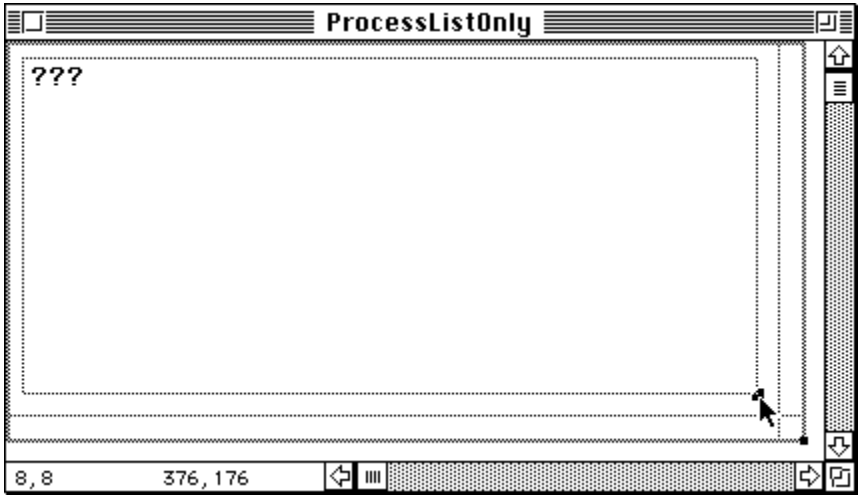


Figure 6-26 Creating the subview in the ProcessListOnly view

8. Choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window. Open the CSubviewDisplayer class, choose ProcessListSubview from the subview pop-up menu.

9. Type `ProcListSubview` in the Identifier field and close the Pane Info window.

   Because the subview handles its own commands, there's nothing left to do with respect to assigning command handlers for this window.

   Now the alternative main window, CProcessListOnly, is complete. You're ready to create the final view.

10. Close the ProcessListOnly view edit window.

Creating the New… Dialog

The dialog that comes up when a user chooses **New** from the application's **File** menu is called a **New** dialog.

1. Choose **New View** from the **View** menu to open the **New View** dialog box.

2. Choose New… Dialog from the View Kind pop-up and type `WindowChooser` as the view's name, as shown in Figure 6-27.

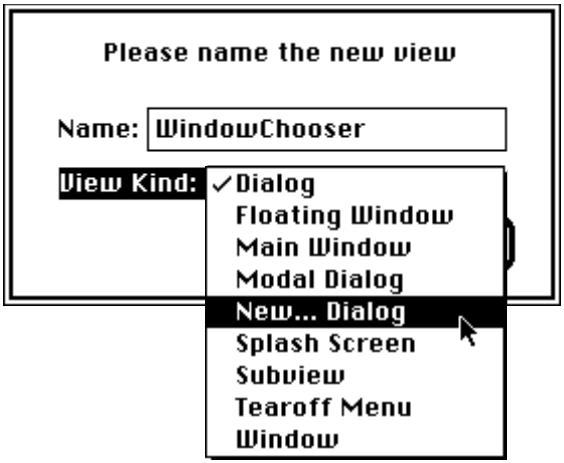


Figure 6-27 Creating a New… Dialog view using the New View dialog box

3. Click OK to close the **New View** dialog box.

The WindowChooser view edit window appears. By default, the dialog is a double-bordered modal dialog window.

Adding radio buttons

You need to add two radio buttons that let the users choose which of the two views they want to create.

1. With the Static Text tool, create a static text item at the top of the dialog and type the text `Create a window showing:`.

2. With the Radio Button tool, create two radio buttons underneath the static text item, one above the other, and give them the labels `Process List and Information` and `Process List Only`.

3. Select each radio button in turn, choose **Identifier** from the **Pane** menu (Command-J), type `ProcListAndInfo` and `ProcListOnly`, respectively, as identifiers, and click OK.

4. Select the Process List and Information radio button, choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window, and open the CControl class.

5. Set contrlValue to 1, as shown in Figure 6-28, so this button will be selected by default, and close the Pane Info window.

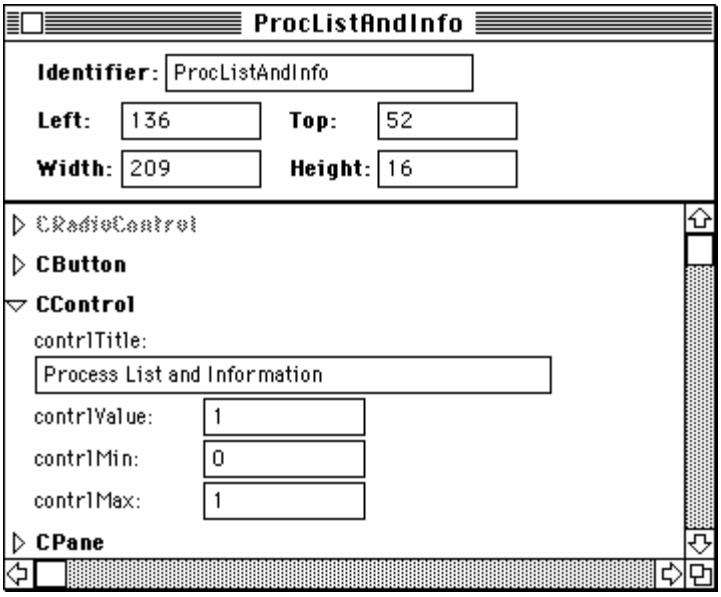


Figure 6-28 The Pane Info window for the radio button

Adding OK and Cancel buttons

1. Select the Button tool and click the lower-right corner of the dialog to create an OK button.

2. Choose **Identifier** from the **Pane** menu (Command-J), type `OK`, and click OK.

   By default, the first button you add to a view is named OK and has `cmdOK` as its command.

3. Select the Button tool again and click to the left of the first button to create a Cancel button.

4. Choose **Identifier** from the **Pane** menu (Command-J), type `Cancel`, and click OK.

By default, the second button you add to a view is named Cancel and has `cmdCancel` as its command.

When you are finished with the buttons, the result should resemble Figure 6-29.

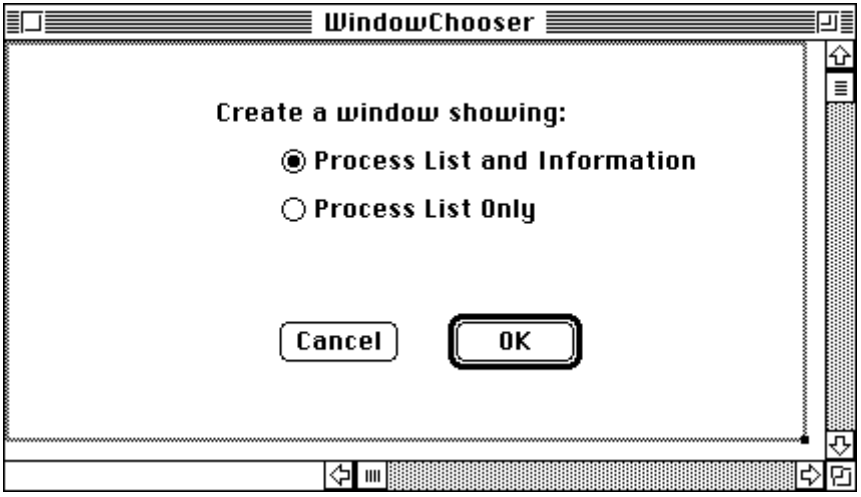


Figure 6-29 The New… Dialog view edit window

Editing menus

The last thing to do before saving your project is to customize the menu bar for your application. In this case, there are only a few interesting things you can do from a menu while running the application, so you should remove menu choices that are not supported.

1. Choose **Menu Bar** from the **Edit** menu to open the **Menu Bar** dialog box.

2. Select the **File** menu from the list and click the Edit Menu Items button to bring up the **Menu Items** dialog box.

3. Delete all menu items except New, Close, and Quit by selecting them from the list and pressing Delete. You should keep the dividers between the three menu items in place.

4. Click OK to close the **Menu Items** dialog box.

5. Select the **Apple** menu in the menu list and click the Edit Menu Items button to bring up the **Menu Items** dialog box.

6. Choose the top menu item in the list, type `About Process Monitor…` to rename it, and click OK.

7. Click OK in the **Menu Bar** dialog box to close it.

   You've finished using Visual Architect to design your user interface, and you can now save the `Visual Architect.rsrc` file.

8. Choose **Save** from the **Edit** menu (Command-S).

Generating Code for Your Application

Having built the user interface for the tutorial application, you are now ready to give it life, that is, to generate code that enables it to run.

The code falls into two main categories. One supports the application's function (the display and manipulation of running processes), and the other supports the user interface. It is the latter that Visual Architect generates; the functional code still needs to be written by the programmer. This code is provided for you in the `Extra Sources` folder and in the pre-edited files of the `Source` folder.

1. Choose **Generate All** from Visual Architect's **THINK Project Manager** menu (the one that looks like the THINK Project Manager application icon), as shown in Figure 6-30.

Figure 6-30 The THINK Project Manager menu in Visual Architect

A dialog box appears showing the progress of the code generation, as in Figure 6-31.

Figure 6-31 The Code Generation Progress dialog box

By default, Visual Architect generates code to the `Source` folder within the `Process Monitor` folder, using standard macro files. After Visual Architect creates the files, it adds them automatically to your project. When generation is complete, the folder in your `Process Monitor` folder called `Source` contains the generated code.

2. Switch to the THINK Project Manager application, scroll the Process Monitor.π project window to the end, and open the Application segment by clicking the small triangle to the left of its name.

Look at the project window to verify that Visual Architect has added the new source files to your project in the Application segment (this will be the last segment of your project), shown in Figure 6-32.

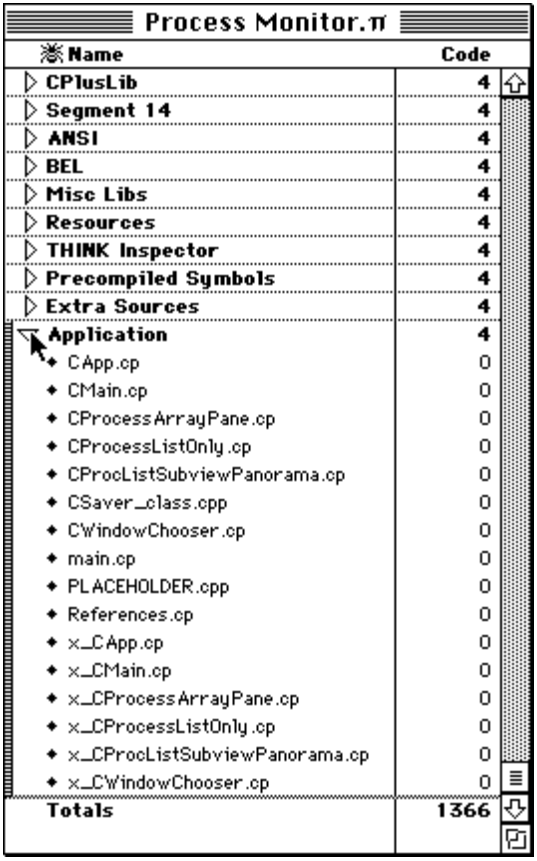


Figure 6-32 The Process Monitor.π project window showing the source files within the Application segment

Customizing your code

At this point, Visual Architect has generated the user interface code. This is normally the time at which you would add your functional code. As mentioned earlier, however, the functional part has been added for you. To get an idea of the changes made, you can look at the source files directly. Edited portions of changed source files are marked clearly so you can distinguish easily between generated and customized code. The edits are preceded and followed by the comments shown following:

```
// ••••• Visual Architect Tutorial Code Change Begin •••••
```

< modified code goes here>

```
// ••••• Visual Architect Tutorial Code Change End •••••
```

In addition, all files in the `Extra Sources` folder were created for the tutorial. These files implement classes that don't have any visual elements, so they can't be specified by Visual Architect. These files have already been added to your project.

Running the application

You first need to compile the project.

1. Choose **Bring Up to Date** from the **Project** Menu.

   If that was successful, run the application.

2. Choose **Run** from the **Project** menu.

If you get compile errors, it's likely that you have misspelled one of the identifiers or forgotten to set an identifier where one is needed. If so, go back to Visual Architect to correct the mistake, then choose **Run** from Visual Architect's **THINK Project Manager** menu. Files that need to be are regenerated automatically.

Once the program is running, experiment as you like. However, keep in mind the following notes:

- If you try to kill a process whose Runs in Background box is unchecked, remember that it won't die until that process is brought to the foreground. This is because the application won't process the quit event until it gets a chance to execute its event loop. Try killing Visual Architect for an example of this.

- Some processes in the list may not respond to the Bring to Front button. Look at the Background Only bit in the Size Resource section. If it's checked, then the selected process doesn't have a user interface and can't be queued as the foreground application.

- If you haven't installed a low-level debugger such as MacsBug, be sure you run the tutorial under the THINK Debugger before trying the Enter Debugger button.

Where to Go Next

This tutorial demonstrated some of the more advanced techniques used in working with Visual Architect. You should now proceed with Chapter 3, “THINK Class Library Basic Concepts,” and Chapter 4, “Visual Architect Basic Concepts,” to begin developing your own applications.

THINK C Symantec C++

Using the THINK Class Library

Part Two

Programming with the THINK Class Library 7

This chapter explores aspects of the THINK Class Library that you encounter in the course of building an application. No topic is covered exhaustively here; the class descriptions in the final part of this book provide the definitive reference. Nevertheless, the discussions are sufficiently detailed to give you a sound understanding of how the Library works and of how to use it to build a robust application. This chapter assumes that you have read those that precede it and that you have worked through the tutorials. It begins by reviewing material from Chapter 3, "THINK Class Library Basic Concepts," to provide background for a more in-depth treatment. After reading this chapter, you will know enough about the THINK Class Library to modify and enhance the sample projects, or perhaps to begin a new project.

Contents

Introduction

This chapter discusses the essential features of the THINK Class Library and talks about some of the general topics in application building. After you read this chapter, you should try running and modifying the demonstration applications. Make sure to follow the instructions in Chapter 2 of the *THINK C User's Guide* for installing the THINK Class Library and its demonstration programs.

Naming Conventions

The THINK Class Library follows these naming conventions:

Name	Description
`CName`	The name of a class
`aName`	A formal parameter
`cName`	A static data member
`fName`	A flag, usually a Boolean data member
`gName`	A global variable
`kName`	A constant, defined with `#define`, `enum`, or `const`
`itsName`	A data member
`theName`	A variable, usually a local variable or data member; sometimes used for formal parameters
`macName`	A Macintosh data structure, used either as a data member or as a local variable

Table 7-1 Naming conventions

You should give your classes names that begin with any letter other than C. Adhering to this practice will avoid conflicts with names of classes in future versions of the THINK Class Library.

Previous versions of the THINK Class Library were designed to be implemented in both Symantec C++ and THINK C with Object Extensions. In order to remain backward compatible with THINK C with Object Extensions, THINK Class Library classes still contain explicit initialization and destruction member functions. The old initialization function for a class `CName` is denoted by `IName`; the

old destruction member function is always named `Dispose`. These member functions are now anachronisms, superseded by genuine C++ constructors and destructors. However, their continued presence guarantees that existing code written for earlier versions of the THINK Class Library will not require extensive rewriting.

Fundamental Structures

The THINK Class Library is organized into three distinct, interacting structures: the class hierarchy, the visual hierarchy, and the chain of command. The class hierarchy is the set of all the classes that make up the THINK Class Library. The visual hierarchy describes the organization of all visible entities in a given application. The chain of command specifies which objects in an application get to handle commands, and in what order.

The THINK Class Library uses an object called a switchboard to convert Macintosh events into appropriate calls to member functions. Each application has only one instance of a switchboard object. Another object, called the bartender, converts menu selections into direct commands. The unique bartender object is pointed to by the global variable `gBartender`.

The class hierarchy

The class hierarchy is the set of all the classes in the THINK Class Library, ordered by the derivation relationship. Seeing a diagram of the class hierarchy can help you understand the structure of the THINK Class Library. Using the Browser, you can display the class hierarchy for a particular class, and then print it.

Do not think of the class hierarchy as a functional description of the THINK Class Library. For example, find the CRadioGroup class toward the right in the diagram. You might believe, wrongly, that this class interacts somehow with the CBureaucrat class. Instead, the hierarchy embodies inheritance, which means that objects of derived classes possess all the attributes and behaviors of objects of ancestor classes.

The THINK Class Library contains abstract classes; only instances of their derived classes are ever created. The most important abstract classes in the THINK Class Library are CCollaborator, CBureaucrat, CView, CDirectorOwner, and CApplication.

The visual hierarchy

The visual hierarchy is composed of all the visible objects, or views, in a running application. The relationship of enclosure imposes a hierarchy on views. Except for the desktop, everything that you see on the screen—every view—is enclosed by another view.

At the top of the visual hierarchy is the desktop, an object of class CDesktop. Every THINK Class Library application has a unique desktop object, pointed to by the global variable `gDesktop`. The desktop encloses all of the windows in an application. Each window encloses one or more panes, which may in turn enclose other panes.

Panes can handle user actions such as mouse clicks. When you click with the mouse, the switchboard determines in which pane the click occurred and calls that pane's `DoClick` function.

As descendants of CBureaucrat, views can also be in the chain of command and respond to direct commands.

The chain of command

The chain of command specifies which objects get to handle direct commands and in what order. An object can be in the chain of command if and only if it is a bureaucrat—an instance of a class derived from CBureaucrat. Every bureaucrat thus has a `DoCommand` function, which derived classes can override to handle specific commands.

The supervisor of a bureaucrat is the next object up the chain of command. If a bureaucrat can't handle a direct command, it passes the command on to its supervisor by calling that object's `DoCommand` function. The application is the only bureaucrat that does not have a supervisor. If the application doesn't handle the command, no object will. Every THINK Class Library application has a unique application object, pointed to by the global variable `gApplication`.

The first object that has the chance to handle a command is called the gopher. The current gopher is pointed to by the global variable `gGopher`. Usually, the gopher is a pane in the active window.

A bureaucrat that supervises a window is called a director. This kind of bureacrat is important enough to merit a class, CDirector, derived from CBureaucrat. Directors handle the communication between the visual hierarchy and the chain of command. For instance, when a

window gets an activate event, it calls the `ActivateWind` function of its supervisor, which is by definition a director. The director can then take some action as a result of the window's becoming active.

Another important kind of bureaucrat is the document. A document is a director that has a file associated with it. The CDocument class defines the properties and behaviors of documents. Documents manage the communication between windows, files, and menu commands. The default document class handles common commands such as **Save**, **Save As**, and **Print**. You can think of a document as a file that is viewed through a window. It is better thought of as the essence of a Macintosh application. A document is anything that you can display and manipulate inside a window.

Bureaucrats themselves are descendants of the class CCollaborator. A collaborator is an object that can inform other objects of changes. The collaborator announcing changes is called the provider; those that receive such announcements are called its dependents. For instance, a provider might be a document that displays data in several windows; its dependents might be the windows. If the data changes, the provider lets the dependents know, so the windows can redisplay the data correctly.

The flow of control

The chain of command and the visual hierarchy get their direction from the switchboard. The switchboard gets events from the Macintosh Event Manager and converts the events to member function calls affecting either the chain of command or the visual hierarchy. Functions that affect the chain of command are usually called via the gopher, the first bureaucrat in the chain. Functions affecting the visual hierarchy are called via the active window or the desktop.

When you click with the mouse, the switchboard calls the desktop's `DispatchClick` function. If the click was in the menu bar, the desktop calls a function of the bartender that updates the state of the menus before they appear. Then the bartender gives every bureaucrat in the chain of command the chance to enable or disable, and check or uncheck, any pertinent menu items. The desktop then uses the bartender to convert the menu selection into a direct command and finally calls the gopher's `DoCommand` function.

If the click was in a window, the desktop calls the window's `DispatchClick` function, which eventually results in a call to the

DoClick function of the pane in which the click occurred. The pane's DoClick function can then do whatever is appropriate for the pane. It could even call the DoCommand function of an object in the chain of command.

When the switchboard gets an activate or an update event, it calls the Activate, Deactivate, or Update function of the window. The window then calls the corresponding member function—ActivateWind, DeactivateWind, or UpdateWind—of its director.

When you type, the switchboard calls the DoKeyDown or DoAutoKey function of the gopher. If you hold down the Command key as you type, the switchboard asks the bartender to convert the key into a command, and then calls the gopher's DoCommand function.

If you're running under System 7 or MultiFinder and you bring another application to the foreground, the switchboard calls the Suspend function of the application object (not the gopher), which in turn calls the Suspend function of every director. Similarly, when your application comes to the foreground, the switchboard calls the application's Resume function, which in turn calls the Resume function of every director.

The THINK Class Library treats desk accessories as if they were in their own layer, even if you're not using System 7 or MultiFinder. Thus, your application still gets suspend and resume "events" when you bring up a desk accessory.

Writing an Application with the THINK Class Library

To create an application with the THINK Class Library, you derive classes from existing classes. The classes you need to derive from are: CApplication, CDocument, and various classes derived from CPane. Your application class determines the overall structure of your application. The document class implements file handling for your application, and the pane classes implement how the information in your files appears in the document windows.

Note

Many THINK Class Library pane classes already supply the functionality you want for your application; these you can simply use as is. Usually you derive new pane classes to customize or add to the behavior of existing classes; but you can also derive pane classes that implement completely original user interfaces.

In addition to the derived classes, you also must create a resource file for your project. This resource file must contain the standard THINK Class Library resources as well as your own. These standard resources are in the file `TCL Resources`. When you use the THINK Class Library, make a copy of this resource file, and name it *Project*`.rsrc`, where *Project* is the name of your project. Then add your own resources to the file or project.

Note

For more information about resource files and the THINK Class Library, see "THINK Class Library Resources" later in this chapter. To learn about using resources with a project, see Chapter 9, "Organizing Projects," in the *THINK C User's Guide*.

Creating the application class

For standard Macintosh applications, you must derive an application class from CApplication. Your class must define a constructor and override these member functions:

```
SetUpFileParameters
CreateDocument
OpenDocument
DoCommand
```

Your application class constructor should initialize any new data members that your derived class declares. It must call CApplication's constructor with the appropriate arguments. Among the many initializations it performs, CApplication's constructor sets the global variable `gApplication` to point to your unique application object.

`SetUpFileParameters` sets up the standard file parameters that specify which files are visible in the standard file box when the user chooses **Open** from the **File** menu.

`CreateDocument` creates a new, untitled document. The class of the document it creates is one that you derive from `CDocument`. After creating the document, your `CreateDocument` function should call the document's `NewFile` function. That function gets called when the user chooses **New** from the **File** menu.

`OpenDocument` is similar to `CreateDocument`. Instead of calling the newly created document's `NewFile` function, though, your `OpenDocument` should call its `OpenFile` function. The `OpenDocument` function takes one parameter, an `SFReply` record, which contains the information about the file that the user chose to open.

`DoCommand` handles all application-specific commands. However, usually you find that most commands specific to your program are best handled at the document level. Some commands, like **New**, **Open**, and **Quit**, are already handled by the default application class, CApplication.

Creating the document class

The document is where your application draws and displays its data. All documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file is created automatically. You must create them yourself.

Your document class must derive from CDocument. It must define a constructor and override these member functions:

```
DoCommand
NewFile
OpenFile
DoSave
DoSaveAs
Revert
```

If your document class defines new data members, its constructor should initialize them. Make sure that your constructor calls the CDocument constructor. The supervisor of a document is always `gApplication`.

If your document allocates memory, you should also define a destructor to deallocate it.

Note

You do not need to delete the `itsWindow` or the `itsFile` data members. The virtual destructor `~CDocument` does that for you.

Your document classes' `DoCommand` functions do most of the work in your application. When a window is active, one of its panes is the gopher, and the pane's supervisor is the same as the window's—namely, the document. Thus, the switchboard calls the pane's `DoCommand` function first. But typically, a pane passes a direct command on to its supervisor. Your document class should handle all the commands it knows about, and call `CDocument`'s `DoCommand` when it can't. Performing this last step ensures that the application object's `DoCommand` is called for commands that your document doesn't process.

Your document classes's `NewFile` function is called when the user chooses **New** from the **File** menu. This function must create a window and attach its panes. The `NewFile` function doesn't have to create a file until the user tries to save the document.

Your document's `OpenFile` function is called when the user chooses **Open** from the **File** menu. The `OpenFile` function has one argument: a pointer to a Macintosh `SFReply` record. When the `OpenFile` function is called, you can be sure that the `SFReply` record is properly filled in. Your `OpenFile` function needs to create a file object (usually of class CDataFile). You can call any of several CDataFile reading functions to obtain the file's contents. Your `OpenFile` function also needs to create a window and its attached panes to display the contents of the file, just as your `NewFile` function does.

When the user chooses **Save** from the **File** menu, your document's `DoSave` function is called. Your `DoSave` function should write the contents of its file to disk. The file object is stored in the data member `itsFile`.

When the user chooses **Save As** from the **File** menu, your document's `DoSaveAs` function is called. This function takes an `SFReply` record as argument, which is guaranteed to be properly

filled in. Your document class needs to override this function to write its data to a file.

If your application supports the **Revert** command, you should implement it in the `DoRevert` function. Your implementation might do the same thing as closing without saving, then opening the file again.

Make sure that your `DoCommand` function itself, or one of the functions it calls, sets the flag data member `dirty` to `TRUE` when the document is changed. Your document class's `DoSave`, `DoSaveAs`, and `DoRevert` functions should clear the dirty flag after successfully performing their roles.

Creating the pane classes

Once you've created your windows and opened your files, you need to display them somewhere. In the THINK Class Library, you don't write directly onto the window. Instead, you create pane classes—classes derived, not necessarily directly, from CPane.

A pane class should define a constructor and override these member functions:

```
Draw
DoClick
```

If your pane class defines new data members, its constructor should initialize them. Your constructor should also call the constructor of its base class. The supervisor of a pane should be either the pane that encloses it or the director that its window belongs to.

The pane constructor is where you set the pane's location in its enclosure and its characteristics. If you want your pane to receive clicks, be sure to call `SetWantsClicks(TRUE)`; otherwise, mouse clicks in your pane are ignored.

If your pane allocates memory, you should also define a destructor to deallocate it.

The `Draw` function is where your pane should draw its contents. You can assume that the port, clip region, and coordinate systems have been set up correctly when `Draw` is called. See the section "Drawing in a pane" later in this chapter to learn about drawing in a pane.

When you click a pane, its `DoClick` function is called. Your `DoClick` can perform an action such as drawing in the pane; or it can initiate a process such as dragging an object or selecting multiple objects. Some panes, such as the edit text pane implemented by CEditText, have built-in `DoClick` functions, which may already implement the behavior needed by your pane class.

If you want your mouse action to be undoable, you need to create a derived class of CMouseTask and call its `TrackMouse` function. The section "Mouse tracking" later in this chapter discusses undoable mouse actions in detail.

Working with Panes

In the THINK Class Library, all your drawing takes place in a pane. A pane is a rectangular region of the screen completely enclosed by a window. A window may have several panes, and each pane may have several subpanes. A pane can handle mouse clicks and events that affect its display such as activate and deactivate events. Each pane has its own drawing environment.

Every pane has an enclosure that completely encloses the pane. A pane also has a supervisor—an object in the chain of command. A pane's enclosure and supervisor can be the same object, particularly if the pane belongs to another pane. In the most common case, though, the supervisor is the director that owns the window or the pane.

Every pane has its own drawing environment. The rectangle that describes the edges of a pane is the frame. The frame defines a local coordinate system for the pane. In most cases, the upper-left corner of the pane is the point (0, 0).

Windows and panes

Every pane is an object of some class derived from the abstract class CView, which defines the behavior of visual entities. CView is the base class of CPane. A window is a view, but it is not a pane; likewise, the global desktop object, `gDesktop`, is a view but not a pane. All other visual entities are panes; they include such things as controls, borders, pictures, and size boxes.

Every view has one enclosure. For example, in the window in Figure 7-1, there are eight views and seven panes (the window is not a pane):

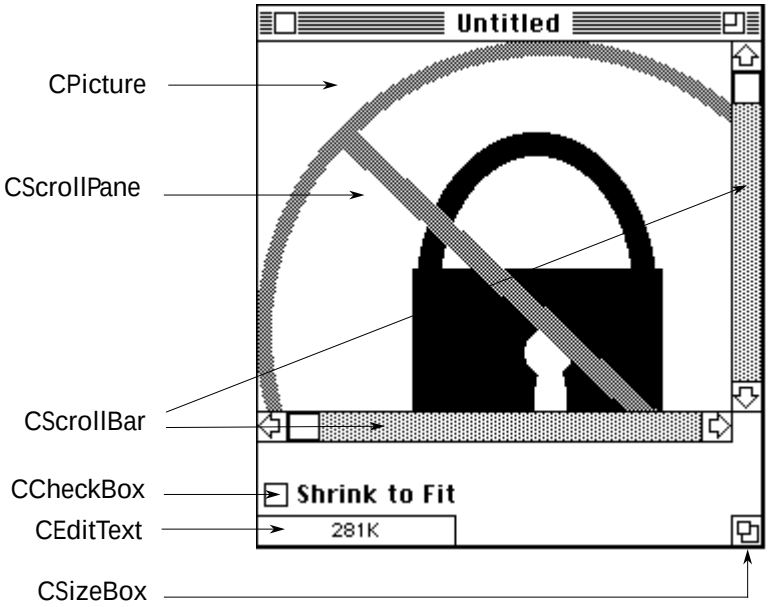


Figure 7-1 Panes in a window

The size box, the check box, the text, and the scroll pane are enclosed by the window. The two scroll bars belong to the scroll pane. The picture is enclosed by the scroll pane.

Coordinate systems

When you're working with panes in the THINK Class Library, you need to know about four coordinate systems:

- Global coordinates
- Window coordinates
- Frame coordinates
- QuickDraw coordinates

The desktop, some internal member functions, and some Toolbox routines use global coordinates. In this coordinate system, all units are in pixels, and (0, 0) is at the top-left corner of the main screen. You rarely use global coordinates in the THINK Class Library.

With window coordinates, the top-left corner of the window's content region is (0, 0) and each unit is a pixel. You need to use window coordinates to set the position of a pane whose enclosure is a window. Window coordinates are also useful as a common point of reference for two different panes in the same window.

Frame coordinates provide a local coordinate system for a pane. Units in frame coordinates are in pixels, and the point (0, 0) is usually the upper-left corner of the pane. If the pane moves within its enclosure, the coordinate system does not change; the upper-left corner is still (0, 0). The only time this origin point changes is when you scroll a panorama, which is a special kind of pane. Each pane can dynamically choose to use long (32-bit) or short (16-bit) coordinates by calling the inherited CView function `UseLongCoordinates( fUseLong )`.

All drawing and mouse tracking is done in QuickDraw coordinates. This is the coordinate system that the Macintosh Toolbox uses for its drawing operations. QuickDraw coordinates are valid only after a call to `Prepare`. The relationship between QuickDraw coordinates and the other coordinate systems depends on whether a pane is using long-frame coordinates or short-frame coordinates.

Short coordinates map directly to QuickDraw coordinates. Each element in a short coordinate uses 16-bit values, so a pane that uses short coordinates is limited to the rectangle (–32768, –32768, 32767, 32767). Long coordinates layer a 32-bit coordinate system on top of the QuickDraw 16-bit coordinates. The long coordinate system lets you use a much larger coordinate area for your pane. Since all drawing takes place in QuickDraw coordinates, you have to map the long coordinates to QuickDraw coordinates when you draw in a pane.

The CPane class defines several functions that transform coordinates from one system to another.

Drawing in a pane

Every pane has its own drawing environment. The rectangle that describes the edges of a pane is the pane's frame. The frame defines a local coordinate system for the pane.

To draw in a pane, you usually override its `Draw` function. The THINK Class Library calls your pane's `Draw` function whenever the pane needs to be updated.

Note

Update events are given low priority by `WaitNextEvent` (and `GetNextEvent`). If you want to redraw your pane immediately, you can call your own `Refresh` function to force an update event.

To draw directly in a pane as a result of a mouse click, override the pane's `DoClick` function and do the drawing in your `DoClick` function.

Use the standard QuickDraw routines to draw. The pane's `Prepare` function sets up the QuickDraw port. If your pane uses short coordinates, the coordinate system is set up correctly. If your pane uses long coordinates, you need to transform frame coordinates to QuickDraw coordinates before you draw. You can use CPane functions `FrameToQD` and `FrameToQDR` to convert frame points and rectangles to QuickDraw points and rectangles.

The THINK Class Library uses the types `LongRect` and `LongPt` for both long and short coordinates. Most of the descendants of CView that work with points and rectangles use these types.

Note

If you've worked with earlier versions of the THINK Class Library, this uniform use of `LongRect` and `LongPt` is the biggest change you'll notice, since it will necessitate some changes to your programs.

These types are defined as follows in `LongCoordinates.h`:

```
typedef struct LongPt
{
   long v, h;
} LongPt;

typedef struct LongRect
{
   long top, left, bottom, right;

} LongRect;
```

If the pane uses short coordinates, frame coordinates and QuickDraw coordinates are identical. Therefore, the values stored in a `LongRect` or in a `LongPt` are already in QuickDraw coordinates. To use them with QuickDraw routines, however, you need to convert those structures to the QuickDraw types `Rect` and `Point`. The THINK Class Library provides several utility routines to perform these conversions. For a complete list, see the section "Long Coordinate Utilities" in Chapter 7, "Programming with the THINK Class Library."

Properties of panes

When a pane moves or changes size, all of the panes that it encloses change as well. The way a pane changes depends on its sizing characteristics. When you create a pane, you specify its horizontal and vertical sizing characteristics.

The horizontal sizing characteristic specifies how the pane's left and right edges change.

Horizontal sizing	Meaning
`sizFIXEDLEFT`	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as it was when originally placed.
`sizFIXEDRIGHT`	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as it was when originally placed.
`sizFIXEDSTICKY`	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
`sizELASTIC`	The width of the pane grows or shrinks by the same amount as the width of the enclosing pane.

Table 7-2 Horizontal sizing characteristics

The vertical sizing characteristic specifies how the pane's top and bottom edges change.

Vertical sizing	Meaning
`sizFIXEDTOP`	The top edge of the pane is always the same number of pixels from the top edge of the enclosing pane as it was when originally placed.
`sizFIXEDBOTTOM`	The bottom edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as it was when originally placed.

Table 7-3 Vertical sizing characteristics

Vertical sizing	Meaning
`sizFIXEDSTICKY`	The top and bottom edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
`sizELASTIC`	The height of the pane grows or shrinks by the same amount as the height of the enclosing pane.

Table 7-3 Vertical sizing characteristics *(Continued)*

Here are a couple of examples. A vertical scroll bar in a window has the horizontal characteristic `sizFIXEDRIGHT`, so that it has a fixed horizontal length and remains anchored to the right edge of the window. Vertically, it changes with the height of the window because it has the characteristic `sizELASTIC`. A status box in the lower-left corner of a window has the horizontal characteristic `sizFIXEDLEFT` and the vertical characteristic `sizFIXEDBOTTOM`. As a result, it has a constant size and remains anchored to the lower-left corner of the window.

In Figure 7-2, the box with the thick outline represents the window. It contains a main pane that takes up most of the window, and several other panes. The two panes that hold the scroll bars are fixed to the edges of the window. When the window is resized, each grows or shrinks in one dimension by the same amount as the window. The panes that hold the grow box and the status box are anchored to the bottom corners of the window. The square pane in the upper-left portion of the main pane is always there, regardless of

how the window changes. Its dimensions do not change automatically.

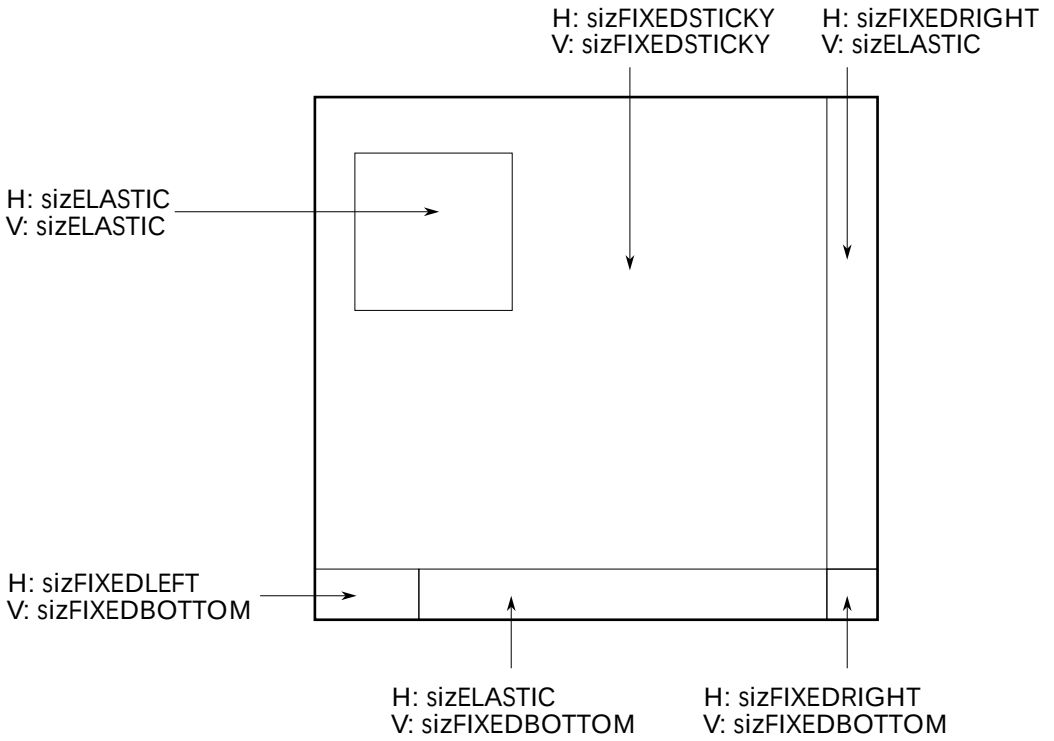


Figure 7-2 Horizontal and vertical sizing

Panoramas

Almost everything you want to display is bigger than a pane. Graphics and text, for instance, frequently take up more room than a pane contains. The THINK Class Library provides a panorama class, CPanorama, that lets you display in a pane portions of a large graphic.

Look at the sailfish in Figure 7-3. Imagine that the only part of the fish you can see is the portion inside the frame. Currently, the frame displays the sailfish's tail; but you can move, or scroll, the frame to see other parts, such as the head. This is how a panorama works.

The rectangle that encloses the entire panorama is called the bounds rectangle. It defines the size and coordinate system of the panorama.

Usually, the upper-left corner of the bounds rectangle is the point (0, 0), and its coordinate system uses pixels as units.

The coordinate system of the bounds rectangle specifies how the frame moves over the panorama when you scroll. Usually, the frame moves one pixel at a time. In some applications, though, scrolling in a different way is more natural. For instance, in a text editor, scrolling vertically one line at a time probably makes the most sense; in a spreadsheet, rows provide the most natural unit of vertical scrolling.

You can specify a scale to indicate how many pixels make up a single panorama unit. You can set different scales for horizontal and for vertical units. In a graphics application, each unit might be 1 pixel. In a spreadsheet, a vertical unit might be 12 pixels, and a horizontal unit might be 60 pixels.

The units of the panorama bounds rectangle are for scrolling only. For drawing, you use the frame coordinates, which are always single pixel units.

There are two ways to talk about the upper-left corner of a frame. Expressed in panorama units, this corner designates the position of the frame in the panorama. The same corner, expressed in frame coordinates, designates the origin of the frame.

Remember that scrolling always occurs in panorama units, and drawing always occurs in frame coordinates. As you scroll, the origin of the frame changes. Here are a couple of examples to clarify these concepts.

In Figure 7-3, both the horizontal and vertical scales are set to 1 pixel per panorama unit. The bounds rectangle of the panorama is (0, 0, 400, 380). The frame's position in the panorama is (165, 210).

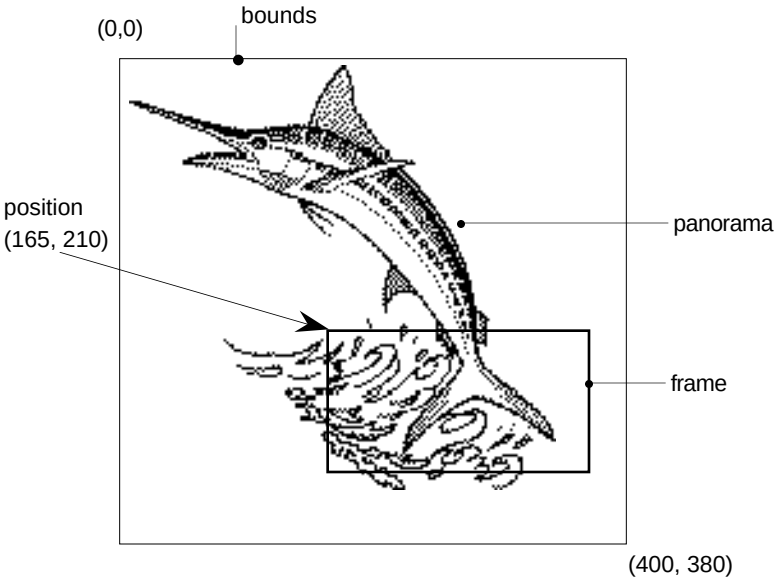


Figure 7-3 Graphic panorama and its scales

Since the panorama units match the frame units, the position of the frame in the panorama and the origin of the frame are the same.

In Figure 7-4, which shows a panorama with text, the horizontal scale is 6 pixels per unit and the vertical scale is 12 pixels per unit. The bounds rectangle of the panorama is (0, 0, 40, 9). In the panorama scale, this means 8 lines of 40 characters each. The position of the frame in the panorama is (0, 3), the beginning of the fourth line. The origin of the frame, however, is at (0, 36). If you

wanted to draw a line to strike out the word “And,” you would draw it from (0, 42) to (18, 42).

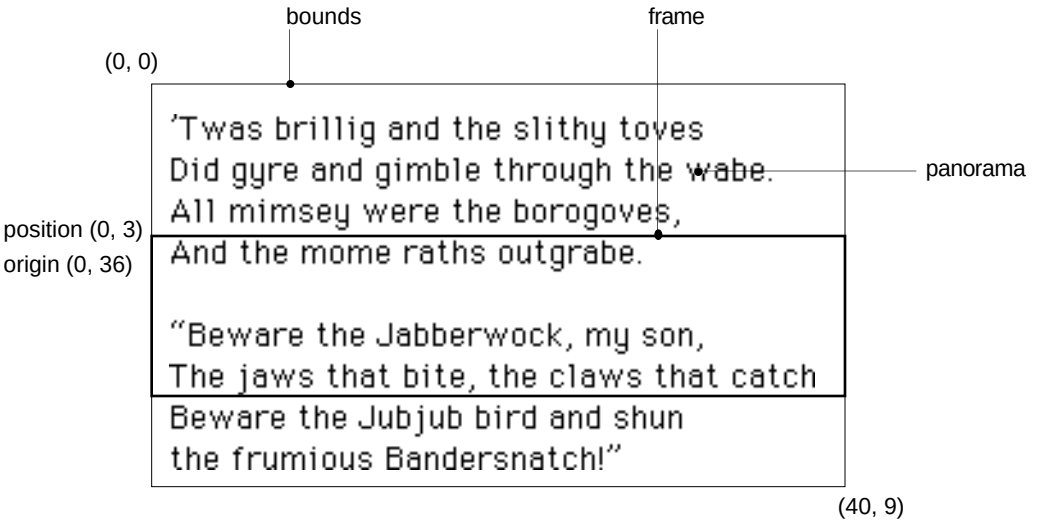


Figure 7-4 Text panorama and its scales

Scroll panes

To make it easy to use panoramas, the THINK Class Library provides a class called CScrollPane that implements a scroll pane. Scroll panes let you attach scroll bars to your panorama.

You create a scroll pane the same way you would any other pane. First, request a vertical scroll bar, a horizontal scroll bar, and a size box. Then use the `InstallPanorama` function to associate a panorama with the scroll pane. The scroll pane examines the panorama and adjusts the scroll bars appropriately. You can specify how many panorama units to scroll when you click on different parts of the scroll bar.

The scroll bars and the panorama communicate through the scroll pane. When you click one of the scroll bars, it informs the scroll pane, which tells the panorama how much to scroll.

Cursor tracking

The `AdjustCursor` function that all panes inherit from CView lets you change the cursor when it moves into your pane. When you need only one cursor for a pane, which is the usual case, all you

have to do is set the cursor with the Toolbox routine `SetCursor`. Look at the `AdjustCursor` function in CEditText for an example.

Sometimes, though, you might want to use different cursors within the same pane. The `AdjustCursor` function lets you do this as well, but at the cost of a little more work. See the description of the CView class in Chapter 121, "CView."

Working with Menus

The THINK Class Library lets you identify menu items by assigning them unique command numbers. Command numbers are positive long integers in the range 0 to 2,147,483,647. Those in the range 1 to 1023 are reserved for the THINK Class Library. Command number 0 is reserved for `cmdNull`, the null command. All other command numbers (1024 to 2,147,483,647) are available for your application.

The reserved command numbers are for the most common Macintosh application commands, such as **Open**, **Save**, **Quit**, and **Page Setup**. Be sure to use the appropriate reserved number to invoke the associated application command implemented by the THINK Class Library. For a list of all the reserved commands, see Chapter 23, "CBartender."

When you choose a command from a menu, the desktop calls the `FindCmdNumber` function of the bartender. The bartender matches the menu ID and the item number to a command number and calls the gopher's `DoCommand` function. Remember, the gopher is the first object in the chain of command, and thus the first object given the chance to handle a command.

Using MENU resources

To create menus with `ResEdit`, you must append the command number to the menu item. The menu item and the command

number are separated by the character #. For example, Figure 7-5 shows the **File** menu as viewed in `ResEdit`.

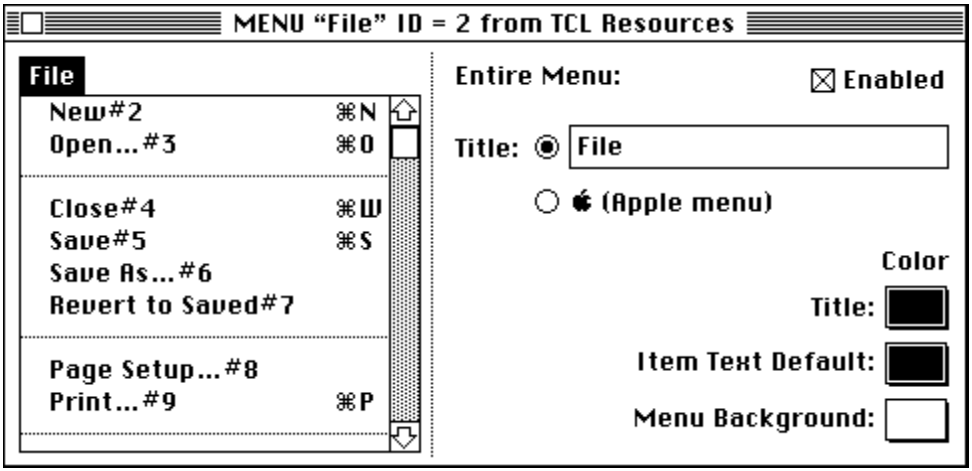


Figure 7-5 File menu viewed in `ResEdit`

Note

If you don't append a command number to a menu item, the bartender automatically assigns that item to the command `cmdNull`.

The `MENU` *resources for these menus are in the file* `TCL Resources`.

You can use any menu ID for your application's menus. The THINK Class Library reserves the following menu IDs for certain menus:

Menu title	Menu ID	Mnemonic
Apple	1	`MENUapple`
File	2	`MENUfile`
Edit	3	`MENUedit`
Font	10	`MENUfont`
Size	11	`MENUsize`

Table 7-4 Menu IDs

After you create all the menus that your application needs, create an `'MBAR' 1` resource that contains the IDs of these menus. The application's `SetUpMenus` function creates the bartender (stored in the global variable `gBartender`) to read the `MBAR 1` resource. The bartender creates the tables that match command numbers to menu items.

Note

The application's menus must be in `MBAR 1` unless you change the definition of `MBARapp` in `Constants.h`.

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command number –1 in your `MENU` resource. The bartender returns the negative of the long integer that contains the menu ID in the high word and the menu item number in the low word.

Building menus on the fly

Menus created while a program is running, such as Font menus, don't have command numbers associated with their items that you've specified at design time. Instead, the bartender's `FindCmdNumber` function returns the negative of the long integer that contains the menu ID in the high word and the item ID in the low word. When your `DoCommand` function gets a negative command number as argument, you must figure out the command from the menu ID and item number.

For example, if `DoCommand` gets a command -655369 (`0xFFF5FFF7`), the sign of the argument indicates that no command number defined at design time is associated with the menu item. To extract the menu ID and the item number, negate the value and split it into two words. The negative of the sample argument is 655369 (`0x000A0009`), so the menu ID is 10 and the item number is 9.

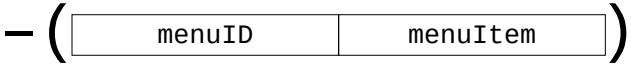


Figure 7-6 How FindCmdNumber builds command numbers

You can add menu items to existing menus. These items must be added at the end of an existing menu; otherwise, the bartender gets confused.

Dimming and checking menu items

The bartender includes functions for enabling and disabling, as well as checking and unchecking, menu items. When you click the menu bar, the bartender calls the gopher's `UpdateMenus` function. Typically, this results in the `UpdateMenus` function being called for every bureaucrat in the chain of command. In the general case, all

the items in the menu start out disabled and unchecked. Then each bureaucrat enables the menu items that pertain to it. Once the appropriate items have been enabled and checked, the Toolbox routine `MenuSelect` is called to display all the menus.

Note

This process of enabling, disabling, checking, and unchecking takes very little time. No noticeable delay occurs between clicking the menu bar and the display of the menu.

Suppose you click the menu bar of a text-processing application. First, the bartender disables all the menu items. Then, the application enables the application-related menu items such as **New**, **Open**, and **Quit**. The document enables the document-related items, such as **Save**, **Save As**, and **Revert** (if the document has been changed). A pane might check the current font and size in the **Font** menu. Finally, the menu appears on the screen with the correct items checked and enabled.

The `UpdateMenus` functions of your application, document, and pane need to enable each item. To ensure that item-enabling proceeds from the general (application) to the specific (pane), call the base class function first in your own `UpdateMenus` function.

You can use the bureaucrat functions `SetDimOption` and `SetUnchecking` in your application `SetUpMenus` function to modify the preliminary disabling (or "dimming") and unchecking process. `SetDimOption` lets you specify whether the bartender should dim all, some, or none of the items when you click the menu bar. For **Font** menus, for instance, dimming all the font names and then enabling them again makes little sense.

Dim Option	Meaning
`dimNONE`	Never dim any of the menu items.

Table 7-5 Menu dimming options

Dim Option	Meaning
`dimSOME`	Dim only the menu items that have command numbers associated with them.
`dimALL`	Dim all of the menu items. Each bureaucrat's `UpdateMenus` function must enable the items for the commands it handles. This is the default.

Table 7-5 Menu dimming options *(Continued)*

`SetUnchecking` lets you specify whether the bartender unchecks all menu items.

Unchecking option	Meaning
`TRUE`	Uncheck all the menu items at menu selection. Your `UpdateMenus` function should check the appropriate items. Use this option for menus, such as **Font** menus or **Style** menus, whose items are mutually exclusive.
`FALSE`	Don't uncheck any menu items at menu selection. This is the default, since most menu items never need to be checked.

Table 7-6 Menu unchecking options

Handling Low-Memory Situations

The CApplication class provides several functions that deal with low-memory situations. These functions use a memory reserve called the rainy day fund.

Note

For details about the functions and data members described in this section, see Chapter 18, "CApplication."

When you create a CApplication object, you supply arguments to its constructor specifying how much memory your application should allocate for the rainy day fund. You also specify how many bytes should be available to satisfy Toolbox routine memory requests, and

how many bytes of that fund should be available to satisfy critical operation requests. These values are stored in the data members `toolboxBalance` and `criticalBalance`.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function. In the THINK Class Library, the grow zone function calls the application's `GrowMemory` function.

The `GrowMemory` function tries several strategies to free memory in the heap. First, it calls the application's `MemoryShortage` function. Your application class should override this function to release memory that is not crucial to execution. For instance, your `MemoryShortage` function might delete a buffer it no longer needs. If the `MemoryShortage` function isn't able to release enough memory, `GrowMemory` starts using the rainy day fund.

`GrowMemory` looks first at the `inCriticalOperation` data member to release memory from the rainy day fund:

If inCriticalOperation is…	Leave this much in reserve
TRUE	`toolboxBalance`
FALSE	`criticalBalance`

Table 7-7 How much memory to leave in reserve

A critical operation is an operation that occurs in response to a single Macintosh event, and that should never fail—even though it might require a large amount of memory. For instance, saving a file is a critical operation. You can use the routine `SetCriticalOperation` to set the `inCriticalOperation` flag.

If `GrowMemory` still isn't able to release enough memory and the `canFail` flag is `TRUE`, it returns without trying to allocate any more memory. Setting the `canFail` flag to `TRUE` indicates that the application can deal with a failing memory request.

If the `canFail` flag is `FALSE`, `GrowMemory` tries to release all the memory left in the rainy day fund, because the application is not prepared to deal with a failing memory request. If there is still not enough memory, even after using the rainy day fund, the application will probably crash.

You can use the routine `SetAllocation` to set and reset the `canFail` flag. In general, your application should be prepared to handle failing memory requests.

Undoing and Mouse Tracking

The THINK Class Library provides a class that lets you implement the **Undo** command easily. In the default implementation, each document has its own undo history.

Undoing

The THINK Class Library uses the abstract class CTask to implement undoable actions. For every undoable action in an application, you need to create a derived class of CTask.

After you perform an action, you store enough information to undo it in the task's data members. Then you call the supervisor's `Notify` function with the task as an argument. The CDocument class implements the `Notify` function to store a task in one of the document's data members. When you choose **Undo** from the **Edit** menu, the document's `DoCommand` function calls the `Undo` function of the task it stored.

Here's an example. Suppose you've derived a class from CTask to change the font in an edit text pane. Before calling the edit pane's `DoCommand` function to change the font, you create a task and store the current font in a data member that you have defined. After you pass the font command to the edit text pane, you call the document's `Notify` function with the task as an argument.

The `Undo` function would simply call the document's `DoCommand` function to change the font back to the saved, previous one. Since the command goes through the regular command chain, the `DoCommand` function would create a task to let you undo what you were undoing.

Mouse tracking

The THINK Class Library uses the undo mechanism to make mouse tracking easier and undoable. The CMouseTask class is an abstract class that defines functions specifically for mouse tracking.

To implement a mouse tracking task, define a derived class of CMouseTask and override the `KeepTracking` and `EndTracking` functions. The `KeepTracking` function does whatever you want to

happen while the mouse is down. The EndTracking function does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, KeepTracking might draw a gray outline that moves as you move the mouse. The EndTracking function would erase the rectangle from its old location and redraw it in the new location.

To make your mouse task undoable, you need to store enough information in the task object to undo the effects of mouse tracking. You must also override Undo (inherited from CTask) to use this information to undo the effects of the mouse task.

After tracking the mouse, you can call the document's Notify function with the task as an argument. When you choose **Undo** from the **Edit** menu, the document calls Undo to undo the effects of mouse tracking.

Debugging and the THINK Class Library

The THINK Class Library incorporates features that help you debug TCL programs.

Debugging aids in Symantec C++

If the preprocessor symbol __TCL__DEBUG__ is defined, various error-checking functions, such as the TCL_ASSERT macro, are enabled. The TCL_ASSERT macro takes a Boolean expression as an argument and displays an alert if the assertion is FALSE.

In Symantec C++, the global variables gBreakFailure and gAskFailure let you examine and simulate exceptions. If gBreakFailure is TRUE, the THINK Class Library calls the Toolbox routine Debugger when Failure is called, and it calls DebugStr when the ASSERT macro is called with an argument that evaluates to FALSE. If gAskFailure is TRUE, the THINK Class Library calls Debugger when the utility routines such as FailOSErr are called that may raise exceptions. You can then change the arguments to the utility routine to simulate an exception.

There are two underscores at the beginning and end of __TCL_DEBUG__. You can set the value of __TCL_DEBUG__ on the Prefix page of the **Symantec C++ Options** dialog.

THINK Class Library Resources

The THINK Class Library requires that certain resources be present in your project's resource file. All of the resources described in this section are in the file `TCL Resources`. The mnemonic constants for all of these resources are in the file `Constants.h`, which resides in the `Core Headers` folder within the `Core Sources` subfolder of the `Class Library` folder.

Alerts

The `'ALRT'` and `'DITL'` resources always have matching IDs. You can change these resources to suit your application.

ALRT/DITL ID	Used for
`128`	General. A handy, all-purpose alert box. The `'DITL'` contains only "`^0`" so you can use the Toolbox routine `ParamText` to set up the text.
`150`	Confirm to revert to last saved version.
`151`	Confirm to save changes before closing or quitting.
`200`	Severe Macintosh error has occurred.
`250`	No printer selected.
`251`	Error alert. The `'DITL'` message is "`Couldn't complete the last command because ^0.`"
`252`	Error alert. The `'DITL'` message is "`Couldn't successfully startup or quit the application because ^0.`"
`253`	Assertion failed. Used by assertion in exception handling.
`300`	Macintosh OS error alert.

Table 7-8 Alert and dialog resources

Controls

The THINK Class Library uses this `CNTL` template for all the scroll bars it creates.

CNTL	Used for
`300`	Scroll bar

Table 7-9 Control resources

Error message strings

The THINK Class Library uses a resource of type `'Estr'` to report Macintosh errors. `'Estr'` resources have exactly the same format as `'STR '` resources. You can use the `ResEdit` command **Open as Template** from the **File** menu to open and edit an `'Estr'` resource as an `'STR '`.

The ID of the `'Estr'` resource is the error code you want to identify. The file `TCL Resources` includes `'Estr'`s. The error-handling class CError uses `'Estr'` resources to display messages. You should create an `'Estr'` for every error an application reports to the user.

Estr	Used for
`-108`	Out of memory
`-192`	Tried to get nonexistent resource

Table 7-10 Error string resources

Menus

The THINK Class Library reserves the menu IDs shown in Table 7-11 for the standard menus. The **File** and **Edit** menus contain all the standard items. You can remove the ones that don't apply to your application. The bartender builds the desk accessory menu for you automatically, but you have to build the **Font** and **Size** menus yourself in the `SetUpMenus` function of the application. For sample **Font** and **Size** menus, see the TinyEdit project in the `TCL Demos` folder.

Menu	Used for
`1`	Apple
`2`	File
`3`	Edit

Table 7-11 Menu resources

Menu	Used for
10	Font
11	Size

Table 7-11 Menu resources *(Continued)*

Note

The mnemonics for these menus are in `Commands.h`, not in `Constants.h`.

Menu bars

The THINK Class Library uses the `'MBAR'` resource to install all the menus in your application. This resource automatically includes the **Apple**, **File**, and **Edit** menus.

MBAR	Used for
1	List of all menus to install at application startup.

Table 7-12 Menu bar resource

Small icon

Earlier versions of THINK Class Library used a small icon instead of the Toolbox routine `DrawGrowIcon` to draw a grow box. If your application uses `SICN`s, you can use the routine `DrawSICN` to draw it in a pane. See Chapter 127, "TCL Utilities" for details.

SICN	Used for
200	Grow box

Table 7-13 Small icon resource

Strings and string lists

The THINK Class Library uses these strings for various prompts and messages. You can modify these to suit your application. The string resources are in Table 7-14.

STR	Used for
150	Prompt for the **Save As** dialog box.
300	Generic operating system error message used when no `Estr` resource is available.
301	Generic error suffix, "`of a Mac OS Error`".

Table 7-14 String resources

The string list resources are in Table 7-15.

STR#	Used for
128	List of common Macintosh words and phrases. This list includes `quitting`, `closing`, `Undo`, `Redo`, `Untitled`, `Show Clipboard`, and `Hide Clipboard`.
129	Strings used for low-memory warnings.
130	Task names for changing the wording of the Undo menu item text. This resource has no strings. An application should add strings to this list if it supports Undo. See the descriptions of CTask and CMouseTask.
131	Strings used by exception handler.
133	Strings used for dialog entry validation.

Table 7-15 String list resources

Window template

The THINK Class Library requires only one window template for the Clipboard window. An application usually defines one or more additional `WIND` templates.

WIND	Used for
200	Clipboard. Window template used for displaying the clipboard.

Table 7-16 Window resource

Segmentation and the THINK Class Library

You can segment your application in any fashion. Remember, however, that certain files and libraries must be in a resident segment—that is, a segment that is never purged.

These files and libraries must be in a resident segment:

- CApplication.cp
- Exceptions.cp
- LongCoordinates.cp
- TCLpstring.cp
- TCLUtilities.cp
- CPlusLib
- MacTraps
- MacTraps2

Modifying the THINK Class Library

In general, to change the behavior of one of the THINK Class Library classes, you create a derived class of the class you want to modify, and add new member functions or override the existing ones you need to change. Almost all member functions in the THINK Class Library are virtual.

In rarer cases, you might decide to change the source code for a THINK Class Library class. However, two dangers are inherent in this approach. First, Symantec will release new versions of the THINK Class Library. Changes made in the source code may make it difficult to use new classes or updates of existing classes. If you do make changes to a class's source code, make sure to keep an archival copy of the original files and to mark your changes clearly.

The second danger is somewhat more subtle. As you use the THINK Class Library, you may want to create general-purpose, reusable classes derived from THINK Class Library classes. Neither you nor anyone else can use these classes in other programs if, in creating these classes, you relied on features of THINK Class Library classes that you introduced by changing source code.

Note

Under your license agreement, you may distribute new classes derived from the classes in the THINK Class Library. You may not, however, distribute modified sources of THINK Class Library classes.

Where to Go Next

Learning to use the THINK Class Library takes time and experimentation. Start with the TinyEdit example in the `TCL Demos folder`. You might start by adding an About box to the TinyEdit application. (Hint: Since displaying the About box should occur in response to an application-wide command, implement it in the application's `DoCommand` function.) You may also want to experiment with the NewClassDemo, ArtClass, and AEStarter demos, also in the `TCL Demos` folder. As you explore how these applications are put together, refer to the chapters in Part 4 of this guide to understand how the classes of the THINK Class Library work.

Using Object I/O 8

The Object I/O capabilities of the THINK Class Library make it easy to store objects to and retrieve objects from disk, preserving their entire state. Thus, your objects can persist even while the application that created them is not running; when the application next retrieves these objects, they are recreated exactly as they were when stored. Among the many uses you'll find for this capability is preserving the complete state of an application before it quits. You can preserve, for example, open documents, windows, panes and their contents, sizes and positions, and so on. This chapter also introduces the THINK Class Library view utilities, which build on Object I/O to facilitate reading and writing views and windows stored as resources.

Contents

Introduction

Object I/O provides everything you need to permanently store objects on disk as documents or resources. Object I/O can:

- Write any object to disk and read it back
- Read and write complete recursive data structures with one program statement
- Read and write simple data, such as integers and strings
- Be easily extended to read and write your own classes, regardless of whether or not they are derived from THINK Class Library classes

Object I/O is also used to read view resources written by Visual Architect. For information on this topic, see the section "Using the View Utilities", later in this chapter.

Object I/O is more than a simple I/O library: It allows you to read and write arbitrary collections of objects with a single program statement. Input in Object I/O is polymorphic; you do not need to know the specific class of the object about to be read in, only its root class—the object's ancestor class that is highest in the class hierarchy and has no base class.

Forty-four of the standard THINK Class Library classes can do Object I/O, including all the collection classes. To extend Object I/O for your own derived classes, you only need to write two macros and define two simple member functions in each derived class.

Streams

Object I/O does its work using streams. A stream is an object that gets and puts information either sequentially or randomly.

CStream

CStream is an abstract base class that defines the basic protocol—the functions used to get and put—for all stream classes.

CStream is different from the C++ iostream classes. The main job of the iostream classes is translating between binary and text format. The CStream derived classes are binary streams. They get and put data in internal format, avoiding conversion overhead. Also, to read

an object from an iostream, you must know the exact class of the object, whereas CStream only requires that you know the object's root class.

CFileStream, CHandleStream, and CCountingStream

The three concrete stream classes in the Object I/O Library are CFileStream, CHandleStream, and CCountingStream; all are descendants of CStream. CFileStream reads and writes a Macintosh data file using a CDataFile object. CHandleStream reads and writes a Memory Manager handle. Typically, CFileStream is used to get and put documents, while CHandleStream is used to get and put resources. CCountingStream does no I/O. It is used to simulate output operations to determine how much space they require.

CFileStream and CHandleStream are both derived from the intermediate class, CBufferedStream. CBufferedStream improves the performance of file streams (up to 10:1) by buffering chunks of data to be read or written in program memory, rather than reading or writing each data item immediately from or to the buffers used by the operating system.

Handle I/O isn't buffered. A CHandleStream uses CBufferedStream purely for its get and put functions. The handle is the buffer, in this case.

Basic stream operations

The five basic functions of a stream are `Open`, `Close`, `AtEnd`, `Get` and `Put`:

- `Open` readies a stream for input or output (or both).
- `Close` completes any outstanding put operations and returns the stream to the unopened state.
- `AtEnd` tests for end-of-stream.
- `Get` reads bytes from a stream opened for reading.
- `Put` writes bytes to a stream opened for writing.

These five functions, together with the usual complement of constructors, initialization functions and destructors, make possible any sort of sequential I/O—at least in principle.

In practice, a number of higher-level functions are provided to make programming more convenient. These functions get and put primitive data types, Memory Manager data, and objects.

Primitive data functions
Functions that get and put primitive data types are listed here:

Get	Put	Type
`GetBoolean`	`PutBoolean`	`Boolean`
`GetChar`	`PutChar`	`char`
`GetCString`	`PutCString`	`char *`
`GetDouble`	`PutDouble`	`short double`
`GetFloat`	`PutFloat`	`float`
`GetInt`	`PutInt`	`int`
`GetLong`	`PutLong`	`long`
`GetShort`	`PutShort`	`short`
`GetStr255`	`PutStr255`	`unsigned char *` (Pascal string)

Table 8-1 Primitive data Get and Put functions

Should you wish to do so, adding more primitive-type functions to your stream-derived class is easy; however, the types listed in Table 8-1 meet most needs when they are coupled with appropriate typecasting.

Note that the `float` and `double` functions use IEEE standard 32- and 64-bit formats, respectively.

Memory Manager data functions
Functions that get and put Memory Manager data are:

Get	Put	Type
`GetHandle`	`PutHandle`	`Handle`
`GetPtr`	`PutPtr`	`Ptr`

Table 8-2 Memory manager data Get and Put functions

Of course, these functions do not get and put the handle or pointer. Instead they get and put the data on the heap pointed to by the handle or pointer.

`GetHandle` and `GetPtr` allocate new space on the heap and then copy the information from the stream to the allocated space. `GetHandle` and `GetPtr` may fail to find the memory they need. If

so, they raise an exception using the THINK Class Library exception mechanism, by calling `Failure`.

Calling `PutHandle` or `PutPtr` with a null pointer is permissible; it does not cause an error. Likewise, `GetHandle` and `GetPtr` may return a null pointer.

Object functions
Member functions that get and put object data are:

Get	Put	Type
`GetObject`	`PutObject`	any class

Table 8-3 Object functions

`PutObject` writes an object to a stream, and `GetObject` reads an object from a stream. A class that is to be used with Object I/O must be declared as a run-time type identification (RTTI) class using the `TCL_DECLARE_CLASS` and `TCL_DEFINE_CLASS` macros described in Chapter 9, "Exception Handling and RTTI."

Like `GetHandle`, `GetObject` allocates memory from the heap and then copies the object's contents from the stream to the allocated space, returning a pointer to the new object. `GetObject` uses the `new_by_name` function, described in Chapter 9, "Exception Handling and RTTI," to create the object.

It is permissible to call `PutObject` with a null pointer, and `GetObject` may return a null pointer.

More about `GetObject` and `PutObject` is in the CSaver section later in this chapter.

Views and bureaucrats
Two additional functions get only object data:

Function	Type
`GetBureaucrat`	`CBureaucrat*`
`GetView`	`CView*`

Table 8-4 Views and bureaucrats

You may never use `GetBureaucrat`, but `GetView` can be handy.

Object I/O can get and put windows, panes, controls, borders, environments—most objects that represent what the user sees. This

capability allows you to get and put the entire visual state of a window almost as easily as reading or writing an integer.

You put a view with `PutObject`. You get a view with `GetView`. `GetView` is used because the newly created object needs to know its enclosure and supervisor during initialization. Having obtained that information, `GetView` calls `GetObject`.

`GetView` is called by the `TCLGetNamedWindow` and `TCLGetNamedSubview` functions used to read view resources written by Visual Architect.

Using get and put

Get and put functions are used in the obvious way. The first argument to all put functions is the data to be written. For example:

```
stream.PutLong(aValue);
```

Get functions return simple data as values, and record or array data as the value of the first argument. For example:

```
aValue = stream.GetLong();
stream.GetStr255(str);
```

Getting and putting objects is no more difficult, though the syntax is a little different. For example,

```
PutObject(stream, itsContents);
GetObject(stream, &itsContents);
```

A document might use the first statement above to write its entire contents to a stream, and the second statement to later read the contents back.

>> and << stream operators

CStream, like C++ iostreams, overloads the << and >> operators to allow you to use the following more succinct notation:

```
stream << along << ashort;
```

in place of:

```
stream.PutLong(along);
stream.PutShort(ashort);
```

where `along` is of type `long` and `ashort` is of type `short`.

The standard THINK Class Library classes that can be read or written also overload the << and >> operators, allowing the same concision for reading and writing objects:

```
stream << myObjPtr;
```

instead of:

```
PutObject(stream, myObjPtr);
```

If you want to override these operators for your own classes, add declarations like those in the following example to your class declarations:

```
friend CStream& operator <<
      (CStream& s, MyClass* p)
      { return PutObject(s, p); }
friend CStream& operator >>
      (CStream& s, MyClass*& p)
      { return GetObject(s, p); }
```

The THINK Class Library overloads these operators only for root classes—such as CCollaborator—in the class hierarchy that can do object I/O, because those classes contain all the information strictly needed by `GetObject` and `PutObject`. Then, for example, when writing or reading a CView object, the THINK Class Library uses code such as the following:

```
stream << aView;
stream >> (CCollaborator*) aView;
```

The CView pointer, being derived from CCollaborator, satisfies the << operator for CCollaborator. However, the argument to >> must be typecast, as C++ will not automatically coerce a reference to a derived class pointer (here, CView*&) to a base class pointer (here, CCollaborator*). You can use this same strategy for your own classes, if you wish, or you can simply overload the << and >> operators for each class that you want to do Object I/O.

PutTo and GetFrom

Object I/O adds two new virtual functions to every class that can do Object I/O: `PutTo` and `GetFrom`. Unlike the put and get functions, which are CStream member functions or friends of that class, `PutTo` and `GetFrom` are member functions of the persistent objects themselves. You generally will use these when reading and writing objects.

`PutTo` and `GetFrom` are called with a reference to a stream their argument. The `PutTo` function for each class is responsible for putting the data members of the class to the stream. The `GetFrom` function is responsible for getting the data members of the class from the stream and further initializing the class as needed.

For Object I/O to work on your objects, you usually need to include these two functions in each object class. (If a class has no data members and requires no special initialization, it does not need to add `PutTo` and/or `GetFrom` functions.)

Most THINK Class Library classes are provided with `PutTo` and `GetFrom` functions, so you have many examples. All follow the same general structure:

- Put data members to the stream
- Call inherited `PutTo`
- Put subordinate objects to the stream

The `GetFrom` function has the following general outline:

- Get data members from the stream
- Initialize as needed by subordinate objects
- Call inherited `GetFrom`
- Get subordinate objects from the stream
- Finish initialization for the class

A subordinate object is one pointed to by a data member of the class is saved with the object being written and restored with object being read.

By convention, a class puts all non-object data members to the stream and lets its base classes put all their non-object data members before it puts any subordinate objects. This stores the primitive data for each object contiguously on the stream. It also allows an object to completely recover its non-object data members and initialize itself before any subordinate objects are read. This can be important if the subordinate objects refer to the object in the course of initializing themselves.

Sample PutTo function

Most `PutTo` and `GetFrom` functions are quite simple. For example:

```
void    CDialogText::PutTo(CStream& stream)
{
   SaveDialogText  s;

   s.maxValidLength = maxValidLength;
   s.isRequired = isRequired;
   s.validateOnResign = validateOnResign;
   stream.PutStruct(s);
   CEditText::PutTo(aStream);
}
```

The above is the actual text of the `CDialogText::PutTo` function. As you can see, the function simply copies data members to a `struct`, puts the `struct` to the stream, and calls its base class's `PutTo` function to do everything else.

Saving subordinate objects

The ability to save subordinate objects allows you to save or restore entire objects with a single call to `PutObject` or `GetObject`.

There is no requirement that a class's `PutTo` and `GetFrom` functions put and get subordinate objects. This is entirely up to the class. How a class treats the objects it points to establishes the boundaries of the data structure that `PutObject` writes each time it is called.

A class may refrain from putting an object it points to for several reasons—most notably, because the object is not saved at all or because some other class will save it. In the former case, the object pointed to must be created afresh by the class's `GetFrom` function. In the latter, the `GetFrom` function must obtain a pointer to the object, such as from a global variable. Both kinds of behavior fit readily within the general structure provided.

Duplicate checking

You may wonder how `PutTo` and `GetFrom` work if an object is part of a pointer-based data structure that isn't a tree, but contains multiple pointers to the same object, or even cycles. For example, each THINK Class Library view object contains a pointer to its enclosure, and many views can share the same enclosure. Nevertheless, `PutTo` and `GetFrom` work properly with views.

CStream deals with recursive data structures by using a feature called duplicate checking. Duplicate checking is an option and must be turned on by calling the `CheckDuplicates` function.

Duplicate checking works by remembering all objects written to a stream. If an object previously written is again an argument to `PutObject`, the object is not written again; instead, an ID number uniquely identifying the object is written to the stream. On input, the process works in reverse; ID numbers are converted to pointers to objects previously read.

CSaver, described next, turns on duplicate checking so that the `itsContents` object can represent a fully general data structure.

CSaver

CSaver, a class derived from CDocument, adds persistence to the behavior of documents. You must derive your document classes from this class if you want them to be capable of Object I/O.

If you examine a project generated with Visual Architect that has a main window that uses files, you will see that while the application has complete file-handling capability—**New**, **Open**, **Close**, **Save**, **Save As**, and **Revert**—it contains no file I/O code. This is because it inherits all its I/O behavior from the CSaver class.

CSaver builds on the CDocument class, implementing file I/O functions that in CDocument are merely abstract. CDocument provides a basic framework for file operations with its `NewFile`, `OpenFile`, `DoSave`, `DoSaveAs`, and `DoRevert` abstract functions. However, these functions do not actually do anything, because `CDocument` makes no assumptions about the contents of the document or how the file is used.

Document ground rules

CSaver fleshes out the CDocument skeleton by establishing some ground rules:

- For a document on disk, a file and window are always open when the document is open and they are closed when the document is closed.
- A new document does not have a file until it is saved.

These rules ensure that the application has exclusive access to the document's file while the document is open.

CSaver reads and writes the file's data fork using a CFileStream. The document's contents are read into memory when the document is opened, written when the document is saved, and removed from memory when the document is closed.

Document contents

This section describes how CSaver knows what to read and write.

CDocument assumes that for each document there is a primary file pointed to by the `itsFile` data member, and a primary window pointed to by the `itsWindow` data member. CSaver adds an `itsContents` data member. `itsContents` is a pointer to the document's contents as represented in memory. Of course, most contents don't fit into a single object. Usually, `itsContents` points to an object that is the root of a data structure consisting of objects linked together by pointers, as shown in Figure 8-1.

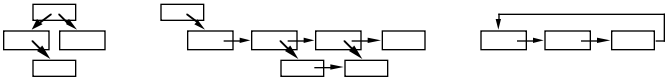


Figure 8-1 Pointer-linked object data structures

The kind of object `itsContents` points to is entirely up to the application. It may be a CList or any other kind of pointer-based data structure. The objects in the data structure can point to other objects, to Memory Manager `Ptr` or `Handle` data, or to each other. The contents data structure does not need to be simple. It can be an arbitrary graph and it can have recursive cycles. CSaver supports this generality because Object I/O does.

When a document is saved for the first time, CSaver creates a new CDataFile object as `itsFile`. It then saves `itsContents` (the entire data structure, not just a single object) to the new file. When a document is opened, CSaver creates a new CDataFile as `itsFile`,

opens it using the information provided by calls to one of its specification member functions, and reads `itsContents` (the entire data structure) from the file.

Reading and writing `itsContents` is done by creating a stream and calling its `GetObject` or `PutObject` functions, as described previously in this chapter. From the point of view of the application code, reading and writing is completely automatic.

All an application needs to do to take advantage of automatic I/O is to derive its document class from CSaver and to follow a simple discipline when defining the classes of objects pointed to by `itsContents`. These steps are outlined here.

Defining your document class

To use CSaver:

- Derive your document class from CSaver, specifying the class of the `itsContents` data member, such as:

```
class MyDoc : public CSaver<MyData>
```

- Add a source file to explicitly expand the CSaver template class. See the Chapter 94, “CSaver.”
- Override CSaver functions as needed to create and manage the document window or windows.
- Follow a simple outline when defining your `itsContents`. See the section “Defining Your Contents Classes” later in this chapter.

The only three virtual functions your document class must override are:

Function	Description
`MakeNewWindow`	Creates a new, empty window
`MakeNewContents`	Creates new, empty contents
`FailOpen`	Fails if file is already open in this application

Table 8-5 CDocument member functions to override when using CSaver

In addition to the required three, most documents need to override the virtual functions listed in Table 8-6 for translating between `itsContents` and `itsWindow`.

Function	Description
`WindowToContents`	Updates `itsContents` from the information displayed in the window. Called before putting `itsContents`.
`ContentsToWindow`	Displays `itsContents` in the window. It is called after getting `itsContents`.

Table 8-6 CDocument updating functions

A third category of CSaver functions allows you to change or customize CSaver's default behavior. These hooks include:

Function	Description
`ReadContents`	Default calls `GetObject` for `itsContents`.
`WriteContents`	Default calls `PutObject` for `itsContents`.
`NewFileType`	Default uses first in applications type list.
`PositionWindow`	Default staggers using decorator.
`MakeWindowName`	Default uses file name or Untitled.

Table 8-7 CSaver functions

Creating and opening documents

The CApplication class defines `CreateDocument` and `OpenDocument` virtual functions. `CreateDocument` creates a new document; `OpenDocument` opens an existing document. Both functions then call various member functions of the document, and recover from any failures in those functions by deleting the document.

Defining your contents classes

Define a single object as the root of your `itsContents` data structure. When you create a new document, store a pointer to this object in `itsContents`.

For Object I/O to work as intended, the root object should define `PutTo` and `GetFrom` functions that put and get the other objects comprising the document. For instance, in some applications the root object is a CList object, which already automatically puts and gets the objects on the list. Of course, the root object does not have to be a CList; it can be of any type.

You must define `PutTo` and `GetFrom` functions for all of the object classes that are part of the document's data. See the "PutTo and GetFrom" section earlier in this chapter for details.

Duplicate checking in CSaver

`CSaver` uses duplicate checking, so the `itsContents` data structure can be fully general. See the "Duplicate Checking" section earlier in this chapter for more information.

Using CSaver with Visual Architect

If you generate code with Visual Architect and your main view uses a file (the default), the document class generated to direct the view will be derived from CSaver. You do not have to worry about `NewFileType`, `PositionWindow`, or `MakeWindowName`; the generated code overrides these functions. Visual Architect also generates `CreateDocument` and `OpenDocument` functions for you.

Two additional files are generated for each document view that uses files: `ItsContentsClass_classname.h` and `CSaver_classname.cp`, where `classname` is the name of the document class. Visual Architect does not know the class of the `itsContents` variable. In the `ItsContentsClass` file, the class defaults to CCollaborator. If this is not correct, change all occurrences of CCollaborator in the `ItsContentsClass` file to the appropriate class name.

Objects on the Scrap

Object I/O can be very handy for working with the scrap, allowing you to cut, copy and paste objects to and from your program with very little effort.

Cutting and copying create a temporary output CHandleStream object in order to write the selected objects to a handle. You save the handle on the scrap with your own resource type. To paste, create an input CHandleStream object to convert the handle back to

objects, then splice the objects into your views and other data structures.

To illustrate this technique, here is a code example quoted verbatim from Visual Architect. The statements TRY, CATCH, and ENDTRY are exception-handling macros; and they are discussed in detail in Chapter 9, “Exception Handling and RTTI.” If you are unfamiliar with exception handling, you may think of TRY and CATCH as providing an object-oriented version of the standard C `setjmp`. The analogue to calling `longjmp`, not seen here, is provided by an object “throwing an exception.” The statements in the block delimited by TRY and CATCH comprise a “try block”; they can cause an exception to be thrown that this example is prepared to deal with. The block of statements between CATCH and ENDTRY deals with, or “catches,” any exception caused by execution of the try block; it is analogous to the block of statements that handles error returns to the location following `setjmp`.

```
void  CTaskCopyCutClear::Do(void)
{
   CHandleStream  *stream =
      NewOutputHandleStream(NULL);
   Handle   h;

   TRY
   {
      if (nameIndex == indexCut
         || nameIndex == indexClear)
         copyList->DoForEach(HideIt);

      PutObject(stream, copyList);
      h = stream->GetStreamHandle();

      gClipboard->EmptyScrap();
      gClipboard->PutData('ITEM', h);

      stream->DisposeAll();
   }
   CATCH
   {
      stream->DisposeAll();
   }
   ENDTRY

   if (nameIndex == indexCut
      || nameIndex == indexClear)
      editor->RenumberObjects();

   done = TRUE;
}
```

As you can see, Object I/O leaves very little to do. The example also illustrates how the THINK Class Library's Clipboard object can be useful in simplifying the scrap interface, even if your program does not display a Clipboard window.

Reading and Writing Resources

In the typical Macintosh application, when a file is opened the entire contents of the document are read into memory. The user may change the contents and save them, causing the entire contents to be written back to the file. Closing the document removes the contents from memory.

Some documents are too large to fit entirely in memory, or have parts that can be saved independently. In this case, a random I/O capability is needed. Object I/O gives you two choices:

- You can implement random access using `CFileStream` as a base class. In this approach, opening a document would read only the main part of the document and a directory to the rest of the document into memory. Your derived class would be responsible for allocating space in the file stream, keeping a directory, compacting, and so on.

- You can use the Resource Manager or an equivalent to store, retrieve, and index objects or object data structures.

This section discusses using the Resource Manager for this task.

Object I/O does not try to fool the Resource Manager and convert resources to objects in place. Saving each object as an individual resource would use the disk inefficiently and would be too slow for most applications.

Saving a collection of objects in each resource can be very fast and efficient. This approach works around some of the restrictions of the Resource Manager (such as the 2727 resource limit), and takes full advantage of Object I/O's ability to save and restore entire data structures at a time.

Object I/O provides a `CHandleStream` class that is designed to be used with resources. A handle stream is used as a buffer between the object or object data structure to be read or written, and the resource represented by the handle.

On input, `GetResource` creates the handle and a `CHandleStream` object translates between handle and object(s). The handle is then released, freeing the buffer storage.

For output, there are two possibilities:

- For an existing resource, a `CHandleStream` object translates between the object(s) and the resource handle. `ChangedResource` and `WriteResource` or `UpdateResFile` change the resource on disk.

- For a new resource, a `CHandleStream` object creates a new handle as it converts object(s) to the handle. `AddResource` is then used to add the handle to the resource file. (Alternatively, `AddResource` can be used for both existing and new resources in a general-purpose

function that always removes any existing resource before attempting to add one.)

The general outline for getting an object from a resource is:

```
aResource = GetResource(...);
aStream = NewInputHandleStream(aResource);
aStream->GetObject(&anObject);
delete aStream;
ReleaseResource(aResource);
```

After this code executes, `anObject` points to the object read from the resource.

Putting an object data structure to a resource can be accomplished as follows. As in the preceding example, `aStream` is a `CHandleStream` variable:

```
aStream = NewOutputHandleStream(NULL);
aStream->PutObject(anObject);
aResource = aStream->GetStreamHandle();
delete aStream;
AddResource(aResource, ...);
ReleaseResource(aResource);
```

These sequences are general-purpose and sufficient for most needs. However, nothing about Object I/O requires you to follow this approach. A file stream can do random I/O, so you can construct any sort of access with it. Or, you might construct a new `CBufferedStream`-derived class to take advantage of the buffered resource I/O available in System 7.

Whatever approach you choose, Object I/O either provides what you need off the shelf or gives you a solid platform for constructing more sophisticated solutions tailored to the needs of your application.

Using the View Utilities

This section describes how to use the Visual Architect view utilities to open and display Visual Architect view resources. Even though Visual Architect generates code to open alerts and windows in response to commands (see Chapter 11, “Creating and Editing Menus and Commands”), the view utilities make it easy to:

- Open windows or subviews defined as view resources in your own code.

- Use Visual Architect view resources in THINK Class Library programs that are not generated by Visual Architect.

The functions introduced in this chapter are documented in the section "View Resource Utilities" of Chapter 127, "TCL Utilities," which you should consult for additional information.

View Resources

Visual Architect views are stored as resources of `'CVue'` type in the resource fork of your application or, during debugging, in your project's `Visual Architect.rsrc` file.

Two kinds of view resources are currently defined: window and pane. Window view resources hold all the windows you define in Visual Architect—dialogs, document windows, plain windows, and floating windows. Pane view resources hold subviews.

Internally, a Visual Architect view resource has two parts: a header, and body. The header is the same for both window and pane resources. It is defined by the C `struct`:

```
typedef struct {
   unsigned char version;
   unsigned char kind;
   Point        initialPosition;
} ViewResourceHeader;
```

This format is supplied for your information only. Using the view utilities, you never have to inspect the contents of a view resource. Additional kinds of view resources, possibly with different header information, may be added in the future.

The body of a view resource is simply the window or pane as written to a stream by Object I/O. Each object in the stream begins with a class identifier and is followed by the data written by the `PutTo` member function of each derived class of the object, beginning at the outermost level. The format of the view resource is implicitly defined by the operation of the `PutObject` function and the `PutTo` member functions of the CView-derived classes you define.

Object I/O is used to read the body of a view resource. The `CStream::GetView` member function does the actual translation

from the resource encoding to objects in memory. `Get View`, in turn, calls `GetObject`.

Since Object I/O does the translation, all that remains for you to do is allocate a stream for input, read the resource from disk, and delete the stream after the view is in memory. In fact, you don't even have to do this—view utility functions do it for you.

TCLGetNamedWindow

The `TCLGetNamedWindow` function creates a window from a view resource. Its prototype is:

```
CWindow *TCLGetWindow(Str255 name,
            CBureaucrat *aSupervisor);
```

Only three lines of code are required for a director to read a window stored on disk as a view resource and display it to the user as the frontmost window. For example:

```
CWindow *theWindow;
theWindow = TCLGetNamedWindow(
                "\pMyWin", this);
theWindow->Select();
```

Any run-time initialization you need to do, such as initializing panes with data from a program, should be inserted between the calls to `TCLGetNamedWindow` and `Select`.

The view utilities also include a `TCLGetWindow` function for fetching a view resource by resource ID. Visual Architect works only with named views.

TCLGetNamedSubview

The `TCLGetNamedSubview` function creates a pane from a view resource stored on disk. Its prototype is:

```
CPane *TCLGetNamedSubview(Str255 name,
          CView *anEnclosure,
          CBureaucrat *aSupervisor);
```

Only two lines of code are required for a document director to read a subview stored as a view resource and add it as a subview of the window's main panorama. This is illustrated in the following example:

```
CPane *thePane;

thePane = TCLGetNamedSubview( "\pMySub",
            itsMainPane, this );
```

Depending upon how you set it up in Visual Architect, the subview pane will be automatically positioned either at a fixed position in the panorama or at the bottom, beneath all other panes. The subview always makes room for itself in the panorama's bounds.

Code to reposition the subview within its enclosure or to do any run-time initialization should be placed after the call to `TCLGetNamedSubview`.

The view utilities also include a `TCLGetSubview` function for fetching a view resource by resource ID instead of by name.

Locating panes within views

Within an application, for every item in a view, Visual Architect assigns a unique value to each `ID` data member. Each item has a related view resource. The value assigned to the `ID` data member is the item number concatenated to an adjusted view resource ID, which is the item's view resource ID minus 128. (128 represents the offset from the lowest-numbered view resource.)

You can call `FindViewByID`, passing this unique ID value to locate the pane corresponding to any item. You do not have to calculate or hard-code these ID values; Visual Architect generates symbolic names for you in a file named `viewnameItems.h`, where `viewname` is the name of the view (not the name of the view's director class).

You do not need to use `FindViewByID` just to obtain a pointer to an item in a window. More direct means exist for this task. For all panes enclosed by a window, with the exception of scrolling panes and panoramas, Visual Architect ensures that a pointer to that pane exists and has a name. Visual Architect generates codes in the `MakeNewWindow` member function of each lower-level director class that initializes a pointer to each item in the window. These pointers' names are created by combining the view name with the item identifier, which you set or Visual Architect generates automatically (such as `fMain_Pict1` for a `PICT` type item in the Main view). You can simply use these pointers in your derived director class to directly reference each item in the window.

Panoramas are treated differently because they are optionally enclosed by a pane of type `CScrollPane`. For example, a table that has a horizontal or vertical scroll bar or a size box will be automatically enclosed in a CScrollPane. Similiarly, when the main panorama of a window or subview has a scroll bar or size box, it is enclosed in a CScrollPane.

The simplest way to locate the outermost panorama in a window is to call the `View Utilities` function `GetItemPointer` with an item number of `0`. To get a pointer to the CScrollPane, first locate the panorama item; the scroll pane will be pointed to by the item's `itsEnclosure` data member.

How objects in view resources are initialized

When generated code reads a view resource, each individual pane is initialized by its default constructor (the constructor with no arguments) and by `GetFrom` functions. If you derive a class from such a pane or panorama and you define it to be in a view, you can perform any additional initialization by adding code to its default constructor and/or `GetFrom` function.

Once a window has been completely initialized, the `MakeNewWindow` function generated in the window's director class sets the `itsWindow` data member to point to it.

8 Using Object I/O

Exception Handling and RTTI 9

The THINK Class Library includes an exception-handling mechanism to detect and respond to failures. This mechanism lets you divide your routines into two parts: one to handle normal operation and another to handle abnormal situations.

The THINK Class Library provides two styles of exception handling. Simplified exception handling, compatible with previous versions of the THINK Class Library, provides a simplified execution model in which functions can "fail" with an error code or message code, and the failures caught by untyped handlers. Typed exception handling, the second style, is similar to exception handling in the C++ draft standard. In this style, functions throw a typed exception, which handlers can catch by type. Two different sets of macros implement the two styles. A single try block must use only one style, but the two can be mixed in the same program.

Run-Time Type Identification (RTTI) is a feature proposed in the C++ draft standard that makes it possible to test and compare the class of objects at run-time. The THINK Class Library implements a large subset of the proposed RTTI standard.

This version of the THINK Class Library uses macros to implement exception handling and RTTI in anticipation of compiler support for these features. When that support becomes available, the macros will be redefined to use it so that you will not have to change your THINK Class Library programs. All exception-handling macros are defined in `Exceptions.h`; all RTTI macros are defined in `TCLClassInfo.h`.

Contents

What Is an Exception Handler?

An exception is an occurrence of an abnormal condition during the execution of a program. For example, your program's normal operation is to open a file, read and write its data, and close the file. In the course of operation, though, your program could encounter problems: for instance, the file may not exist, the file may be unavailable, media errors may exist, or your program may not be able to close the file.

The exception-handling mechanism in the THINK Class Library lets you separate the part of a routine that specifies normal operation from the part that handles errors. Without an exception handler, your program might have this kind of structure:

```
create a file object
if creation failed
   report an error
else
   open the file
   if open failed
      delete the file object
      report an error
   else
      allocate a buffer
      if allocation failed
         delete the file object
         report an error
      else
         ...
```

As you can see, the normal flow of control is hidden in all the error-checking code. With an exception handler, your code looks more like the following:

```
TRY TO DO THE FOLLOWING:
   create a file object
   open the file
   allocate a buffer
   read the contents of the file
   close the file
CATCH ANYTHING THAT WENT WRONG:
   if the file is still open, close it
   if the file was created, delete it
   if the buffer was allocated, release it
```

The normal portion is called the try block and the portion that handles abnormal conditions is called the handler. A statement that is in the try block or is called from the try block can "throw an exception" to stop normal operation and go to a handler.

The exception-handling mechanism keeps a stack of handlers. If a function throws an exception and there is no handler in that function, control passes to a handler attached to the most recently entered try block. The THINK Class Library has a handler in CApplication's Run function, so any exception is caught that is thrown after the Run event loop has been entered.

Here's an example. Suppose you have three functions, each with a try block and handler:

```
void A()
{
    try
        call B()
    catch
        clean up A
}

void B()
{
    try
        call C()
    catch
        clean up B
        throw same exception
}

void C()
{
    try
        do something
    catch
        clean up C
        throw same exception
}
```

If an operation in function C's try block throws an exception, control passes to C's handler. When that handler is finished and the exception is rethrown, control passes to B's handler, and then finally to A's handler. If B had no handler, control would proceed from C's handler to A's handler.

Not every routine needs to have try blocks and handlers, only those that need to clean-up errors. Sometimes you can rely on a caller's handler to do the clean-up for you. See the section "Use handlers only when necessary" later in this chapter for more information.

Note

Although the exception-handling routines are designed to work with the THINK Class Library, you can use them in a limited way with non-THINK Class Library programs. To do this, you must include `BELLib` and `Exceptions.cp` in your project, and compile with `NO_TCL` defined.

When Does the THINK Class Library Throw An Exception?

In general, whenever the THINK Class Library calls a Toolbox function that returns an error code, it tests the value of the code and throws an exception if it is anything other than `noErr`.

If an object cannot be created (that is, if a call to `operator new` fails), the THINK Class Library throws an exception. Hence, `operator new` never returns `NULL`. Therefore, your code must be prepared to deal with an exception whenever it creates an object on the heap or calls a function that does so.

After the THINK Class Library calls the Memory Manager directly to create a `Handle` or `Ptr`, it checks to see that the requested memory was actually allocated; if the request was not granted, the THINK Class Library throws an exception. Similarly, after the THINK Class Library attempts to load a resource, it checks to make sure the operation succeeded (there may not have been enough memory in which to load the resource), and throws an exception if it did not.

Typed Exception Handling

The THINK Class Library supports a subset of exception handling proposed by the C++ draft standard. Using this subset, your functions can throw and catch typed exceptions.

The macros provided for typed exception handling are:

Macro	Description
`try_`	Begins a try block. Equivalent to the standard `try` keyword.
`catch_ (class, name)`	Catches an exception of the indicated class. The variable `name` is defined within the handler as a pointer variable of the class. `catch_(A, a)` is equivalent to the standard `catch(A *a)`.
`catch_reference_ (class, name)`	Catches an exception of the indicated class. The variable `name` is defined within the handler as a reference variable of the class. `catch_reference(B, b)` is equivalent to the standard `catch(B& b)`.
`catch_no_instance_ (class)`	Catches an exception of the indicated class. `catch_no_instance_(A)` is equivalent to the standard `catch(A)`.
`catch_all_()`	Catches any previously uncaught exception. `catch_all_()` is equivalent to the standard `catch(...)`.
`end_try_`	Ends a try block and sequence of handlers. `end_try_` has no standard equivalent.

Table 9-1 Typed exception-handling macros

Macro	Description
`throw_(value)`	Throws the specified value as an exception. `throw_(a)` is equivalent to the standard `throw(a)`.
`throw_same_()`	Rethrows the current exception. `throw_same_()` is equivalent to the standard `throw()`.

Table 9-1 Typed exception-handling macros *(Continued)*

Defining exception classes

The typed exception macros can throw and catch only classes declared to be `TCL_EXCEPTION_CLASS` or classes derived from such classes. Each root class that is to be thrown should be declared in the following manner:

```
class MyException TCL_EXCEPTION_CLASS
{
    member functions and data as needed
};
```

Note

The `TCL_EXCEPTION_CLASS` macro actually derives the root class from a base exception class named _BR_CException. However, the inner workings of `TCL_EXCEPTION_CLASS` and other exception-handling macros are not further documented, as they will change or disappear when the compiler supports exception handling. No class or interface underlying these macros is guaranteed to appear in future releases of the THINK Class Library.

The CException class

All exceptions thrown by the THINK Class Library are of class CException, which is declared as:

```
class CException TCL_EXCEPTION_CLASS
{
public:
   CException(short err, long msg);
   short GetErr();
   long GetMsg();
   void SetErr(short err);
   void SetMsg(long msg);
};
```

See “Displaying error messages” later in this chapter to see how the `err` and `msg` values are used.

Using typed exceptions

The form of a typed try block with handlers looks like this:

```
try_
{
   anything that might throw an exception
}
catch_(Class1, var1)
{
   recover from exceptions of Class1
   var1 is a pointer to an exception object of that class
}
...
catch_all_()
{
   recover from any other exceptions
}
end_try_
```

When an exception is thrown, the catch expressions are examined in order. If the class of a catch expression is the same as (or a base class of) that of the thrown object, the corresponding handler is executed. `catch_all_` matches any thrown class. Handlers appearing after the first matching handler are not executed. If no catch expression matches, no handler is executed and exception handling resumes with the handlers of the next dynamically enclosing try block.

Note

The way in which catch expressions are matched imposes two conditions on the order of handlers. First, handlers for derived classes must precede any handlers for their base classes. Second, `catch_all_`, if provided, must be the last handler.

In `catch_` expressions, the second argument provides the name of a pointer variable that the exception handler initializes to point to a copy of the thrown object; in `catch_reference_` expressions, the second argument provides the name of a reference variable that is initialized in the same way. These variables let you obtain further information from the exception objects.

Unlike simplified exception handlers, typed exception handlers do not propagate automatically. That is, they do not rethrow the exception automatically. If you want a handler to pass control to a handler in the next active try block, you must explicitly call `throw_` or `throw_same_`.

The following example illustrates how to catch an exception thrown by the THINK Class Library, how to intercept and exit normally for any "file not found" errors, and how to rethrow any other exceptions thrown by the THINK Class Library.

```
try_
{
   ...
}
catch_(CException, thrown)
{
   if (thrown->GetErr() == fnfErr)
   {
      tell user
   }
   else
      throw_same_();
}
end_try_
```

Example 9-1 Catching THINK Class Library exceptions with typed handling

Returning

You must not `return` from within a try block or handler. Except for `throw_` and `throw_same_`, try blocks and handlers should follow structured programming rules and have a single entry and a single exit.

Exception specifications

In the draft C++ standard, exception specifications let you limit the types of exceptions a function can handle. The THINK Class Library does not implement exception specifications. All functions can handle exceptions of any type.

Simplified Exception Handling

The simplified exception-handling macros are compatible with previous releases of the THINK Class Library. Using the simplified exception macros, you can define try blocks and catch all thrown exceptions. The primary ways in which simplified exception handling differs from typed-exception handling are:

- You cannot access a thrown value or detect its type. Thus, only a single handler is allowed.
- By default, simplified exception handlers rethrow the current exception at the end of the block. If you do not want the exception rethrown, you must suppress it by executing the `NO_PROPAGATE` macro.

The macros provided for simplified exception handling are:

Macro	Description
`TRY`	Begins a try block.
`CATCH`	Catches all exceptions. Only one `CATCH` macro per `TRY` is permitted.
`ENDTRY`	Ends a try block and handler.
`NO_PROPAGATE`	Does not rethrow the current exception at the end of the handler. Valid only within a `CATCH` handler.

Table 9-2 Simplified exception handling macros

Note

The simplified exception handling macros are actually implemented using typed exception handling. The `Failure` function throws an exception with a value of class CException.

The basic form

This is the basic form of a simplified exception handler:

```
void Example()
{
    TRY
    {
        This is the normal flow.
        Anything here can throw an exception that will be
        caught in the handler.
    }
    CATCH
    {
        This is the handler.
        Clean up after errors here: close files, release
        memory, etc.
    }
    ENDTRY
}
```

Here is a concrete example:

```
void Example()
{
    CThing *object1 = new CThing;
    CThing *object2 = NULL;

    TRY
    {
        object2 = new CThing;
    }
    CATCH
    {
        TCLForgetObject(object1);
        TCLForgetObject(object2);
    }
    ENDTRY
}
```

Example 9-2 A simple exception handler

In Example 9-2, there are two places in which an exception can be thrown. First, if you create `object1`, and there is not enough memory, `new` fails and throws an exception. In this case, no recovery is necessary in the `Example` function, so the creation of `object1` is attempted outside a try block. The exception passes to the nearest handler further up the call chain.

In the second case, if you create `object2` and there is not enough memory to allocate and initialize it, `new` takes place within the try block and control passes to the `Example` function's `CATCH` handler. `TCLForgetObject(object1)` deletes `object1` (which is always non-`NULL` if control reaches the handler). `TCLForgetObject(object2)` either does nothing if `object2` is `NULL` (memory allocation failed), or deletes the object if it is non-`NULL` (failure occurred during initialization).

Retrying a try block

Previous releases of the THINK Class Library defined a `RETRY` macro that allowed a handler to restart its try block at the beginning. This macro is no longer supported. Instead, code such as the following can be used for the same effect:

```
Boolean retry = FALSE;
do
{
   TRY
      try block
   CATCH
   {
      ...
      if (condition)
      {
         retry = TRUE;
         NO_PROPAGATE;
      }
   }
   ENDTRY
}
while (retry);
```

Example 9-3 Retrying a try block

Exception Handling And Destructors

When an exception is thrown, the handler that catches it should be able to restore the program to a correct state. However, there are some clean-ups that handlers cannot perform when an exception is thrown by a constructor. These are:

- Calling stack object destructors
- Ordering of the calls to destructors

- Freeing memory in the heap allocated by `operator new`
- Using the stack to clean up the heap

With respect to the sequence of calling destructors, handlers cannot perform some clean-ups when an exception is thrown by a constructor:

- Calling destructors of sub objects
- Calling the destructor of a base class

Calling stack object destructors

The draft C++ standard requires that when an exception is thrown, destructors must be called for all live objects on the stack until the first handler is executed. The THINK Class Library supports this requirement with the `TCL_AUTO_DESTRUCT_OBJECT` macro.

This macro is used to declare root classes, as in the following:

```
class MyClass TCL_AUTO_DESTRUCT_OBJECT
{
public:
   MyClass();
   ~MyClass();
   ...
};
```

Example 9-4 Auto-destruct class declaration

The macro declares the class to be an auto-destruct class. When an object of this class or any derived class is constructed on the stack and an exception occurs later in its scope, the object's destructor will be called. For example, suppose MyClass's constructor and destructor are defined as follows:

```
MyClass::MyClass()
{
   gCounter++;
   TCL_END_CONSTRUCTOR
}

MyClass::~MyClass()
{
   TCL_START_DESTRUCTOR
   gCounter--;
}
```

Note

See the "Ordering of the calls to destructors" section below for an explanation of the `TCL_END_CONSTRUCTOR` and `TCL_START_DESTRUCTOR` macros.

Then the following behavior is seen:

```
void foo()
{
   gCounter = 0;
   MyClass myobj;     // gCounter==1
   do something that may throw exception
}                     // gCounter==0
```

Regardless of whether `foo` returns normally or throws an exception, `foo`'s caller always sees the value of `gCounter` as `0`.

Although the example just given is somewhat artificial, it does illustrate the behavior of auto-destruct objects. It also provides a model on which you can base your auto-destruct classes.

Classes without destructors do not require auto-destruct capability. However, declaring a class auto-destruct can make programs easier to maintain if that class has derived classes with destructors.

Ordering of the calls to destructors

When a constructor throws an exception, the draft C++ standard requires that destructors be called for all objects fully constructed by the constructor. Next, the destructor for the object's base class must be called before further exception handling is performed. The THINK Class Library performs the first operation—calling derived object destructors—automatically, provided the classes of the derived objects are declared as `TCL_AUTO_DESTRUCT_OBJECT`.

To call a base class's destructor, however, the THINK Class Library must know which constructors are finished and which destructors have already been called. You perform this by using the `TCL_END_CONSTRUCTOR` and `TCL_START_DESTRUCTOR` macros. The `TCL_END_CONSTRUCTOR` macro must be the last statement executed by a constructor of an auto-destruct class; the `TCL_START_DESTRUCTOR` macro must be the first statement in an auto-destruct class destructor. Both are written without trailing semicolons.

Freeing heap memory allocated by operator new

When you attempt to create an object on the heap using `operator new`, memory for the object is allocated on the heap. If the object's constructor throws an exception, your program does not have a pointer to this memory. Consequently, there is no way you can write a handler to recover the memory. If handlers were solely responsible for freeing heap memory, any exception in a dynamic object's constructor would cause a memory leak.

To prevent such leaks, the THINK Class Library provides the `TCL_NEW` macro, which you can use wherever you would have used `operator new`. For example, use:

```
MyClass *p = TCL_NEW(MyClass, ());
```

instead of:

```
MyClass *p = new MyClass();
```

The first argument to `TCL_NEW` is the name of the class; the second is the constructor argument list enclosed in parentheses. If the constructor has no arguments, `()` must be written for the second argument, as shown above.

The macro works by first calling global `operator new` to allocate memory for the object, and then calling a replacement `operator new` to invoke the object's constructor. For this reason, the macro cannot be used with classes that overload `operator new`.

Note

Classes used with TCL_NEW must be declared as auto-destruct classes. The THINK Class Library must determine the proper destructor to call prior to freeing the memory for the object; it can do so only if the TCL_END_CONSTRUCTOR and TCL_START_DESTRUCTOR macros are used.

Using the stack to clean up the heap

Auto-destruct stack objects can also clean up memory allocated on the heap without using try blocks. For example, this is the definition of the CWatchDesc class provided by the THINK Class Library:

```
class CWatchDesc TCL_AUTO_DESTRUCT_OBJECT
{
public:
   CWatchDesc(AEDesc *watch)
      { watched = watch; TCL_END_CONSTRUCTOR }
   CWatchDesc(AEDesc& watch)
      { watched = &watch; TCL_END_CONSTRUCTOR}
   ~CWatchDesc()
      { TCL_START_DESTRUCTOR;
         if (watched) TCLForgetDesc(watched);
      }
   void Keep() { watched = NULL; }
private:
   AEDesc *watched; // The watched descriptor
};
```

Example 9-5 CWatchDesc definition

The CWatchDesc class watches Apple event descriptor records, which contain a handle to heap storage. When the end of a block is reached in which a CWatchDesc object is declared, or when an exception is thrown in the block, the Apple event descriptor pointed to by the watched variable is disposed by the CWatchDesc destructor. At any time in the block, the program can elect to Keep the descriptor, in which case the watcher has no further effect on it.

This allows you to write code such as the following:

```
void MyCreateTwoDescs()
{
   AEDesc d1;
   AEDesc d2;

   MyCreateDesc(..., &d1);
   CWatchDesc watchd1(d1);
   MyCreateDesc(..., &d2);
   CWatchDesc watchd2(d2);
   operations that may throw exception
}
```

The alternative is longer code that is less easily understood:

```
void MyCreateTwoDescs()
{
   AEDesc d1;
   AEDesc d2 = {typeNull};

   MyCreateDesc(..., &d1);
   try_
   {
      MyCreateDesc(..., &d2);
      operations that may throw exception
      TCLForgetDesc(d1);
      TCLForgetDesc(d2);
   }
   catch_all_()
   {
      TCLForgetDesc(d1);
      TCLForgetDesc(d2);
      throw_same_();
   }
   end_try_
}
```

Using Exceptions

Use handlers only when necessary

You don't need a handler when you create an object if you know that the object will be destroyed by another handler. Here is an example:

```
void CMyApp::CreateDocument()
{
   CMyDoc *newDoc = NULL;

   TRY
   {
      newDoc = new CMyDoc(...);
      newDoc->NewFile();
   }
   CATCH
   {
      TCLForgetObject(newDoc);
   }
   ENDTRY
}

void CMyDoc::NewFile()
{
   BuildWindow();
   itsWindow->Select();
}

void CMyDoc::BuildWindow()
{
   CCoolPane *myPane;

   itsWindow = new CWindow(...);
   myPane = new CCoolPane(itsWindow, ...);
   ...
}
```

Example 9-6 A set of routines with only one handler

In Example 9-6, only `CMyApp::CreateDocument` needs a handler. The others do not. `BuildWindow` creates a window and immediately stores it in `itsWindow`, which is a data member of CMyDoc. `BuildWindow` also creates a pane, which is immediately added into the view hierarchy by the `CCoolPane` constructor. If an exception is thrown, it will be caught in `CreateDocument`'s handler, which calls `TCLForgetObject`. `TCLForgetObject` deletes `newDoc`, which deletes `itsWindow`. In turn, `itsWindow` deletes all panes that the window encloses.

Returning values

You must not return from a try block. If your function returns a value, it should return after ENDTRY. For example, look at CDataFile::ReadAll:

```
Handle CDataFile::ReadAll()
{
   register OSErr errCode;
   long length;
   Handle contents = NULL;

   TRY
   {
      FailOSErr(GetEOF(refNum, &length));
      contents = NewHandleCanFail(length);
      FailNIL(contents);
      FailOSErr(SetFPos(refNum,
                        fsFromStart, 0L));
      FailOSErr(FSRead(refNum, &length,
                        *contents));
   }
   CATCH
   {
      TCLForgetHandle(contents);
   }
   ENDTRY

   return contents;
}
```

Example 9-7 Returning a value from a function with try blocks and handlers

The ReadAll function in Example 9-7 is a good example of using the exception-handling mechanism. Note how all of the File Manager routines that return error codes are wrapped in calls to FailOSErr. This utility function checks the error return value; if the return value is not noErr, it throws an exception. Note also how the function uses NewHandleCanFail to allocate a handle. The latter function ensures that the Memory Manager won't use any of the memory reserves to get memory. The call to FailNIL throws an exception if the handle couldn't be allocated.

Displaying error messages

Handlers ensure that the program is correct in the face of failures, but they usually do not make the program succeed. When your program is unable to carry out an action the user has requested, you need to tell the user that the action couldn't be performed.

The most common approach is to display an alert, informing the user that the most recent operation was not carried out with a brief explanation. The THINK Class Library encapsulates this approach as the default behavior of THINK Class Library applications. The `Run` event loop has a CException handler that calls `ErrorAlert` to display an alert, at the same time displaying the error code with or without a short message specified for the exception. If the exception is not derived from CException, the `Run` handler displays an alert saying only that an exception occurred during the last operation.

The error code and message displayed are obtained from the thrown CException object. The error code is usually an `OSErr` value returned by a Toolbox function. Message values are constructed by calling the `SpecifyMsg` utility function, which packs the resource ID of a `'STR#'` resource, and an index into that resource, into a single long value. See the description of `ErrorAlert` in Chapter 126, "TCL Utilities," to learn how the function interprets the error code and message values.

CException's `SetErr` and `SetMsg` functions allow you to reset the error code and message values in a handler; you can then call `throw_same_` to propagate the exception with modified values. The primary use of this capability is to change the value of the error code to `kSilentErr` after you have completely handled the error, so that any higher-level handlers can be invoked without displaying an error alert to the user. You can also use `SetMsg` to change the alert the user sees, to provide an explanation more informative than a mere numeric code.

If you want a more elaborate error message for a given exception, catch the exception in one of your own typed handlers, display the desired message to the user, and call CException's `SetErr` member function to change the error code value to `kSilentErr`. Then rethrow the exception to allow other handlers to clean up. `kSilentErr` tells the handler in the application's `Run` event loop not to display an alert to the user, but instead to proceed silently to fetch the next event.

Exception Handling Functions

These functions facilitate throwing and responding to exceptions.

Exception throwing functions

Failure

```
void Failure (short error, long message);
```

Throws a CException class object containing the `error` and `message` values. If `error` is not `kSilentErr` and the exception is not handled by one of your handlers, the handler in CApplication's `Run` function displays an error alert.

Note

Previous versions of the THINK Class Library set the global variables `gLastError` to `error` and `gLastMessage` to `message`. Developers were invited to examine and modify these values. For compatibility, these variables are still set by `Failure` and examined by `CApplication::Run`. However, this usage is discouraged and the variables will be removed in a future version of the THINK Class Library. Use typed exceptions if you need to see the values of `error` or `message`. Instead of assigning values directly to these variables, call `SetErr` or `SetMsg` before rethrowing the exception.

For an explanation of the message parameter, see the description of `ErrorAlert` in Chapter 127, "TCL Utilities."

Fail

```
short Fail (short error);
```

Calls `Failure(error, 0)`.

FailEvent

```
short FailEvent ();
```

Calls `Failure(errAEEventNotHandled, 0)`.

FailEventMsg

```
short FailEventMsg (long message);
```

Calls `Failure(errAEEventNotHandled, msg)`.

Note that error codes and messages thrown during the processing of Apple events are not seen by users, but are passed back in the reply to the Apple event.

Error detection functions

Because most exceptions occur as a result of error codes returned by the Toolbox, the THINK Class Library provides a number of functions that test for errors and throw an exception only when one

is found. You should use these routines whenever you call the Resource Manager or Memory Manager, or any other Toolbox routine that returns an error code.

FailMemError

```
void FailMemError ();
```

Throws an exception if the Toolbox function `MemError` returns anything but `noErr`; in that case, it calls `Failure(err, 0)`, where `err` is the value returned by `MemError`.

FailNIL

```
void FailNIL (void *p);
```

Throws an exception if `p` is `NULL`; in that case, it calls `Failure(memFullError, 0)`.

FailResError

```
void FailResError ();
```

Throws an exception if the Toolbox function `ResError` returns anything but `noErr`; in that case, it calls `Failure(err, 0)`, where `err` is the value returned by `ResError`.

FailNILRes

```
void FailNILRes (void *p);
```

Throws an exception if `p` is `NULL`; in that case, it calls `Failure(err, 0)`, where `err` is the value returned by `ResError` if that is non-zero, or `resNotFound` otherwise.

FailOSErr

```
void FailOSErr (OSErr errCode);
```

Throws an exception if `errCode` is anything but `noErr`; in that case, it calls `Failure(errCode, 0)`.

Utility routines

SpecifyMsg

```
long SpecifyMsg (short strListID,
   short strIndex);
```

Creates a message value. `StrListID` is the resource ID of a `'STR#'` resource, and `strIndex` is the index into that resource. The value returned should be passed to `Failure` (or one of the routines that calls `Failure`). See the description of `ErrorAlert` in Chapter 127, "TCL Utilities."

SetFailInfo

```
void SetFailInfo (short newError,
   long newMessage);
```

Sets the global variables `gLastError` and `gLastMessage`. Sets `gLastError` to `newError`. Sets `gLastMessage` to `newMessage` if that global is zero, leaving it unchanged otherwise.

Note

This function will be removed in a future release of the THINK Class Library. If you want to change an active CException, use a typed handler and call `SetErr` or `SetMsg`. For more information, see the section "Displaying error messages" earlier in this chapter.

Run-Time Type Identification (RTTI)

Run-Time Type Identification (RTTI) is a feature that makes it possible to test and compare the class of objects at run-time. Although current C++ implementations do not provide this capability, the feature has been proposed in the C++ draft standard. The THINK Class Library implements a large subset of the functionality of the proposed RTTI standard.

RTTI support

Using RTTI, it becomes possible to do the following at run-time:

- Safely cast a pointer to a base class to a pointer to a derived class
- Test whether or not the class of an object is derived from a given class
- Test whether or not the class of an object is equal to a given class
- Test whether or not the classes of two objects are equal
- Obtain the name of a class as a null-terminated string
- Obtain the name of an object's class as a null-terminated string

Because RTTI is not yet implemented by the compiler, RTTI support is available only to those classes (and their instances) that have been explicitly declared to be RTTI classes. All classes in the THINK Class Library are RTTI classes that instantiate THINK Class Library applications or derive further classes. The procedure for making a class an RTTI class is explained in the section "Defining RTTI classes" later in this chapter.

Dynamic casting

The `TCL_DYNAMIC_CAST` macro provides safe casting of a base class pointer to a derived class pointer.

TCL Macro: `TCL_DYNAMIC_CAST(T, p)`
Proposed standard: `dynamic_cast<T*>(p)`

If the class of `*p` is `T` or a class derived from `T`, the macro returns `p` cast to type `T*`; otherwise, it returns `NULL`. Ordinarily, casting from a base class to a derived class is unsafe because `((T*)p)` returns garbage if `p` is not of type `T` or derived from `T`. However, it is safe to perform such a cast using `TCL_DYNAMIC_CAST` because the macro first performs a membership test.

`TCL_DYNAMIC_CAST` returns `NULL` if either `T` or the class of `*p` is not an RTTI class.

`TCL_DYNAMIC_CAST` can be used to test whether an object is of a given class or derived class. The following example illustrates this:

```
CPanemyPane;
...
if ( TCL_DYNAMIC_CAST(CButton, &myPane) )
{
   // This block always executes
}
else
{
   // This block never executes
}
```

Type identification
The type identification macros provide for run-time comparison of types.

TCL Macros: `TCL_TYPEID_FROM_TYPE(T)`
`TCL_TYPEID_FROM_POINTER(p)`
Proposed standard: `typeid(T)`
`typeid(*p)`

These macros return a type identifier. Two type identifiers can be compared for equality and inequality, as in the following sample comparisons:

```
TCL_TYPEID_FROM_TYPE(CButton)
== TCL_TYPEID_FROM_POINTER(p)

TCL_TYPEID_FROM_TYPE(CButton)
!= TCL_TYPEID_FROM_POINTER(p)
```

`TCL_TYPEID_FROM_TYPE(T)` returns `NULL` if `T` is not an RTTI class. `TCL_TYPEID_FROM_POINTER(p)` returns `NULL` if `*p` is not an RTTI class.

Obtaining class names

The class name macros let you obtain the name of a class or of an object's class as a string.

TCL Macros: `TCL_CLASSNAME_FROM_TYPE(T)`
`TCL_CLASSNAME_FROM_POINTER(p)`
Proposed standard: `typeid(T).name()`
`typeid(*p).name()`

Each of these macros returns the name of its argument as a `const char *` as the following example illustrates:

```
CButtonmyButton;
...
if (!strcmp("CButton",
      TCL_CLASSNAME_FROM_POINTER(&myPane)))
{
   // This block always executes
}
else
{
   // This block never executes
}
```

`TCL_CLASSNAME_FROM_TYPE(T)` returns `NULL` if `T` is not an RTTI class. `TCL_CLASSNAME_FROM_POINTER(p)` returns `NULL` if `*p` is not an RTTI class.

Note

`TCL_CLASSNAME_FROM_POINTER` is equivalent to `class_name` from earlier versions of the THINK Class Library.

The member, class_name, and new_by_name macros

The THINK Class Library provides the functions `member`, `class_name`, and `new_by_name` for compatibility with previous releases. They are implemented as macros that use the RTTI macros.

Function	Description
member(T, p)	Returns `TRUE` if the class of `*p` is `T` or derived from `T`; returns `FALSE` otherwise, or if either `T` or the class of `*p` is not an RTTI class. Equivalent to `!!TCL_DYNAMIC_CAST(T, p)`.
class_name(p)	Returns the name of the class of `*p` as a `char *`, if that class is an RTTI class. Equivalent to `TCL_CLASSNAME_FROM_POINTER(p)`.
new_by_name(T)	Returns a pointer to an object of class `T`, or `NULL` if `T` is not an RTTI class. Similar in effect to `new T`.

Table 9-3 Backward-compatible RTTI functions

To allocate memory for the object it creates, `new_by_name(T)` uses the class `T`'s `operator new` if it exists; otherwise, `new_by_name` uses the global `operator new` to allocate heap memory. The object is then constructed by the default constructor for class `T`, namely `T()`.

Note

Classes used with `new_by_name` should define a default consructor.

Defining RTTI classes

Every class that is to have RTTI capability or will be used with `member`, `class_name` or `new_by_name`, must be explicitly made an RTTI class. You do this by executing the `TCL_DECLARE_CLASS` macro in the `.h` file that declares the class, and one of the `TCL_DEFINE_CLASS` macros in the `.cp` file that defines it. Moreover, this must be done for every base class of the class (and not just immediate base classes), because the base classes of an RTTI class must themselves be RTTI classes.

TCL_DECLARE_CLASS requires no arguments and is written without a trailing semicolon. The define class macros are more complicated.

To define a class, you must specify the class name and the names of its immediate base classes, and whether the class is to be used with new_by_name. There are 10 macro variants, as follows:

- TCL_DEFINE_CLASS_M0(*class*);
- TCL_DEFINE_CLASS_M1(*class*, *base*);
- TCL_DEFINE_CLASS_M2(*class*, *base1*, *base2*);
- TCL_DEFINE_CLASS_M3(*class*, *base1*, *base2*, *base3*);
- TCL_DEFINE_CLASS_M4(*class*, *base1*, *base2*, *base3*, *base4*);

- TCL_DEFINE_CLASS_D0(*class*);
- TCL_DEFINE_CLASS_D1(*class*, *base*);
- TCL_DEFINE_CLASS_D2(*class*, *base1*, *base2*);
- TCL_DEFINE_CLASS_D3(*class*, *base1*, *base2*, *base3*);
- TCL_DEFINE_CLASS_D4(*class*, *base1*, *base2*, *base3*, *base4*);

The numbers 0 through 4 specify the number of immediate base classes. If a class has more than four base classes, you have to define your own macro. The M macros allow only the RTTI macros, member, and class_name to be used; the D macros (for "Dynamic") allow new_by_name as well.

There is also a separate set of macros for defining dynamic template classes:

- TCL_DEFINE_TEMPLATE_CLASS_D0(*name*, T);
- TCL_DEFINE_TEMPLATE_CLASS_D1(*name*, T, *base*);
- TCL_DEFINE_TEMPLATE_CLASS_D2(*name*, T, *base1*, *base2*);
- TCL_DEFINE_TEMPLATE_CLASS_D3(*name*, T, *base1*, *base2*, *base3*);
- TCL_DEFINE_TEMPLATE_CLASS_D4(*name*, T, *base1*, *base2*, *base3*, *base4*);

These template class definition macros work when there is only a single variant class. For example, to define CList<CView>, write:

```
TCL_DEFINE_TEMPLATE_CLASS_D1(CList,
CView, CPtrArray<CView>).
```

See the `CList_CView.cpp` file for examples of the template class definition macros.

The declarations and definitions of the THINK Class Library classes contain abundant examples of how to use the `TCL_DECLARE_CLASS` and `TCL_DEFINE_CLASS` macros, which you should examine to reinforce the previous discussion.

Note

If you declare a class to be an auto-destruct class and you want it also to be an RTTI class, then you should use one of the root class macros `TCL_DEFINE_CLASS_M0` or `TCL_DEFINE_CLASS_D0` to define it. Although your auto-destruct class is actually derived from _BR_Exception, you should treat it as a root class when defining its heritage for RTTI.

THINK C Symantec C++

Using Visual Architect

Part Three

Creating and Editing Views 10

This chapter explains how to create and initialize new views, and edit existing ones.

Contents

Creating Views

To create a new view, choose **New View** from the **View** menu. **New View** is enabled whenever there is a Views List window open.

When you create a new view, you are asked to specify some information about the view in the **New View** dialog box.

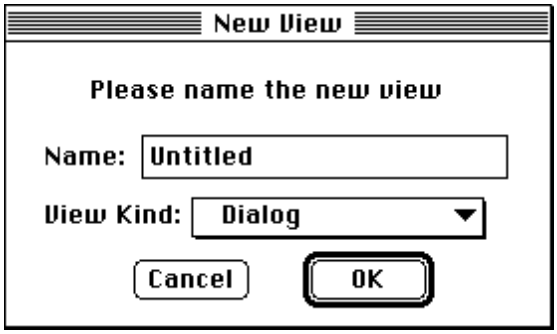


Figure 10-1 The New View dialog box

The name you specify becomes the name of the `'cVue'` resource created for the view. Visual Architect requires that the view name be unique within the project, so it can be used to construct identifiers that are also guaranteed to be unique. If you specify a name that already appears in the project, you will be asked to specify another name.

The View Kind defines the role the view plays in your application. It must be selected from the choices in the pop-up menu, as illustrated in Figure 10-2.

Figure 10-2 The View Kind pop-up menu

Choose the role that best fits the window or subview you want to design. Then click OK or press Return to create the new view.

The new view appears as a View Edit window, ready for you to begin populating with panes, as described later.

Kinds of views

There are nine kinds of views available. All are windows except the subview kind, which is a pane.

Dialog

A dialog view may be used in any context, usually to present and gather information. Dialog views are preset to modeless. A dialog view uses a CDialog window and is implemented using a CDialogDirector-derived director.

Floating Window

A floating window view is drawn in front of all non-floating windows. A floating window is often used as a palette, that is, a collection of drawing tools, colors, or patterns in which one is the current selection. A floating window is never activated or deactivated. A CSelector-derived panorama displays the palette and monitors selections. If a floating window is to be used as a menu, it should be implemented as a tear off menu view instead of as a floating window view. At run-time, all floating windows are created during initialization and hidden offscreen until needed. A floating window view uses a CWindow window and is implemented using a CFloatDirector-derived director.

Main Window

A main window view serves as the center of the user's attention. It displays the document in a document-editing application, or serves as a home display in a utility or database-type application. A main window is created when **New** or **Open** is chosen from the **File** menu. A document-based application usually defines just one main window, but it may define more if it edits more than one document type. A main window view usually uses a CWindow window, but can instead use a CDialog window if the programmer chooses. A main window view is always implemented using a CDocument-derived director. If the main window view is associated with a file (the default), the document class is derived from CSaver.

Modal Dialog

A modal dialog view is the same as a dialog view, as described earlier, but is preset as modal. As with a dialog view, a modal dialog view uses a CDialog window and is implemented using a CDialogDirector-derived director.

New… Dialog

A **New… Dialog** view is a modal dialog view used to select the type of document to create when **New** is chosen from the **File** menu. It is used only if the application needs to create more than one document type. The **New… Dialog** view uses a CDialog-derived window and is implemented using a CDialogDirector-derived director.

Splash Screen

A splash screen view is a dialog view that is displayed as soon as possible after the application launches, and removed as soon as initialization completes. A splash screen is modeless. A splash screen view uses a CDialog-derived window and is implemented using a CDialogDirector-derived director.

Subview

A subview is the only kind of Visual Architect view that isn't a window. A subview is a pane. Visual Architect allows you to define panes independent of the windows that display them. This allows you to reuse the same pane in multiple windows, or to nest the view hierarchy of a window arbitrarily deep while retaining the ability to edit it in a manageable way. A subview can scroll and can contain any other pane types, including other subviews.

Tearoff Menu

A tear off menu view is a particular type of floating window view that is used as a menu. A tear off menu uses a CWindow window and is implemented using a CTearOffMenu-derived director.

Window

A window view is a separate kind of view that functions as a plain window. A window view is used when a document needs more than one window to display its contents. The view directly supervised by the document is designated as a main window view, while all other views are window views. You can also use the plain window kind when none of the other kinds seems to meet your needs. A window view uses a CWindow window and is usually implemented using a CDocument-derived director.

Automatically-derived director class

Usually you have to open the **Classes** dialog box to define your own derived classes. But Visual Architect automatically defines a director class for each view you create. The default name given to the director class is C*viewname*, where *viewname* is the name you give the view. (You can change the class name in the **Classes** dialog box, but you cannot delete the class or change the base class.)

By way of background, every window in the THINK Class Library consists of at least two objects: a window and a director. The window object is responsible for display; it contains the panes

the user sees. The director object is responsible for control; it supervises the window and receives all commands and `ProviderChanged` calls generated by or for the window. The director serves as the intermediary between objects inside and outside the window, and often helps coordinate the actions of multiple objects within the window. For example, when the value of a control changes, the director receives a `ProviderChanged` call with a `controlValueChanged` reason code. If another control dims as a result of this change, the director-derived class is responsible for making it happen.

Deleting views

You can delete a Visual Architect view from your THINK Project Manager project by selecting the view name in the Views List window and choosing **Delete View** from the **View** menu. Note that this does not automatically remove the files from the THINK Project Manager project associated with, for example, the view's director class. These files must be removed by hand in the THINK Project Manager.

The View Edit Window

Although Visual Architect has many other capabilities, most work with Visual Architect is in the view editor, designing windows for your application.

Whenever you create a new view, as described in the previous section, or open an existing view from the Views List window (by choosing **Open View** from the **View** menu, or by double-clicking the name of the view), you are presented with a View Edit window like that shown in Figure 10-3.

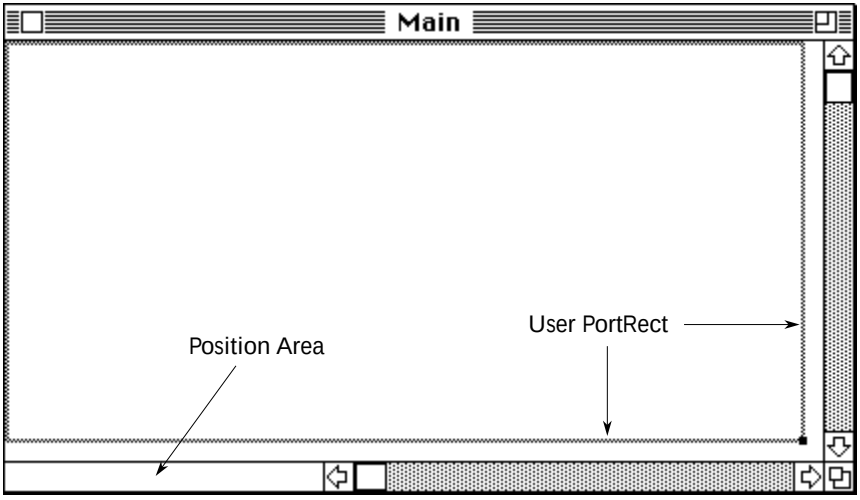


Figure 10-3 An empty View Edit window

The Visual Architect View Edit window is similar to the familiar MacDraw™ drawing window. The name of the view appears as the title of the window. Note that the view name is not necessarily the same as the title of the user window. The view name is set when the view is created and can be changed in the **View Info** dialog box.

User PortRect

The thick gray rectangle outline extending from the upper-left corner to the lower-right corner of the View Edit window, with a sizing handle in the lower-right corner, is the User PortRect, as shown in Figure 10-3. The User PortRect defines the actual size of the user window or subview you are editing. You can make the user window larger or smaller by dragging the sizing handle until the rectangle outline reaches the desired size. The Show Position option, described in “The Options Menu” section later in this chapter, is

helpful when resizing the user window. To see a window exactly as it will appear to the user, choose **Try Out** from the **View** menu (Command-Y) at any time.

The fact that the drawing area extends beyond the User PortRect means that you can edit objects that are outside the user window. This can be useful for editing panes that are only visible under certain conditions in the running application and are kept offscreen until needed.

If the user window has scroll bars, an additional one or two gray lines appear inside the User PortRect, showing the space occupied by the scroll bars.

Position Area

The white rectangle in the lower-left corner of the View Edit window is the Position Area, as shown in Figure 10-3. When the Show Position option is enabled (see "The Options Menu" later in this chapter), the coordinates of the currently selected object or objects are shown in this area.

Panes

Every graphical object inside a view is a pane—an object of a class derived from CPane. Panes are created by selecting a pane type from the Tool Palette, described later in this chapter. By selecting a tool, you can instantly drop or draw a pane object of the corresponding type into a view.

The View Info Dialog Box

Whenever the View Edit window is displayed, you can edit the attributes of the view as a whole by choosing **View Info** from the **View** menu.

The **View Info** dialog box lets you see the class of the window and director objects, and lets you specify other information as appropriate for the kind of view. You can edit the information in this dialog box at any time, not just before you begin laying out your windows.

The **View Info** dialog box is customized for each kind of view. Each kind of **View Info** dialog box is discussed in a separate section.

Dialog Info

The **Dialog**, **Modal Dialog**, **New... Dialog**, **Splash Screen**, and **Window Info** dialog boxes have the same layout, shown in Figure 10-4.

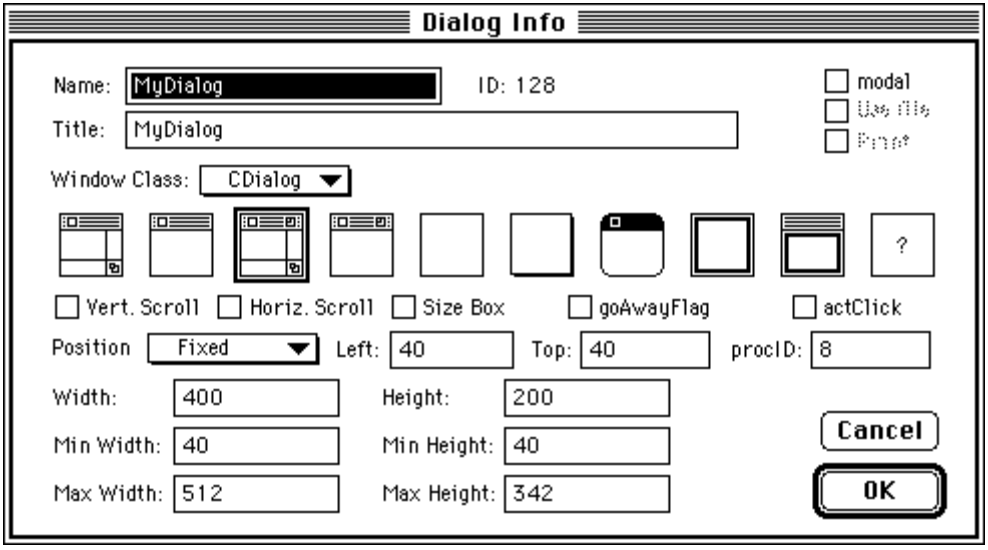


Figure 10-4 The Dialog Info dialog box

The dialog items are described below:

Name
The name of the view; also the name of the view resource.

ID
The view's resource ID. Automatically set by Visual Architect when the view is created.

Title
The title of the window or dialog.

Modal
If checked, the dialog is modal. Not relevant for window views.

Director Class
The name of the CDirector-derived (for the window view) or CDialogDirector-derived (for the other views) class.

Window Class
The name of the CWindow-derived (for the window view) or CDialog (for the other views) class.

Window Type
Click one of the ten window type icons corresponding to the known window `procID`s. If you select the icon with the question mark, you must supply the `procID` yourself (see "procID" below).

Vert. Scroll
Checked if the window or dialog has a vertical scroll bar. If checked, the main panorama of the window or dialog is enclosed in a CScrollPane.

Horiz. Scroll
Checked if the window or dialog has a horizontal scroll bar. If checked, the main panorama of the window or dialog is enclosed in a CScrollPane.

Size Box
Checked if the window or dialog has a size box. If checked, the main panorama of the window or dialog will be enclosed in a CScrollPane.

goAwayFlag
Checked if the window or dialog has a goAwayFlag. Not relevant if the view is a modal dialog.

Position
This indicates whether the view is initially centered, staggered, or has its upper-left corner at a fixed position.

Left, Top
The upper-left corner of the view, if Position is set to Fixed.

Width, Height
The width and height of the view.

Min Width, Min Height, Max Width, Max Height
The `sizeRect` for the window or dialog.

procID
The numeric `procID` used to define the window type; can also be set using the window type icons, as discussed above.

helpResID
Balloon Help resource ID. Automatically set by Visual Architect when you edit Balloon Help, which is discussed later in this chapter. Do not change this value unless you are certain of what you are doing.

actClick
Normally, if the user clicks a window or dialog in the background, the window or dialog is activated (brought to the foreground) but the pane that was clicked does not see the activate click. If actClick is checked, the pane does see the activate click.

Main Window Info

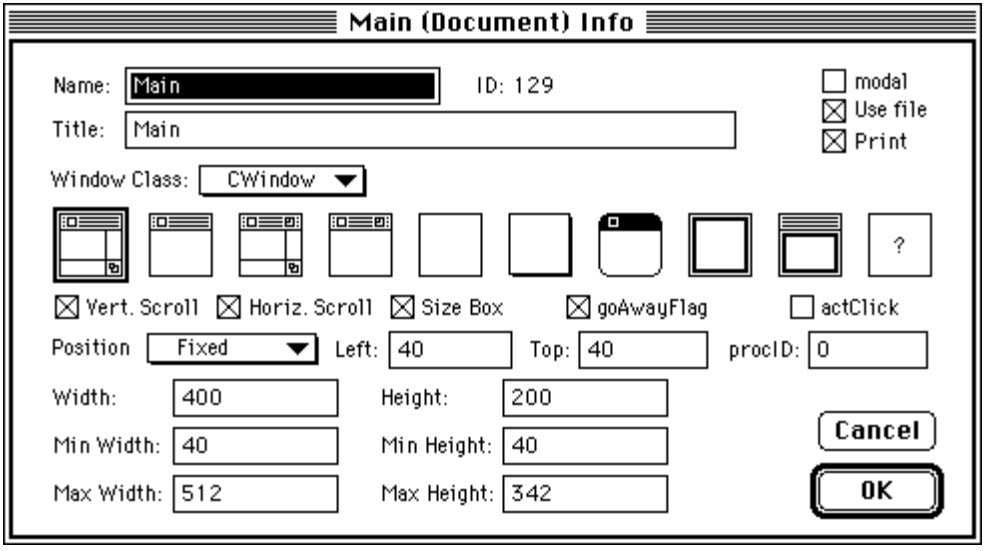


Figure 10-5 The Main Info dialog box

Most of the dialog items are the same as those of the **Dialog Info** dialog box. The items that are different or are used differently are described in the following sections.

Modal
Since the main window should generally be modeless, leave this check box unchecked.

Use file
If checked, the main window supports open, save, and revert file functions; the director class is derived from CSaver. If unchecked, the main window has no associated file; the director class is derived from CDocument.

Print
If checked, the contents of the window or dialog can be printed.

Window Class
Normally, a main window's window object is derived from CWindow. Use this pop-up menu if you have defined a CWindow-derived class you want to use for this window. A main window can also use a CDialog-derived window, but its director class is always derived from CDocument, not CDialogDirector.

Floating Window Info

The **Floating Window** and **Tearoff Menu Info** dialog boxes are the same.

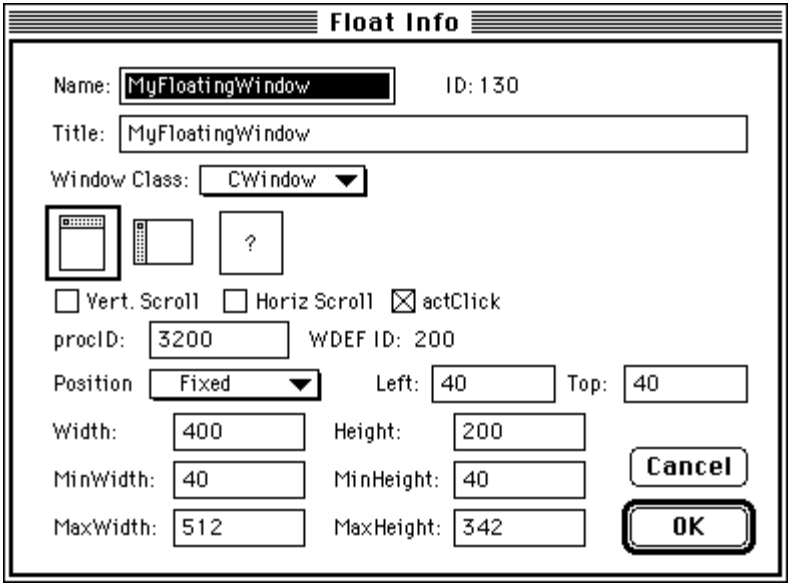


Figure 10-6 The Floating Window Info dialog box

Many of the dialog items in these dialog boxes are the same as those in the **Dialog Info** dialog box discussed above. The items that are different are described here.

Window Type
The window type icons represent the `procID` for the window. The standard floating window `WDEF` supports a drag bar at the top or left side of the window. Click one of the first two icons to select the standard floating window `WDEF`. If you click the icon with the question mark, you must supply the `procID` yourself.

Menu
The menu used for a tear-off menu. This is only enabled for a tear-off menu view.

WDEF ID
The resource ID of the `'WDEF'`. This value is set automatically according to the following formula:

```
WDEF ID = (procID - variation code) / 16
```

Since the floating window and tear-off menu views cannot use files, be modal, or print, these three check boxes are not present in the **View Info** dialog boxes.

Subview Info

The **Subview Info** dialog box is much smaller than the other **View Info** dialog box because this subview is not a window but a panorama (with an optional scroll pane).

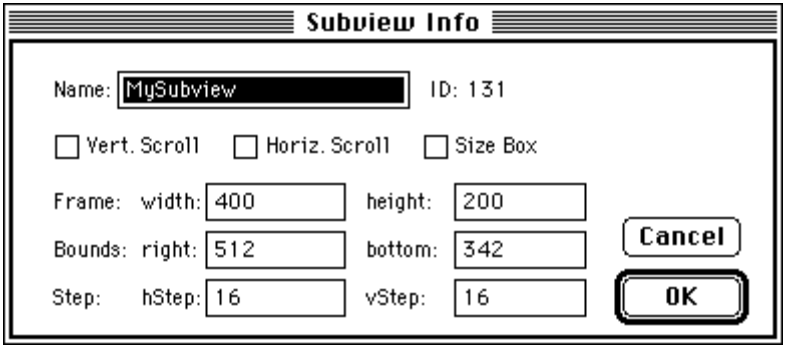


Figure 10-7 The Subview Info dialog box

Name
Name of the subview.

ID
The `'CVue'` resource ID.

Class
The name of the CPanorama-derived class for this subview, or None if you have not changed the subview's main panorama's class. See "Defining a subview class" below.

Vert. Scroll
Checked if the subview has a vertical scroll bar. If checked, the subview's main panorama will be enclosed in a CScrollPane.

Horiz. Scroll
Checked if the subview has a horizontal scroll bar. If checked, the subview's main panorama will be enclosed in a CScrollPane.

Size Box
Checked if subview has a size box. If checked, the subview's main panorama will be enclosed in a CScrollPane.

Frame width and height
The width and height of the subview.

Bounds right and bottom
The bottom-right corner of the panorama's bounds rectangle. The top-left value is always (0, 0).

Note

You can set the bounds of the subview's panorama directly in the **Subview Info** dialog box. This method is particularly useful for a scrolling subview.

Step hStep and vStep
The horizontal and vertical step values (number of pixels the panorama scrolls in a single step).

Defining a subview class
Unlike other views, no derived class is defined automatically when you create this view. This is because it is not strictly necessary to define a panorama-derived class in order to use a subview if the panes in the panorama do all the work. However, if you want to draw in your subview's panorama, receive commands, or override other member functions, you must derive your own panorama class and use it for the subview's panorama. Here's how to do it:

- Use the **Classes** dialog box to define your own derived class of CPanorama.

- In the View Edit window, with no panes selected, choose **Class** from the **Pane** menu and choose your derived class from the submenu.

The Tool Palette

The Tool Palette, shown below, is displayed when you click the **Tools** menu. The palette is a tear-off menu, so you can drag it to another position on the screen; it remains visible until you close it.

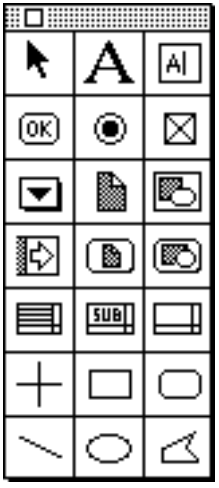


Figure 10-8 The Tool Palette

Select

Choosing the Select tool changes the cursor to an arrow, the standard Macintosh selection cursor. Select single pane objects in the usual way—by clicking them; select multiple panes by clicking an empty part of the drawing area and, holding the mouse button down, dragging the cursor until the selection rectangle encompasses the panes you want to select.

As described later in the section “The Options Menu,” Visual Architect offers two selection modes: Normal and Lazy Select.

Normally, after you use a tool once, the tool reverts to the Select tool. However, if you double-click a tool, the tool “sticks on,” so you can use it multiple times without reselecting. When a tool is stuck on, you can unstick it by selecting another tool.

For the Static Text tool, the cursor changes to an I-beam when the cursor is positioned over an activated View Edit window.

The crosshair cursor is used for all other tools.

Drop or draw

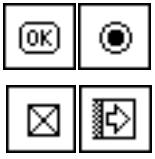
For all tools except Select and Static Text, after selecting the tool, you can click and drag in the View Edit window to create a pane object of arbitrary size, or you can create a new pane of a default size simply by clicking the mouse in the View Edit window. When you click with the Static Text tool, Visual Architect displays the I-beam cursor. You can type in as much text as you like; the new static text pane is always sized to the amount of the text.

Creating panes: the pane tools

Each pane tool is described separately. The class of the pane each tool creates is indicated in parentheses—for example, (CStaticText).

Static Text and Edit Text

The Static Text and Edit Text tools create static (CStaticText) and editable (CDialogText) text panes. For static text, a blinking insertion bar appears, letting you enter text once you click in the View Edit window to position the pane. For both types, you edit the text of an existing pane by selecting the pane and pressing Enter (or Return). This changes the cursor to an I-beam, which you then click within the text to position the cursor. Terminate text editing either by pressing Enter or clicking outside the pane. You can set the font, size, style, and color of text panes with the appropriate items in the **Pane** menu.



Push Button, Radio Button, Check Box, and Scroll Bar

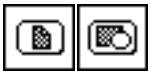
The Push Button, Radio Button, Check Box, and Scroll Bar tools create standard control pane objects with push button, radio button, check box, and scroll bar behavior (CButton, CRadioControl, CCheckBox, and CScrollBar classes).

Note

The CScrollBar class is specialized to deal with Apple's standard scroll bar control, which behaves rather idiosyncratically. Chances are you will have to override one or more CScrollBar member functions to handle another kind of slider.

Pop-up Menu
The Pop-up Menu tool creates standard pop-up menu panes (CStdPopupPane), which are used to display any menu you create. See Chapter 11, "Creating and Editing Menus and Commands."

Icon and Picture
The Icon and Picture tools create standard THINK Class Library icon (CIconPane) and picture (CPicture) pane objects in black-and-white or in color.



Icon Button and Picture Button
The Icon Button and Picture Button tools create icon (CIconButton) and picture (CPictureButton) multi-state button panes. These buttons can have a different appearance for on/off and highlight states, or can be color-highlighted or framed. Icon and picture buttons can be configured to act as push buttons, radio buttons, or check boxes. For example, the lock-down, pop-up buttons such as you find in a cassette recorder behave like radio buttons, while a light switch behaves like a check box.

List/Table
The List/Table tool creates scrollable lists or tables (CArrayPane). You can easily create lists of text, icons, or pictures.

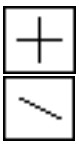
Subview
The Subview tool creates panes (CSubviewDisplayer) that refer to subviews that are displayed in this pane at run-time. The subview itself is a separate view and is edited in its own View Edit window. Subviews are scrollable panoramas that can contain any number of other panes.

Panorama
The Panorama tool creates generic scrollable panoramas (CPanorama). You use this to set aside sub-areas within your view's main panorama for special purposes, such as text editing or drawing. Unlike a subview pane object, a panorama object is not dynamic: It

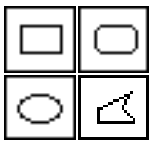
consists of only one pane, whose type is specified using the **Pane Info** dialog box described later in this chapter.

Note

You do not have to explicitly create a panorama object to make your view scrollable, since views are always enclosed in a panorama. Use the panorama tool only if you want to create a "sub-"panorama that does not contain the entire view.



Straight Line and Unconstrained Line
The Straight Line and Unconstrained Line tools create "straight" or unconstrained line graphic panes (both of the CLine class). The slope of a straight line is restricted to be a multiple of 45 degrees. Line pane objects can have any pen size, mode, color, or pattern.



Rectangle, Rounded Rectangle, Oval, and Polygon
The Rectangle, Rounded Rectangle, Oval, and Polygon tools create graphic panes—rectangle (CRectOvalButton), rounded rectangle (CRoundRectButton), oval (CRectOvalButton), or polygon (CPolyButton) objects, respectively.

The previous six graphic panes can have any pen size, mode, color, or pattern, and can be unfilled or filled with any pattern. Graphic panes are not just drawings; any graphic pane can act as a button of any kind. One very good use for these graphic buttons is to draw them over picture panes and make them invisible by setting the pen size to 0. In this way, since the graphic buttons can be of any arbitrary size and shape, you can make different parts of a picture click-sensitive. By setting the `clickCmd` for shape buttons using the **Pane Info** dialog box (described later in this chapter), each different picture part can respond differently to user clicks.

All pane objects are created at the same view nesting level, even radio buttons. The drawing order and tab order of panes is initially the order in which they are created. You can change the relative order of panes by choosing **Bring To Front** and **Send To Back** from the **Pane** menu. You can see the relative order of panes by choosing **Show Item Numbers** from the **Options** menu.

Panoramas (panes of class CArrayPane, CPanorama, CEditText, or any class you derive from these classes) are implicitly enclosed in a CScrollPane whenever they have a horizontal or vertical scroll bar or

a size box. You can change the attributes of a pane's scroll pane using the Pane Info window, described later in this chapter. If a panorama has no scroll bars or size box, the CScrollPane will not be present in the view resource or in the view at run-time.

Note

Although CDialogText is derived from CEditText, panes of the CDialogText class (those created with the Edit Text tool) do not contain a scroll pane.

In addition to using the Tool Palette, you can create static text or picture panes by pasting `PICT` or `TEXT` data from the Clipboard.

Editing Panes

You can alter almost all attributes of a pane. Many attributes can be changed through menu commands. Others can be changed using the Pane Info window, which allows you to edit the data members of the panes directly. This section will describe both approaches.

Editing a pane's text

You can directly edit the text of static text, edit text, push button, radio button, or check box panes without using a separate dialog box. Simply select the pane and press Enter. If your keyboard does not have an Enter key, press Return.

You can type in or edit any text in the pane, up to 32K characters. Press Return to add new lines to the text. When you are finished editing, press Enter or click outside the pane. Panes are resized automatically to fit the text.

Text in control panes is automatically centered vertically when you enter multiple lines of text. Push button text is centered horizontally as well. Static text panes more than one line high retain their original shape; to change the shape, reselect the pane and pull the knob in the lower-right corner to the desired shape. Edit text panes of any size retain their shape unless you extend the pane by typing more text than the box can hold; that is, you can make an edit text pane larger, but not smaller, by entering text into it.

Static and edit text panes are edited with the `wholeLines` attribute on, so there is no extra space at the bottom of the pane.

Font attributes

The font, size, and style of panes that display text can be changed by selecting the desired value in the **Font**, **Size**, or **Style** sub-menus in the **Pane** menu. Visual Architect alters the associated environment objects or, if appropriate, the TextEdit records associated with selected panes according to your selections.

Visual Architect creates controls using a control procedure that is sensitive to style, so font changes apply to push buttons, check boxes, and radio buttons, as well as text panes.

The Pane Info window

Double-clicking a pane, or selecting a pane and choosing **Pane Info** from the **Pane** menu (Command-L), opens a dialog box containing information about the currently selected pane. If a pane has an associated scroll pane, you can also choose **Scrollpane** from the **Pane** menu to bring up the Pane Info window for the scroll pane.

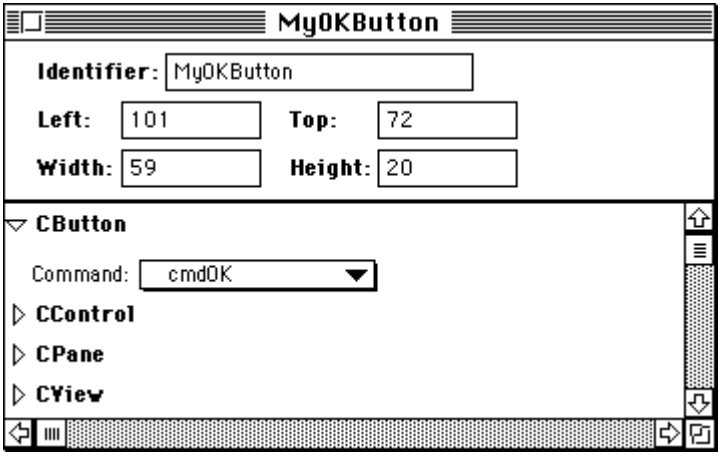


Figure 10-9 The Pane Info window

The Pane Info window is different for each pane class, but all are organized similiarly, as shown in Figure 10-9.

The title of the Pane Info window is the identifier you set by editing the Identifier edit text box at the top of the window or by choosing **Identifier** from the **Pane** menu. (Visual Architect gives each pane a default name when it is created.)

The Left, Top, Width, and Height edit text boxes at the top of the window let you change the position and size of the pane relative to the view's main panorama (that is, the enclosing window).

The remainder of the Pane Info window shows the pane's class hierarchy, beginning with the outermost-derived class of the pane and ending with the CView class. (Base classes of CView are not shown because they do not have any data members that can be edited.)

Clicking the right-pointing triangle to the left of a class name opens a sub-area, reveals the contents of that class, and pushes down the classes below it. Clicking a down-pointing triangle closes the subarea, hides that class's contents, and pulls up the classes below.

Each class sub-area contains the editable subset of the data members for that class. See Part Four of this guide, "THINK Class Library Reference," for definitions of these data members. The labels shown in the Pane Edit window are generally the same as the data member names in the THINK Class Library. Data members that do not appear in the subpanes are dynamically computed from the other data members when the view is actually run.

Changes made in the Pane Info window are reflected immediately in the target pane. For example, if you type a value into the `width` or `height` data members of the CPane class, the size of your pane changes while you are typing.

The Pane Info window is specific to a view. If you have more than one view open at the same time, you also can have more than one Pane Info window open. If you close a view, any Pane Info window associated with the view is automatically closed.

Although it takes only a few paragraphs to describe the hierarchical subpanes, they are one of the most powerful features in Visual Architect and worth studying further.

Cutting and pasting panes

You can cut panes and paste them to the same or to different views.

You can paste `TEXT` and `PICT` data from other programs into views. If you paste into a selected pane, the text or picture for that pane is replaced. If you paste `TEXT` and `PICT` data when no panes are selected, a new static text or picture pane is created.

Choose **Duplicate** from the **Edit** menu (Command-D) to make a copy of panes without involving the Clipboard. If you duplicate a pane and then move it, subsequent duplicate commands place the new panes the same distance apart.

Choosing the **Clear** command from the **Edit** menu deletes selected panes.

Cut, **Copy**, **Paste**, **Clear**, and **Duplicate** are undoable.

Editing Panoramas

For every window or subview, Visual Architect creates a panorama to hold all items you place in the view. If you select scroll bars or a size box for the view, the panorama is placed inside a scroll pane. This is all done automatically. In some cases you will want to change the class of the panorama so that it can perform specialized functions. To do this, define a CPanorama-derived class in the **Classes** dialog box. Then, in your view, make sure that no panes are selected and change the view panorama's class by choosing the **Class** hierarchical menu in the **Pane** menu and then choosing the class you want from the submenu.

The Options Menu

Visual Architect lets you customize the view editor's action by selecting one or more options from the **Options** menu.

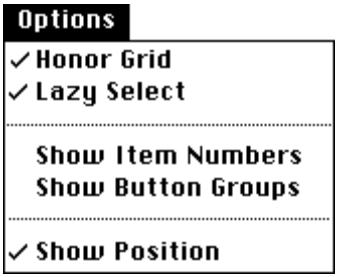


Figure 10-10 The Options menu

All options are set to the default settings, which may be changed in the **Preferences** dialog box. The **Preferences** dialog box is accessed by choosing **Preferences** from the **File** menu.

Honor Grid

When the **Honor Grid** menu item is checked, top-left and bottom-right coordinates of newly drawn or resized panes in the View Edit window are forced to be even multiples of the grid step size. When unchecked, panes can be placed at any pixel position in the View Edit window and can be any size.

The grid step size is initially four pixels and can be set to any positive power of two value in the **Preferences** dialog box.

Even when the grid is on, a pane may be positioned off the grid in several ways:

- The pane may have been created before the grid was turned on.
- With the Command key held down, the arrow keys move a pane in single-pixel steps. See the “Keyboard Shortcuts” section later in this chapter.
- Entering numeric values in the hEncl or vEncl data members text boxes of a pane in the Pane Info window moves the pane to that exact position.

When a pane is moved by dragging, it maintains its relative offset from the grid. When multiple panes are moved at the same time, they maintain their positions relative to each other.

Lazy Select

When the **Lazy Select** menu item is checked, the selection rectangle needs only to intersect a pane’s frame to select it. When unchecked, a pane is selected only if the selection rectangle completely encloses the pane’s frame (similar to how select is done in MacDraw).

One reason to use **Lazy Select** is that it is not always obvious what a pane’s frame size is—the frame may be larger than the drawing inside. On the other hand, when panes overlay, it is easier to select small panes on the inside with **Lazy Select** off.

Show Item Numbers

When the **Show Item Numbers** menu item is checked, the number of each pane in the drawing order is displayed in a small box in the upper-right corner of the pane's frame. When unchecked, item numbers are not shown.

Panes earlier in the drawing order (lower numbers) appear behind panes later in the drawing order (higher numbers).

The drawing order is the same as the tab order in dialog views.

To change a pane's item number, choose the **Bring To Front** and **Move To Back** items from the **Pane** menu. Newly created or pasted panes always appear at the end of the drawing order, that is, with the highest item numbers. Checking **Show Item Numbers** automatically unchecks **Show Button Groups**.

Show Button Groups

When the **Show Button Groups** menu item is checked, the button group of each button is displayed in a small box in the upper-right corner of the pane's frame. For buttons that are not a part of a button group, or non-button panes, the button group is shown as 0. When unchecked, button groups are not shown. Checking **Show Button Groups** automatically unchecks **Show Item Numbers**.

Show Position

When **Show Position** is checked, the coordinates of the currently selected pane or panes, or the coordinates of the User PortRect, are shown in the Position Area in the lower-left corner of the View Edit window. When unchecked, nothing is displayed in the Position Area.

Adding Balloon Help

To add Balloon Help to a pane, choose **Balloon Help** from the **Edit** menu while the pane is selected. The Balloon Help window appears as shown in Figure 10-11. The four balloons correspond to the different states of the pane object at run-time. You type into these balloons the text you want to appear when the user points to the

pane with Show Balloons active. Refer to *Inside Macintosh Volume VI* for details on the uses of these four balloon states.

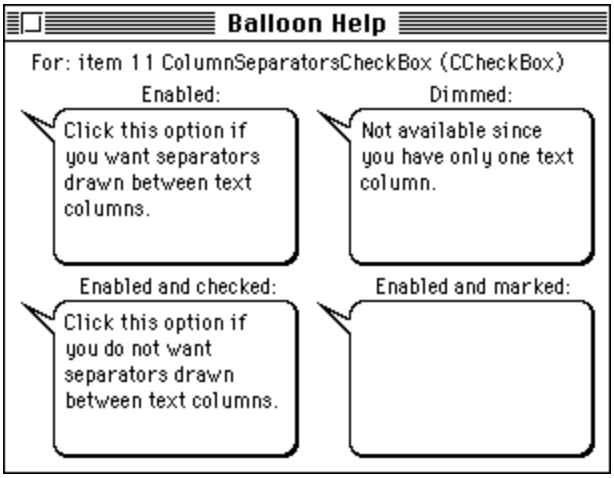


Figure 10-11 The Balloon Help window

The Balloon Help window is associated with whichever pane is selected in the active View Edit window. Thus, if you leave the Balloon Help window open and select another pane, the Balloon Help window is updated to reflect this new pane. The top of the Balloon Help window shows the current pane object.

Keyboard Shortcuts

Visual Architect provides several keyboard shortcuts for editing views.

Scroll keys

The Page Up key scrolls the View Edit window back one page. The Page Down key scrolls the window forward one page. The Home key scrolls to the upper-left corner. The End key scrolls the window to the lower-right corner.

Arrow keys

The arrow keys provide a quick way to move or resize a pane in small increments.

- The arrow keys move a pane one grid step in the desired direction, whether or not the grid is on.

- With the Command key held down, the arrow keys move a pane one pixel in the desired direction.
- With the Option key held down, the arrow keys resize a pane by moving the corresponding edge one grid step in the desired direction, whether or not the grid is on. For example, pressing Option-Left Arrow moves the left side of the selected pane(s) one grid step to the left, leaving all other sides in their original positions.
- With the Command and Option keys held down, the arrow keys resize a pane by moving the corresponding edge one pixel in the desired direction.

Creating and Editing Menus and Commands 11

This chapter describes how to create and edit menus and commands.

Contents

Editing Menus

This section explains how to create and edit menus, menu bars, and menu items.

Menus

Choose **Menus** from the **Edit** menu to create and edit menus. The **Menus** dialog box is shown in Figure 11-1.

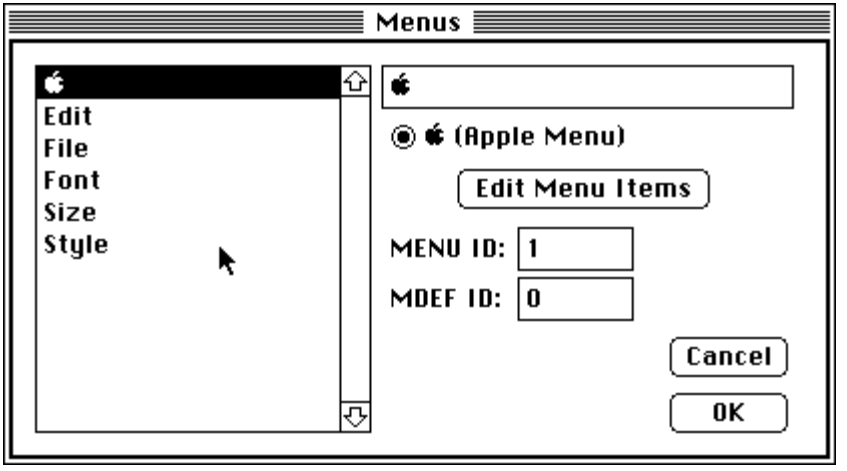


Figure 11-1 The Menus dialog box

The **Menus** dialog box lets you create and delete menus and change the title, menu ID, and MDEF ID associated with each menu. Menus in the **Menus** dialog box scrolling list appear in alphabetical order by title.

Editing a menu title. Select the menu in the scrolling list to bring its title into the edit text box in the upper right of the dialog box. Then edit the title.

Note

Visual Architect currently does not let you define a menu with no characters in the title. If you need such a menu, you have to edit the `'MENU'` resource in ResEdit.

To set the symbol as the title of the **Apple** menu, click the radio button below the edit text box.

As you type or correct a menu title, it is automatically put in alphabetical order in the list of menus.

Creating a new menu. Choose **New Menu** from the **Edit** menu (Command-K) or press Return. This creates a new, untitled menu. Give the menu a title by typing into the edit text box in the upper right of the dialog box.

Deleting an existing menu. Select the menu in the scrolling list and press Delete. If you delete a menu that appears in the menu bar, the menu is removed from the menu bar.

Editing menu items. Click the Edit Menu Items button to edit the contents of a selected menu. See the “Menu items” section later in this chapter.

Menu attributes

The following attributes can be set for each menu:

MENU ID. When you create or import a menu, Visual Architect automatically assigns that menu a unique resource and menu ID. You can change the menu and resource ID by entering a value in the Menu ID field. Visual Architect does not let you create a menu whose resource ID and menu ID are different. (This is the most common programming mistake associated with menus.) Seldom do you need to change this field.

MDEF ID. The MDEF ID field lets you use a custom menu definition procedure (MDEF). The system default value is 0. Seldom do you need to change this field.

Menu bars

Choose **Menu Bar** from the **Edit** menu to edit the menu bar. The **Menu Bar** dialog box, shown in Figure 11-2, is very similar to the **Menus** dialog box discussed in the previous section.

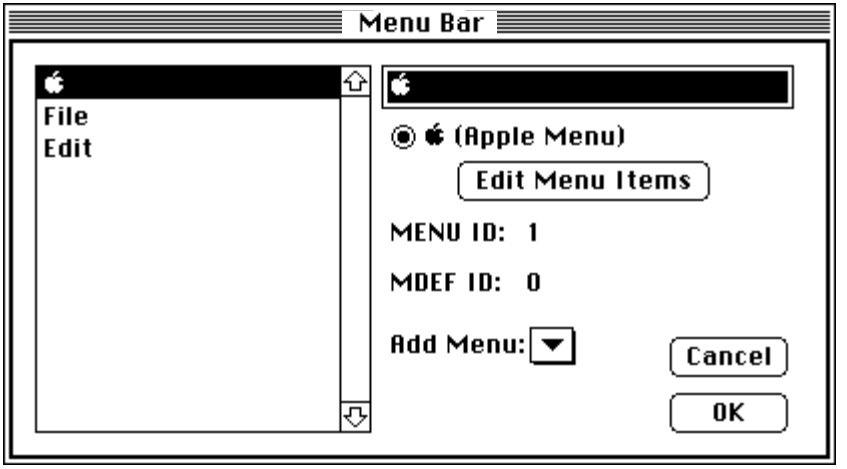


Figure 11-2 The Menu Bar dialog box

The **Menu Bar** dialog box lets you create menus, add and delete menus in the menu bar, and change the title and relative position of menus in the menu bar. Menus in the **Menu Bar** dialog box scrolling list appear in order of their appearance in the menu bar.

Editing a menu title. Select the menu in the scrolling list to bring its title into the edit text box in the upper right of the dialog box. Then edit the title.

To set the symbol as the title of the **Apple** menu, click the radio button below the edit text box.

Creating a new menu. Choose **New Menu** from the **Edit** menu (Command-K) or press Return. This creates a new, untitled menu and adds it to the menu bar. Give the menu a title by typing into the edit text box in the upper right of the dialog box.

Adding an existing menu. Select the name of the existing menu to be added to the menu bar from the Add Menu pop-up.

Removing a menu from the menu bar. Select the menu in the scrolling list and press Delete. Note that the menu will not be deleted, only removed from the menu bar.

Moving a menu up or down. Click the menu in the scrolling list and drag it up or down to the desired position.

The top-to-bottom position in the scrolling list corresponds to the left-to-right position of the menus in the menu bar displayed by your program.

Editing menu items
Click the Edit Menu Items button to edit the contents of a selected menu. See the "Menu items" section later in this chapter.

Menu attributes
Menu attributes, discussed in the "Menus" section earlier in this chapter, can be examined but not edited in the **Menu Bar** dialog box.

Menu bar ID
The menu bar is created with `'MBAR'` resource ID 1. This is the menu bar loaded by default when a THINK Class Library program starts up, and it is the only menu bar supported by the THINK Class Library. Use ResEdit to create additional menu bars if your program supports them.

Menu items

The **Menu Items** dialog box, shown in Figure 11-3, is opened from the **Menus** and **Menu Bar** dialog boxes either by selecting a menu and clicking the Menu Items button or by simply double-clicking the menu title in the scrolling list.

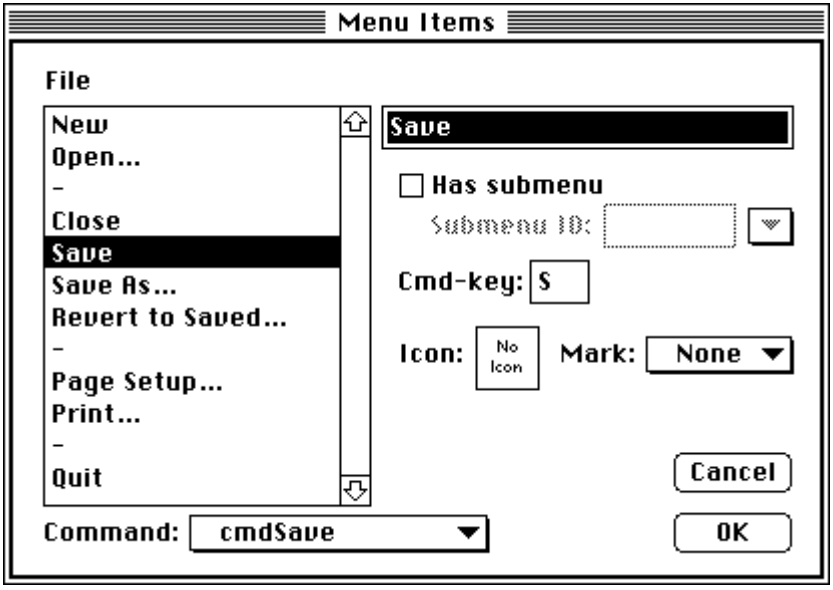


Figure 11-3 The Menu Items dialog box

The **Menu Items** dialog box lets you add, delete, move, and change menu items, as well as change the attributes associated with each menu item.

The edit text box in the upper right of the dialog box holds the title of the currently selected menu item from the scrolling list. Edit this box to change the title of the menu item.

Creating a menu item. Choose **New Menu Item** (Command-K) or press Return. This creates a new, untitled item at the bottom of the menu. Give the menu item a title by typing into the edit box in the upper right of the dialog box.

Deleting a menu item. Select the menu item in the scrolling list and press Delete.

Moving a menu item up or down. Select the menu item and drag it up or down to the desired position.

Menu item attributes
To edit the attributes of the currently selected menu item, edit or select the appropriate dialog field or button.

Has submenu. Check this box if the menu item is hierarchical and has a submenu.

Submenu ID. If the menu item has a submenu, this field is enabled. Enter the menu ID of the submenu. Because it is not easy to remember numerical menu IDs, you can optionally select the submenu from the pop-up menu to the right of the submenu ID edit text box.

Cmd-key. Enter the Command key, if any, associated with this menu item. Note that this item is dimmed if the menu item has a submenu.

Note

Do not use Command keys in pop-up menus. For a Command key in a CPopupPane to be recognized as a menu command, the pane must be the gopher or a supervisor of the gopher. Since a pop-up pane is not a candidate to be the gopher, and pop-up panes seldom supervise anything, these conditions will not be met.

Icon. Click the icon to set an icon for the menu. (The No Icon icon shows when an icon is not selected.) This brings up an icon selection dialog box that allows you to pick any `'ICON'` or `'SICN'` (small icon) resource in the project.

Mark. This command lets you choose the check mark to be used with the menu or choose None.

Note

Only the Chicago font in 12-point size provides the ✓ check mark character. For pop-up menus in other fonts or sizes, use the √ (square root) or • (bullet) character.

Command. Command lets you set the command to be sent when the menu item is selected. The pop-up shows the commands defined in the THINK Class Library file `Commands.h` and those you define in the **Commands** dialog box in Visual Architect.

Color

The **Menu Items** dialog box does not allow you to define color menu items. Use ResEdit or Rez if you need to do this. The Macintosh Human Interface Guidelines recommend coloring menus sparingly or not at all.

Adding Balloon Help

To add Balloon Help to a menu or menu item, choose **Balloon Help** from the **Edit** menu while the menu or menu item is selected. A Balloon Help window appears with four balloons labeled Enabled, Dimmed, Enabled and Checked, and Enabled and Marked, as shown in Figure 11-4. Enter into these balloons the text you want to appear when the user points to the menu or menu items with Show Balloons active. The four balloons correspond to whether the menu item is enabled, dimmed (disabled), enabled and checked, or enabled and marked (with a symbol other than a check mark—for example, a bullet), respectively, at the time the balloon is displayed

to the user. For menus, the behavior of the Enabled and Checked and the Enabled and Marked balloons is undefined but the balloons may be used by your application for special purposes.

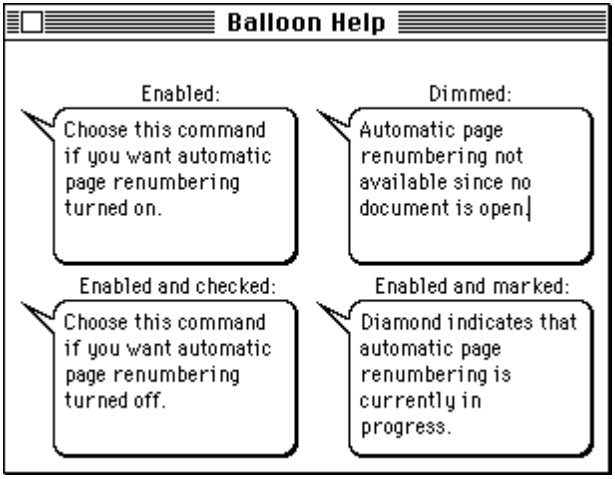


Figure 11-4 The Balloon Help window for menu items

Editing Commands

This section explains how to define THINK Class Library commands, as well as how to specify what the commands do and the classes that handle them.

Commands

Choose **Commands** from the **Edit** menu to open the **Commands** dialog box, shown in Figure 11-5.

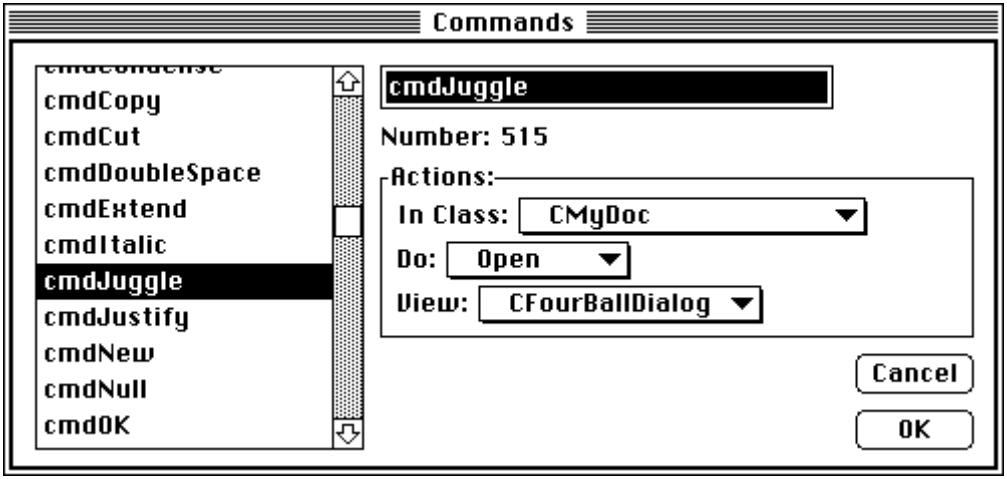


Figure 11-5 The Commands dialog box

The **Commands** dialog box lets you add, delete, and rename commands, as well as change the handlers and actions associated with each command.

The edit text item in the upper right of the dialog box holds the name of the currently selected command from the scrolling list. Edit this item to change the name of the command. Visual Architect does not let you change the names of commands predefined by the THINK Class Library.

The scrolling list, which is in alphabetical order by command name, lets you select a command to edit or delete.

Adding a new command. Choose **New Command** from the **Edit** menu (Command-K) or press Return to create a new, untitled command entry. Give the command a name by typing into the edit text box in the upper right of the dialog box.

Deleting an existing command. Select the item in the scrolling list and press Delete. Visual Architect does not let you delete commands predefined by the THINK Class Library.

Command attributes

You can edit the following attributes of the selected command:

Number. This is the number associated with the command. New commands are assigned numbers automatically by Visual Architect. Your program never deals with command numbers directly but with symbols for the command names you define, as described in the "Command handling in generated code" section below.

In class. Select one of your derived classes to handle the command. More than one class can handle the same command. Each one that does is checked when you pop up the In Class menu. To remove command handling from a class, uncheck it.

Do. In this pop-up, specify the action taken by the command handler defined in the In Class pop-up. Three actions are supported: Open, Call, and None, as described in the "Command handling in generated code" section below.

View. In this pop-up, specify the view object that is affected by the action. For an Open action, the view objects listed in the pop-up are the names of any dialog or window you have defined. The pop-up is disabled when Call or None is selected.

Command handling in generated code

Symbols for commands you define in Visual Architect are generated to the file `AppCommands.h`. You can `#include` this file in any of your classes that need to refer to commands.

A `DoCommand` member function is generated in each derived class on which you select to take action, or handle.

- If the action is Open, Visual Architect generates code in the `DoCommand` member function of the specified class to call a function named `Do`*Commandname* (where *Commandname* is replaced by your command name). Code generated for this function opens and selects the specified dialog or window. This generated code goes into the lower-level class.

- If the action is Call, Visual Architect generates the DoCommand case and the Do*Commandname* function into the lower-level class, as with Open. But the function does nothing; you must override it in your derived (upper-level) class.

- If the action is None, Visual Architect generates a case for the command switch in the DoCommand function, but the case does nothing. This prevents supervisors from receiving the command.

Command hints

Command editing is easy to describe, but it is quite powerful once you become accustomed to making objects send commands that other objects in the chain of command intercept and handle in their DoCommand functions. The following hints will start you in the right direction.

Remember the chain of command

The object that is to handle the command must be in the same chain of command as, and upstream of, the object that sends the command. Commands sent by menu bar menus can be handled by any object upstream of the gopher in the gopher's chain of command, while commands sent by a button, for example, are seen only by objects upstream of the button in the button's chain of command.

The only exception to the chain of command rule is a floating window. The floating window director relays all commands it receives and does not handle to the gopher (unless the gopher is inside the floating window). This starts the command up the gopher's chain of command, so commands from floating windows can behave like menu commands.

Commands from tear-off menus

For a tear-off menu, the grid selector sends a command that ultimately gets directed to the gopher. The command sent is the generic form of a menu command: a negative long integer whose absolute value has the menu ID in the high-order word and the ordinal of the selected item, starting from 1, in the low-order word. This means that you do not have to specify commands for menu items in tear-off menus. An added advantage of using selector commands is that the selector sends the same command, whether or not the menu is torn off.

Useful command actions

Experience shows that the most useful of the command actions is `Open`. `None` is seldom used because it usually does not matter that other objects in the chain of command see a command they do not handle. `Call` is also seldom used because it is easier for your upper-level class to override the `DoCommand` member function and handle the command case directly than to go through an extra level of indirection.

Commands in modal dialogs

In modal dialogs, the convention is to assign `cmdOK` to the OK button and `cmdCancel` to the Cancel button. When the dialog director receives either command, it returns from `DoModalDialog`. Visual Architect generated code always calls `DoModalDialog` with `cmdOK` as the default command.

Often you want to take one action if the users close a modal dialog with the OK button and another if they use the Cancel button. The simplest way to do this is to override the `EndDialog` member function in your view director class. Let CDialogDirector decide whether the command should dismiss the dialog box, then have your code take the appropriate actions. For example:

```
void MyDialog::EndDialog(long withCmd,
                  Boolean fValidate)
{
   Boolean dismiss = CDialogDirector::EndDialog(
                        withCmd, fValidate);
   if (dismiss)
      if (withCmd == cmdOK)
      {
         response to OK
      }
      else if (withCmd == cmdCancel)
      {
         response to cancel
      }
   return dismiss;
}
```

Buttons or other objects in modal dialogs should not send a `cmdClose` command.

Closing a modeless dialog

The Macintosh Human Interface Guidelines recommend that the only ways to close a modeless dialog box are with the Close menu item or the close box. (A modeless dialog box should always have a close box.) Therefore, just as in modal dialog boxes, buttons or other objects should not send a `cmdClose` command.

Editing Classes 12

This chapter explains how to create and define derived classes within Visual Architect.

Define a derived class whenever you want to extend the behavior of a director, window, or graphical object in your user interface. Visual Architect generates code for the derived class (see Chapter 13, “Generating Code”) that you can extend however you like.

Contents

Classes

Choose **Classes** from the **Edit** menu to display the **Classes** dialog box, shown in Figure 12-1.

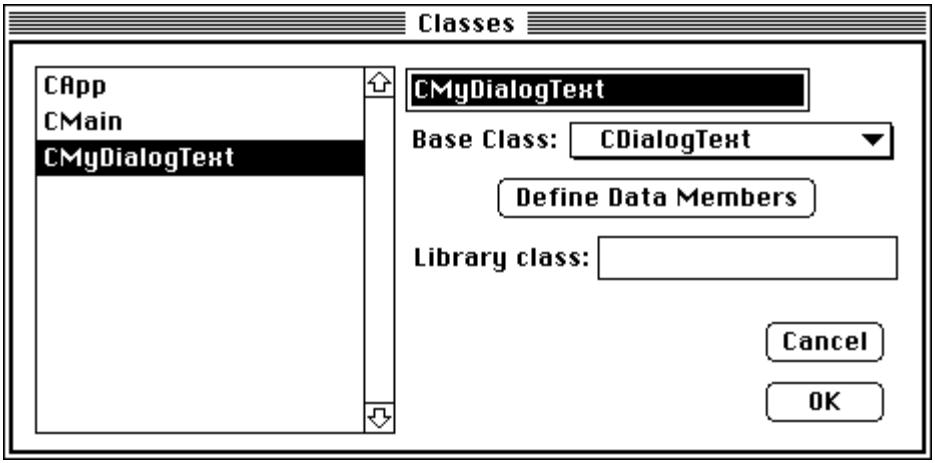


Figure 12-1 The Classes dialog box

The **Classes** dialog box lets you create and define new derived classes and delete and edit certain existing classes. Derived class names are listed in alphabetical order in the scrolling list.

Once you have defined a derived class, you can specify it as the class for your user interface objects. To do this, select the object and choose **Class** from the **Pane** menu and choose the derived class from the submenu.

The edit text field in the upper right of the dialog box holds the name of the currently selected class from the scrolling list. Use this box to change the name of a class.

Delete a class

Select the class in the scrolling list and press Delete. Classes created automatically by Visual Architect for the application class and view directors cannot be deleted.

Create a class

Choose **New Class** from the **Edit** menu (Command-K) or press Return. This creates a new, unnamed class. Give the class a name by typing into the edit text box in the upper right of the dialog box.

Set the base class

Pop-up the Base Class menu to select the base class of the class you are creating or editing. Only user-interface base classes supported by the THINK Class Library appear in the Base Class menu. The Base Class pop-up is disabled for the application class and for the director classes created automatically for views.

Set a library class

Ordinarily, a class you define in the **Classes** dialog box is derived directly from the base class shown in the Base Class pop-up. If you want to derive a class from one of your own library classes, type the name of that library class into the Library Class edit text box.

When Visual Architect generates code using the supplied macro files, it derives the lower-level class from your library class instead of from the class specified in the Base Class pop-up. Visual Architect assumes that your library class is derived from the base class shown in the Base Class pop-up. This class is used to select the appropriate macro file for code generation. Visual Architect also assumes that the name you specify for your library class is valid and does not conflict with any other class names.

Define Data Members

Clicking the Define Data Members button in the Classes dialog box brings up the Define Data Members dialog box, shown in Figure 12-2.

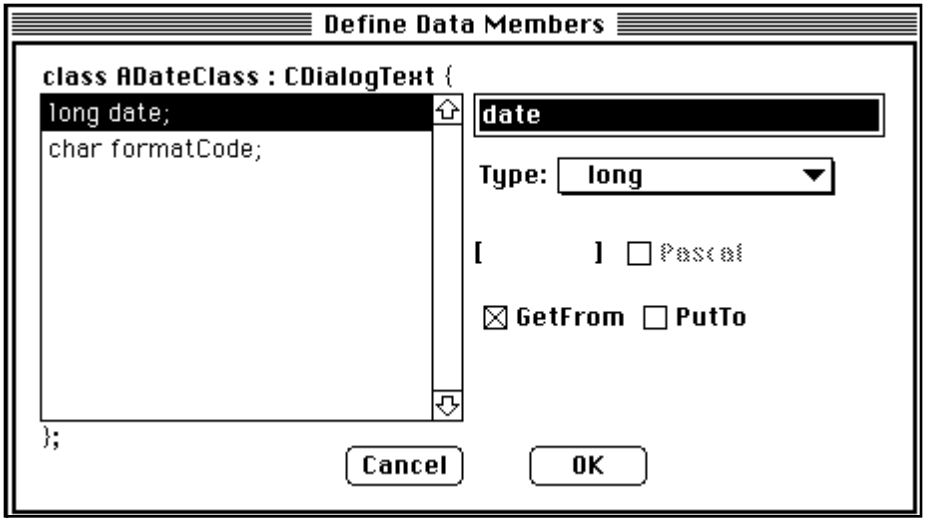


Figure 12-2 The Define Data Members dialog box

Actions enabled by Define Data Members dialog box

The **Define Data Members** dialog box lets you quickly define data members used to parameterize your derived classes. The values of data members of pane-derived classes can be defined within Visual Architect through the Pane Info window (see Chapter 10, “Creating and Editing Views”).

Add a new data member
Choose **New Data Member** from the **Edit** menu (Command-K) or press Return to create a unnamed member entry, and then give the data member a name by typing into the edit text box in the upper right of the dialog box. Next, select a type for the data member from the Type pop-up menu. Data members are listed in the scrolling list in the order in which you add them.

Delete an existing data member
Select the data member from the scrolling list and press Delete.

Reorder data members
Click the data member in the scrolling list and drag it to the desired position.

Change the type of a data member
Select the data member in the scrolling list and select a new type from the Type pop-up menu.

Define a char array
To define an array of `char` or `unsigned char`, create a data member of type `char` and type a number between the [ ] brackets in the edit text box . Check the Pascal box if you want an `unsigned char` array to be interpreted as a Pascal string (beginning with a length byte) rather than as a C string (ending with a zero byte).

GetFrom and PutTo
Check these boxes to have Visual Architect include the currently selected data member in the GetFrom and PutTo streams, respectively.

Visual Architect automatically generates the object I/O functions GetFrom and PutTo in derived classes. Views and subviews you define in Visual Architect are read as resources and interpreted by Object I/O. Object I/O input works by defining a GetFrom function in each derived class for which data members appear in the view

resource. Object I/O output works by defining a PutTo function in each derived class. See Chapter 8, “Using Object I/O,” for more information on this topic.

Limitations of Define Data Members dialog box

The **Define Data Members** dialog box is not intended to provide a general-purpose data design capability. Rather, it is a quick and easy way to parameterize your user interface classes. For example, whenever MyDialogText is used as the class of a dialog text box in a view, you can preset `magicNum` to any value you like.

It is possible to define data members of arbitrary types for your derived classes even though there are only a small number of primitive types in the Type pop-up. You can define as many data members as you like, of any type, in the header file of your upper-level class.

ADateClass example

The example in Figure 12-2 illustrates the benefits and limitations of this feature. Suppose you need a dialog text box that accepts and displays dates in a variety of formats. Since the THINK Class Library doesn’t have such a class, you have to derive it from CDialogText.

Your derived class, ADateClass, needs at least two data members: a long integer to hold the date and an `unsigned char` to hold the desired data format leading to the definitions shown in Figure 12-2. With Visual Architect you can define these members easily, and you are able to give them initial values in the **Pane Info** dialog box, discussed in Chapter 10, “Creating and Editing Views.” Visual Architect automatically generates the code to read your initial values from the view resource.

However, if you are defining these members by hand, you might prefer to give the `dateFormat` member a type of `DateForm`, which is equated to `unsigned char` in `Packages.h`. This signifies that this member holds values from the standard `DateForm` enumeration. But because Visual Architect is not a compiler, you are restricted to built-in types.

The two data members in ADateClass are generated in the lower-level class of your derived class. You can still add members, such as handles and pointers to other objects or structures, to the upper-level class. These members will be initialized to zero when the object is created from a view resource.

Effect of Changes on Existing Views

You can freely add, delete, and change derived classes. If a derived class is currently in use, any change to that class necessitates a corresponding change in all views containing objects of that class. Visual Architect makes these changes automatically.

If a class name is changed, the new name replaces the old in the affected views. If a class is deleted or its base class is changed, current objects of that class revert to the former base class. For example, if you define a class MyButtonClass with base class CButton and use several MyButtonClass objects in views, the objects revert to CButton class when you delete MyButtonClass.

If you add or delete a data member or change its type, the old data in existing objects of the class is discarded and the new data is initialized to binary zeros.

Generating Code 13

This chapter describes how to generate the code for a complete application with Visual Architect.

Contents

13 Generating Code

Generating the Code Files

To generate code, choose **Generate** or **Generate All** from the **THINK Project Manager** menu, shown in Figure 13-1. **Generate** generates code only for files that need updating as a result of changes you have made in Visual Architect since you last generated files. **Generate All** generates code for all files. **Generate** and **Generate All** commands will not affect generate-once files that have already been generated.

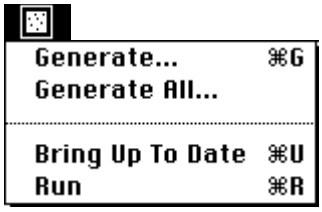


Figure 13-1 The THINK Project Manager menu in Visual Architect

If you have checked the Confirm Saves option in the **Preferences** dialog box, you are asked if you want to save your Visual Architect project before generating. Click OK to save and generate. Otherwise, the project will be saved automatically.

While code generation is underway, you are informed of progress in a dialog box like the one shown in Figure 13-2.

Figure 13-2 The Code Generation Progress dialog box

The **Code Generation Progress** dialog box shows the name of each file as it is being generated. To stop generation in progress, click the Stop button or press Command-. (Command-Period). Stopping generation does not remove any files already generated.

When all files have been generated, the dialog box changes to **Updating Project**, and your THINK Project Manager project is

updated. All generated files not already in the THINK Project Manager project are added; generated files already in the project are marked for recompilation. If you have a generated file open in the THINK Project Manager, it is closed without saving its contents to prevent your project from containing the wrong version of the file.

Setting Up Generation

Code generation in Visual Architect is controlled by a macro file that you select. By default, Visual Architect looks for a macro file named `GenerateTCLApp`, first in the THINK Project Manager tree and then in your project tree.

`GenerateTCLApp` contains macros that tell Visual Architect which code files to generate. The code files are generated to a folder named `Source`, located inside the folder that contains your `Visual Architect.rsrc` file.

You can change which macro file Visual Architect uses to generate code by choosing **Set Generate File** from the **File** menu and selecting the macro file in the standard **File Open** dialog box.

Structure of the Generated Code

Visual Architect supplies a set of macro files capable of generating a complete THINK Class Library application. Before you generate code for the first time, you should fill out the **Application Info** dialog box and define a Main view.

The following sections explain the code generation strategy followed by `GenerateTCLApp` and the other macro files in the `Macros` folder supplied with Visual Architect.

Preserving your code when regenerating

Code generation doesn't happen only once. Each time you make changes to views, classes, commands, and so on in Visual Architect, new files must be generated and/or previously generated files must be regenerated. For each class you define, Visual Architect generates files defining the class and implementing the behavior you specified in Visual Architect. You can then modify these files by hand to add additional behavior.

To preserve code that you add by hand through multiple passes of code generation, the standard macro files generate two kinds of files: lower-level and upper-level. The terminology comes from visualizing

the class tree with the root class at the bottom, the lower-level class above it, and the upper-level class on top.

For every derived class you define, Visual Architect produces an upper-level file defining the derived class, and a lower-level file defining the immediate base class of that class. The THINK Class Library base class of the derived class, the lower-level derived class, and the upper-level derived class form a sort of class sandwich, with the lower-level class in the middle. For example, if you derive a class named CMyButton from the THINK Class Library CButtonProc, Visual Architect constructs the following class hierarchy:

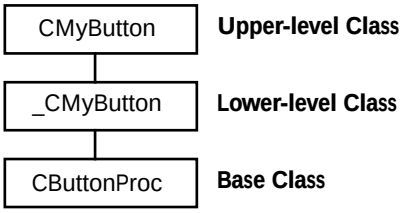


Figure 13-3 Sample split-level class hierarchy

Lower-level files are rewritten each time Visual Architect generates source code for a class. Upper-level files are generate-once files and are written only the first time Visual Architect generates source code for a class. Subsequent generation does not touch upper-level files.

You fill in the upper-level files with your own code. You can add or remove member functions or data members and fill out the generated functions with your own code. This split-level approach adds only a few bytes to your program size, but it makes your generated code immune to subsequent changes that you make in Visual Architect.

Lower-level files are not restricted to files defining derived classes. Visual Architect also generates header files as lower-level files to define symbols. An example is the `viewnameItems.h` file, which contains symbolic names for the panes in a view.

Lower-level files contain as much of the information that might change as a result of editing in Visual Architect as possible. While this technique shields your code from the effects of most iterative design changes, there are several things to watch out for, as described in the following paragraphs.

Deleting or renaming panes

If your handwritten code uses a symbolic name for a pane, and you change the name or delete the pane, your code obviously must change. Usually, the compiler warns you of errors of this sort.

Changing class names

If you change the name of one of your derived classes, an entirely new set of upper- and lower-level files is generated for the class, and you have to copy your code from the old upper-level files to the new ones.

The names of lower-level files for derived classes begin with the characters `x_` (x underscore). Otherwise, they are the same as the names of the corresponding upper-level files.

Generated files

The files listed in the table below are generated for every application, regardless of size.

File name	Description
`x_`*appclass*`.cp`	The lower-level source code for your application class.
`x_`*appclass*`.h`	The header file for the lower-level application class.
appclass`.cp`	The upper-level source code for your application class.
appclass`.h`	The header file for the upper-level application class.
viewname`Items.h`	Header files defining symbols for panes in each view.
`AppCommands.h`	The header file defining symbols for the commands you define in Visual Architect. The `AppCommands.h` file is regenerated each time.
`References.cp` `References.h`	The files defining a `ReferenceStdClasses` function that force a reference to every THINK Class Library that can do Object I/O.

Table 13-1 Code files generated by Visual Architect

Inside Macro Files

This section tells how to modify and write your own macro generation files. If you always use the macro files supplied with Visual Architect, you can skip the rest of this chapter.

Macro files that drive code generation are ordinary text files. Macro files shipped with Visual Architect were created in THINK Project Manager. You can use any text editor to create the files as long as they have file type TEXT.

The fact that generated macro files are ordinary text files makes them easy to customize. To customize, you need to know the Visual Architect macro language, which is described in the remainder of this chapter.

Visual Architect macro language

The Visual Architect macro language embeds macro processing statements in ordinary C++ source text. Visual Architect uses the $ character to indicate the start and end of a Visual Architect macro. When a macro file is used to generate source code, all text outside a macro is copied to the output file without change. Text inside a macro, including the $ delimiters, is replaced with zero or more characters that depend on the current value of the macro.

There are two kinds of macros: statement macros and expression macros. A statement macro begins with the first non-whitespace character on a line and occupies the entire line. The entire line is removed from the output. In computer jargon, a statement macro and the line it is on are replaced by a null string. An example of a statement macro is:

```
$do windows$
```

An expression macro can appear anywhere on a line. Visual Architect evaluates the expression, computes a character string value, and replaces the macro with the value. An example of an expression macro appearing in a source code line is:

```
itsWindow->SetTitle("$window.title$");
```

Expression macros can appear anywhere in any line, including within statement macros. In the latter case, expression macros are evaluated and replaced before the statement macro is interpreted. If

you want to use the $ character as ordinary text, you must double it, as in:

```
CopyPString("\p$$2.99", str);
```

which copies the Pascal string `"$2.99"`. Doubling works inside C++ macros as well.

Statement macros

Each statement macro begins with a keyword immediately following the leading `$`, which tells what kind of statement it is. The keyword must be one of the following:

```
define
do
else
elseif
end
generate
if
pop
push
```

Each statement type is described below.

define statement

Use `define` to set the value of a macro variable to a desired value. The format of a define statement is:

`$define` *variablename expression*`$`

The variable name must begin with an alphabetic character and must be less than 31 alphabetic or numeric characters. The underscore character `"_"` is considered an alphabetic character.

The expression may be any combination of variable names, constants, and operators that would be permitted within the body of an expression macro. Visual Architect evaluates the expression just as it would an expression macro and assigns the result to the macro variable.

Once a variable is defined, it holds that value until it is redefined in another `define` statement or until the variable is removed by a `pop` statement.

Certain variables are predefined by Visual Architect to hold character strings, records, or arrays. If you define a variable with the same

name, it hides the predefined variable until the variable you defined is popped. In general, it is a good idea to choose variable names that do not conflict with those listed in the “Predefined variables” section later in this chapter.

do statement

Use a `do` statement to iterate over each element in an array. The format of a `do` statement is:

```
$do arrayvariablename$
```

The array variable name must be one of those predefined by Visual Architect.

The `do` statement operates by repeatedly scanning the text between the `do` statement and the closest matching `end` statement, once for each array element. If there are no elements in the array, or if the array variable is undefined, the text between the `do` and `end` statements is skipped.

Before starting each iteration, two variables are automatically defined: the array element name and the variable `i`. The variable `i` is assigned the index of the current array element, from 1 through the number of elements in the array.

All array variable names defined by Visual Architect are plural, for example, `windows`. For each array variable, the array element name is singular, for example, `window`. Usually, an array element is a record, which means it has subvalues. Subvalues can be accessed by placing a period after the element name, followed by the name of the record item. For example:

```
window.title
```

is the title of the “current window,” that is, the current value of the `window` variable.

The following example shows a `do` loop, which generates a C++ macro for every window:

```
$do windows$
#define WIND$window.id$ $window.resid$
$end$
```

else statement
Use an `else` statement to reverse the sense of a preceding `if` or `elseif` statement and to conditionally expand one or more lines. The format of the `else` statement is:

```
$else$
```

If the value of the preceding `if` or `elseif` expression is `TRUE`, the `else` statement causes lines between it and the nearest matching `end` statement to be skipped. If the value is `FALSE`, the lines following `else` are expanded.

elseif statement
Use an `elseif` statement to reverse the sense of a preceding `if` or `elseif` statement and conditionally expand one or more lines. The format of the `else` statement is:

```
$elseif expression$
```

If the value of the preceding `if` or `elseif` expression is `TRUE`, the `elseif` expression is not evaluated; statements between `elseif` and the nearest matching `end` statement are skipped. If the value is `FALSE`, the expression is evaluated and the `elseif` statement behaves like an `if` statement.

end statement
Use an `end` statement to end the scope of a `do`, `if`, `else`, or `elseif` statement. The format of the `end` statement is:

```
$end comment$
```

If a comment appears in the `end` statement, there must be at least one whitespace character after the `end` keyword.

generate statement
Use a `generate` statement to create a new output file. The format of the `generate` statement is:

```
$generate outputfilename macrofilename
    [once] [keep] $
```

The `generate` statement creates a new output file with the specified name and fills it with text produced by expanding the specified macro file.

The output file name can contain a folder name. If no folder by that name exists in the Source folder, a folder is created inside the Source folder. If a folder with that name already exists, the file is placed inside that folder.

If the keyword once is specified, the file is not generated if a file with the same name already exists in the specified folder. Use once for files you want to modify by hand, so that they will not be overwritten the next time code is generated. If you want to regenerate a once file, move the previously generated file out of its folder.

If the keyword keep is specified, variables defined while the file is being generated are retained. By default, these variables are popped automatically. The use of the push and pop statements is described shortly.

if statement
Use an if statement to conditionally expand one or more subsequent lines. The format of the if statement is:

```
$if expression$
```

If evaluating the expression results in a value other than zero or results in an empty string, the expression is considered to be TRUE and the lines following the if statement until the next matching else, elseif, or end statement are expanded. Otherwise, the expression is FALSE, and all text between the if statement and the next matching else, elseif, or end is skipped. You can nest if statements.

pop statement
A pop statement discards all variables defined since the last push statement. The format of the pop statement is:

```
$pop$
```

The variables defined automatically by a do statement are popped automatically when the matching end statement is processed. Similarly, variables defined during processing of a generate statement without a keep keyword are popped automatically as soon as the file is generated. Otherwise, variables you define remain defined until you explicitly pop them. You rarely need to use push and pop, but in some circumstances they are useful in preventing inactive variables from wasting memory.

push statement
Use a push statement to establish the context for a subsequent pop statement. The format of the push statement is:

```
$push$
```

Expression macros

An expression macro evaluates an expression, producing a character string whose contents replace the macro in the output text.

The result of an expression macro is not rescanned for macros; it goes directly to the output (or statement macro) without being interpreted further.

An expression is composed of variable names, constants, and operators. The value of any expression or subexpression is a character string. Certain operators may cause string operands to be converted to long integers; the numerical result of the operation is then converted back to a string.

An undefined variable is interpreted as a null string.

Operators
Macros may contain the following operators which, except for automatic string-to-long coercion, are defined like their C and C++ language counterparts:

```
+ - * / << >> ~ & | ^ && || < <= > >= == != !
```

In addition, the dot operator (. operator) is used to select a record item.

Expressions may be put in parentheses.

Constants
String, character, and integer constants are permitted in expressions. Long integer and hexadecimal forms are also allowed. Octal constants are not supported.

Predefined variables

Visual Architect predefines a number of variables, which are listed in the table below. For further definition of records or arrays that have record elements, see “Record types” later in this chapter.

Variable name	Predefined as...
`abbrevDate`	The current date as an abbreviated date string
`app`	The application record
`barmenus`	An array with one menu record element for each menu you define to be part of the THINK Class Library menu bar
`classes`	An array with one class record for each class displayed in the **Classes** dialog
`classeschanged`	`1` if **Generate All** is chosen or if any classes changed since last generate; otherwise, `0`
`commands`	An array with one command record for each command you define in the **Commands** dialog
`commandschanged`	`1` if **Generate All** is chosen or if any commands changed since last generate; otherwise, `0`
`copyright`	The copyright text you specify in the **Application Info** dialog
`date`	The current date as a date string
`dialogs`	An array with one view record for each modal or modeless dialog view you define
`documents`	An array with one view record for each document (Main) view you define
`floats`	An array with one view record for each floating window view you define
`longDate`	The current date as a long date string

Table 13-2 Predefined variables

Variable name	Predefined as...
`menus`	An array with one menu record for each menu you define
`newdialog`	A view record for the **New... Dialog**, if any
`popmenus`	An array with one menu record for each pop-up menu you define
`shortDate`	The current date as a short date string
`splash`	A view record for the splash screen
`submenus`	An array with one menu record for each hierarchical submenu you define
`subviews`	An array with one view record for each subview you define
`tearoffs`	An array with one view record for each tear-off menu you define (the menu associated with the view is in the record variable `menu`)
`time`	The current time
`views`	An array with one view record for each view you define (all views)
`viewschanged`	`1` if **Generate All** is chosen or if any views changed since the last generate; otherwise, `0`
`windows`	An array with one element for each window view you define
`year`	The current year

Table 13-2 Predefined variables *(Continued)*

Record types

A macro record is a variable with named elements. Like a C or C++ `struct`, a record element always has a record name with a period, followed by an element name. For example:

```
view.changed
```

A `do` statement implicitly defines a record on each iteration that corresponds to the `i`th element of the array controlling the `do`. For example:

```
$do views$
```

defines a record variable named `view` within the scope of the `do`. On each iteration, `view` is set to a different view record. Elements of this record are referred to within the `do` as. For example:

```
view.actions
```

The following sections define each record type separately:

App record
An app record contains application information you set in the **Application Info** dialog box.

Record element	Description
`changed`	`1` if application changed since the last generate; otherwise, `0`
`class`	A class record for the application class
`copyright`	The value of the copyright string
`docType1`	The first document OSType
`docType2`	The second document OSType
`docType3`	The third document OSType
`docType4`	The fourth document OSType
`filetypes`	An array of filetype records for the application's document types; same values as `docType1`–`docType4`
`name`	The application name
`Name`	Application name, with the first character capitalized
`numfiletypes`	The number of file (document) types defined
`signature`	The application signature (creator code)

Table 13-3 Application record description

Action record
An action record describes a command action defined in the **Commands** dialog box. Each action defines a handler class, what

happens, and, in the case of an Open action, the class of the view that is opened.

Record element	Description
`class`	A class record for the class of the view created as a result of Open; `NULL` if not an Open action
`className`	The name of the class that handles this action
`command`	A command record for the command that causes the action
`view`	A view record for the view created as a result of an Open action; `NULL` if not an Open action
`what`	Which action: None, Call, or Open

Table 13-4 Action record description

Class record

A class record describes each class displayed in the **Classes** dialog box.

Record element	Description
`actions`	An array of action records for each action handled by this class
`basename`	Name of nominal base class or base of library class, if defined
`changed`	`1` if class changed since last generate; otherwise, `0`
`kind`	An ordinal denoting the kind of class; class kinds are defined in the `GenerateTCLApp` macro file
`members`	An array of member records describing the data members defined in Visual Architect
`nactions`	The number of actions handled by this class
`name`	The name of the class
`nmembers`	The number of data members defined in Visual Architect
`supername`	The name of the base class; either a user or a THINK Class Library class
`view`	A view record for the class; `NULL` if the class is not the director class of a view

Table 13-5 Class record description

Command record

A command record describes each command you define in the **Commands** dialog box.

Record element	Description
`actions`	An array of action records for the command
`nactions`	The number of actions caused by the command
`name`	The command name
`Name`	The command name with the first letter capitalized
`num`	The command number

Table 13-6 Command record description

File type record

A file type record contains a single, four-character file type, for example, `TEXT`.

Record element	Description
`type`	The four-character type

Table 13-7 File type record description

Menu record

A menu record describes each menu you define in the **Menus** or **Menu Bar** dialog box.

Record element	Description
`ID`	The `'MENU'` resource ID
`mdefID`	The ID of the menu definition procedure
`menuID`	The `menuID` inside the `'MENU'` resource (Visual Architect always sets this the same as the `'MENU'` resource ID.)
`menuItems`	An array of menu item records for this menu
`name`	The name of the `'MENU'` resource
`nMenuItems`	The number of items in this menu
`title`	The menu title

Table 13-8 Menu record description

Member record
A member record describes each data member you define in the **Define Data Members** dialog box. A data member declaration can be generated from the macros:

```
$member.type$  $member.name$ $member.elem$;
```

Record element	Description
`elem`	The number of elements in an array data member as a string enclosed in brackets, for example, `[15]`; `NULL` if not an array
`getfrom`	`TRUE` if member to be included is generated `GetFrom` function (object I/O)
`ispascal`	`1` if a Pascal string; `0` if a C string
`name`	The name of the data member
`nelements`	The number of elements in an array data member, for example, `15`; `0` if not an array
`putto`	`TRUE` if member to be included in generated `PutTo` function (object I/O)
`type`	The type of the data member as a character string
`typecode`	A numeric code indicating the type of the data member, numbered from `1` through `12`: `Boolean`, `char`, `double`, `float`, `long`, `short`, `Str31`, `Str255`, `Str63`, `unsigned char`, `unsigned long`, `unsigned short`

Table 13-9 Member record description

Menu item record
A menu item record describes each menu item you define in the **Menu Items** dialog box.

Record element	Description
`command`	A command record for the command sent by the menu item; `NULL` if no command
`icon`	The resource ID of the `'ICON'`, reduced icon, or `'sicn'` resource
`key`	The key character if nonzero and outside the range `0x1B` through `0x1F`; otherwise, `NULL`

Table 13-10 Menu item record description

Record element	Description
`mark`	The mark character if nonzero and outside the range `0x1B` through `0x1F`; otherwise, `NULL`
`name`	The name (string value) of the item
`reduced`	`1` if reduced; `0` if not
`sicn`	`1` if icon is a `'sicn'` resource; `0` if not
`submenu`	A menu record for the submenu if the item has a hierarchical menu; otherwise, `NULL`
`submenuID`	The resource ID of the submenu if the item has a hierarchical menu; otherwise, `NULL`
`title`	The title of the menu

Table 13-10 Menu item record description *(Continued)*

Pane record

A pane record describes each pane in a view.

Record element	Description
`active`	`1` if pane is initially active; `0` if not
`autoRefresh`	`1` if `autoRefresh` set for pane; `0` if not
`baseclass`	The name of the standard THINK Class Library or user library class that is the immediate base of the (lower-level) pane class
`canBeGopher`	`1` if `canBeGopher` set for pane; `0` if not
`classname`	The name of the pane class
`height`	The pane's `height` value
`helpResIndex`	The pane's `helpResIndex` value
`hEncl`	The pane's `hEncl` value
`hSizing`	The pane's `hSizing` value
`ID`	The pane's `ID` value
`identifier`	The identifier of the pane as defined in the **Identifier** dialog or defined automatically by Visual Architect
`Identifier`	The identifier with the first letter capitalized
`kind`	An ordinal that identifies the superclass category of the pane; the ordinals are defined in the `GenerateTCLApp` macro file
`usingLongCoord`	The pane's `usingLongCoord` value
`vEncl`	The pane's `vEncl` value
`visible`	The pane's `visible` value

Table 13-11 Pane record description

Record element	Description
vSizing	The pane's vSizing value
wantsClicks	The pane's wantsClicks value
width	The pane's width value

Table 13-11 Pane record description *(Continued)*

Point record
A point record contains a point.

Record element	Description
h	Horizontal coordinate
v	Vertical coordinate

Table 13-12 Point record description

Rectangle record
A rectangle record contains a rectangle.

Record element	Description
bottom	Lower vertical coordinate
left	Upper horizontal coordinate
right	Lower horizontal coordinate
top	Upper vertical coordinate

Table 13-13 Rectangle record description

View record
A view record contains information about a view defined in Visual Architect. Record elements indicated as "(windows only)" are defined only for views that are windows, that is, not subviews.

Record element	Description
actClick	1 if the view's panes see the click that activates the window (windows only)
actions	An array of action records with one element for each action performed by this view, as set up in the **Commands** dialog
centered	1 if the view is centered; 0 if not
changed	1 if the view has been changed since the last generate; 0 if not
directorclass	The class name of the view's director (windows only)

Table 13-14 View record description

Record element	Description
`directorkind`	A numeric code indicating the kind of director that supervises the view; director kinds are defined in the `GenerateTCLApp` macro file (windows only)
`fixedPosition`	`1` if the view's window is to be opened at a fixed position on the screen; `0` if not (windows only)
`floating`	`1` if the view is a floating window; `0` if not (windows only)
`goaway`	`1` if the view's window has a goAway box; `0` if not (windows only)
`height`	The window height (windows only)
`helpResID`	The resource ID of the `'hmnu'` resource for this view (windows only)
`horizscroll`	`1` if the view's panorama is enclosed in a scrollpane with horizontal scroll bars
`ID`	The view's `'CVue'` resource ID
`isdialog`	`1` if the view is a modeless dialog; `0` if not (windows only)
`isdocument`	`1` if the view is a document (Main view); `0` if not (windows only)
`ismodal`	`1` if the view is a modal dialog; `0` if not (windows only)
`issubview`	`1` if the view is a subview; `0` if a window
`iswindow`	`1` if the view is a window; `0` if a subview
`items`	An array of pane records for the items in this view; same as `panes`
`maxHeight`	The maximum height (windows only)
`maxWidth`	The maximum width (windows only)
`menu`	The `'MENU'` resource ID; `0` unless view is a tear-off menu
`minHeight`	The minimum height (windows only)
`minWidth`	The minimum width (windows only)
`modal`	1 if the view is modal; `0` if not (windows only)
`nactions`	The number of actions performed by this view; `0` if none
`name`	The view name as defined in Visual Architect
`Name`	The view name with the first letter capitalized

Table 13-14 View record description *(Continued)*

Record element	Description
`nitems`	The number of items in this view; same as `npanes`
`npanes`	The number of panes in this view; same as `nitems`
`panes`	An array of pane records for the panes in this view; same as `items`
`panorama`	A pane record for the view's main panorama
`position`	A point record containing the view's initial position; the value of `position` is only meaningful if `fixed` is `TRUE`
`print`	`1` if the view is printable; `0` if not (windows only)
`procID`	The window's `procID` (windows only)
`scrollpane`	`1` if the view's main panorama is enclosed in a scrollpane; `0` if not
`sizebox`	`1` if the view's panorama is enclosed in a scrollpane with a size box; `0` if not
`staggered`	`1` if the view's window is initially staggered; `0` if not (windows only)
`title`	The view's window title (windows only)
`usefile`	`1` if the view uses a file; `0` if not (windows only)
`vertscroll`	`1` if the view's panorama is enclosed in a scrollpane with a vertical scroll bar; `0` if not
`viewkind`	A numeric code indicating the kind of view; view kinds are defined in the `GenerateTCLApp` macro file
`WDEFid`	The view's window definition procedure `'WDEF'` ID (`procID` / 16) (windows only)
`width`	The window width (windows only)
`windowclass`	The class name of the view's window (windows only)
`windowkind`	A numeric code indicating the view's window kind; window kinds are defined in the `GenerateTCLApp` macro file (windows only)

Table 13-14 View record description *(Continued)*

THINK C
Symantec C++

THINK Class Library Reference

Part Four

CAbstractText 14

Introduction

CAbstractText is an abstract class that provides you with a template for creating a pane that displays text.

Heritage

Base Class	CPanorama
Derived Classes	CEditText

Using CAbstractText

CAbstractText is an abstract class for creating panes that display text. A text pane is usually the panorama in a scroll pane, so the text can be scrolled through. The class supports word wrap, which you can control with the line width argument to CAbstractText's constructor. The `Specify` member function lets you specify whether the user can edit and copy your text pane's text. The `DoCommand` member function handles all common text-editing commands such as cutting, copying, pasting, font selection, and line spacing.

To handle the **Undo** command, CAbstractText creates task objects for many common commands, and tells them to perform or undo their tasks by calling their member functions. CTextEditTask, described in Chapter 118, handles typing, cutting, copying, and pasting. CTextStyleTask, explained in Chapter 120, handles font, size, and style commands.

When you perform a task that can be undone, the name of the **Undo** command changes. For example, it becomes **Undo Typing** after the user types. The **Undo** labels for CAbstractText are stored in the resource `'STR#' 130`. The index of the first string is stored in the class data member `cFirstTaskIndex`. By default, `FirstTaskIndex` is 1. If you move the strings to a different place in `'STR#' 130`, you should set this data member to the index of

the first string. If you don't use **Undo** labels, set this data member to 0. The labels must be in the following order: Typing, Cut, Copy, Paste, Clear, Formatting.

Because it's an abstract class, you can't create an instance of CAbstractText. You must create an object of a derived class, such as CEditText or CStyleText. Those classes are based on the Macintosh TextEdit (TE) routines and become slow if they contain over 4000 characters. If your text pane needs to handle more text, derive your own class from CAbstractText.

Most of CAbstractText's functions do nothing. They are placeholders for the functions that derived classes must override. The rest of CAbstractText's member functions call them. These are the functions to override:

```
DoClick                 SetTextPtr
GetTextHandle           SetFontNumber
SetFontStyle            SetFontSize
SetTextMode             SetAlignCmd
SetAlignCmd             SetSpacingCmd
GetHeight               GetCharOffset
GetCharPoint            GetTextStyle
GetCharStyle            FindLine
GetLength               TypeChar
SetSelection            GetSpacingCmd
GetSelection            GetNumLines
CopyTextRange           PerformEditCommand
InsertTextPtr           Dawdle
```

The following two functions assume that the text in your pane is stored in a single contiguous buffer. If it isn't, you should override these functions:

```
GetCharBefore           GetCharAfter
```

Data Members

The following data members are public. `itsTypingTask` can be used by any class. `cFirstTaskIndex` is a static data member that only CAbstractText and its derived classes should use.

Data member	Type	Description
`itsTypingTask`	`CTextEditTask*`	Active typing task
`cFirstTaskIndex`	`short`	Index in `'STR#' 130` resource of first undo label

The following data members are protected:

Data member	Type	Description
`lineWidth`	`short`	Width of a text line in pixels. If negative, lines are as wide as the pane.
`fixedLineHeights`	`Boolean`	`TRUE` if all lines are the same height.
`wholeLines`	`Boolean`	`TRUE` if lines aren't cut off vertically.
`lastFontNum`	`short`	Last font number.
`lastFontCmd`	`long`	Last font-command number.
`lastTextSize`	`short`	Last seen text size.
`lastSizeCmd`	`long`	Last size-command number.
`editable`	`Boolean`	`TRUE` if user can edit text.
`stylable`	`Boolean`	`TRUE` if user can change font, size, or style of text.
`scrollHoriz`	`Boolean`	`TRUE` if automatic horizontal scrolling is enabled.

Member Functions

Creation and destruction

CAbstractText

```
CAbstractText(CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth, Boolean aScrollHoriz);
```

Constructor. Initializes an abstract text pane.

Most of the arguments to the constructor are identical to those of CPane's constructor. `aLineWidth` specifies how wide the lines are to be. If a line width is less than 0, the text wraps to the width of the pane. Supplying the value MAXINT disables word wrap. The default value is -1. `aScrollHoriz` specifies whether the text should scroll horizontally. TRUE enables horizontal scrolling; FALSE disables it. Setting `aScrollHoriz` to TRUE is meaningful only if aLineWidth is positive and is greater than the width of the pane. The default value is FALSE.

Note

The descriptions of the other arguments are in CPane.

CAbstractText

```
CAbstractText ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. It can also be used in combination with `IAbstractText` for backward compatibility.

IAbstractText

```
void IAbstractText (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initializes an abstract text pane. If no constructor arguments are specified, the pane must be further initialized by calling `IAbstractText`. If any constructor arguments are specified, `IAbstractText` must not be called. See constructor for descriptions of the arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   Ptr viewData );
```

Initializes an abstract text object from a view template. The data members are set using the view template pointed to by `viewData`.

Accessing

Specify

```
void Specify (Boolean fEditable,
   Boolean fSelectable, Boolean fStylable);
```

Specifies whether the user can edit, select, and style text. This member function sets `editable` to `fEditable`, `wantsClicks` to `fSelectable`, and `stylable` to `fStylable`. By default, text is editable, selectable, and stylable. If editing is allowed, the user can insert and delete text. If selecting is allowed, the user can select text for copying to the Clipboard. If styling is allowed, the font, style, and size of the text can be changed.

To make your `Specify` calls more understandable, the THINK Class Library defines constants that you can use instead of `TRUE` and `FALSE`:

```
kEditable           kNotEditable
kSelectable         kNotSelectable
kStylable           kNotStylable
```

This list describes some common ways to call `Specify`:

- To let the user edit and style text, call `Specify(kEditable, kSelectable, kStylable)`.
- To prevent the user from editing, styling, or selecting text, call `Specify(kNotEditable, kNotSelectable, kNotStylable)`.
- To let the user select text (to copy it, for example) but not edit or style it, call `Specify(kNotEditable, kSelectable, kNotStylable)`.
- To let the user select and edit text, but not style it, call `Specify(kEditable, kSelectable, kNotStylable)`.
- To let the user select and style text, but not edit it, call `Specify(kNotEditable, kSelectable, kStylable)`.

Note

If `fSelectable` is `kNotSelectable`, `Specify` disables editing and styling, regardless of the values of `fEditable` and `fStylable`.

You can call `Specify` at any time. For example, you can lock text (that is, disable editing) to prevent accidental editing, then let the user unlock it later.

GetSpecification

```
void GetSpecification (Boolean *fEditable,
   Boolean *fSelectable, Boolean *fStylable);
```

Returns whether the text in this pane is editable, stylable, or selectable.

Command

DoCommand

```
void DoCommand (long theCommand);
```

Handles the common commands that apply to text such as cutting, copying, pasting, font selection, styling, alignment, and spacing. `DoCommand` handles the commands listed below, passing all other commands to its supervisor.

- Editing commands: `cmdCut`, `cmdCopy`, `cmdPaste`, `cmdClear`, `cmdSelectAll`
- Style commands: `cmdPlain`, `cmdBold`, `cmdItalic`, `cmdUnderline`, `cmdOutline`, `cmdShadow`, `cmdCondense`, `cmdExtend`
- Alignment commands: `cmdAlignLeft`, `cmdAlignCenter`, `cmdAlignRight`
- Spacing commands: `cmdSingleSpace`, `cmd1HalfSpace`, `cmdDoubleSpace`
- Font selection from the **Font** menu (`MENUfont`)
- Size selection from the **Size** menu (`MENUsize`)

To let the user undo actions, this member function creates a task object for most commands. It creates a CTextEditTask object for an editing command and a CTextStyleTask object for a style, alignment, spacing, font, or size command.

UpdateMenus

```
void UpdateMenus();
```

Updates the menus items that pertain to text processing just before the menu appears. This function enables the **Cut**, **Copy**, and **Clear** commands if there is a current selection, and enables the **Paste** command if there is text in the Clipboard. This function also checks the current font, size, style, alignment, and spacing commands.

Note

See the implementation of this member function for an example of how to write your own `UpdateMenus` function.

DoKeyDown

```
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handles a key down in a text pane. Usually, `DoKeyDown` passes function keys and command keys to its base class and inserts other characters into the text. This table shows how `DoKeyDown` handles command keys and function keys:

If the key is…	This member function…
Any command key	Passes it to the base class
Home, Page Up, Page Down	Passes it to the base class
End	Scrolls to the bottom left of the text
A cursor key	Calls `TypeChar` and notifies the editing task of the change in the selection
Any other key	Passes it to the editing task, which should insert the character if the text pane is editable

Table 14-1 How DoKeyDown handles keys

DoAutoKey

```
void DoAutoKey (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handles an auto key in a text pane. If the command key is down, this function does nothing. Otherwise, this function calls `DoKeyDown`.

PerformEditCommand

```
void PerformEditCommand (long theCommand);
```

Performs the standard **Cut**, **Copy**, **Paste**, and **Clear** commands on the text. The task classes call this function to perform and undo an edit command. The default function does nothing. Your derived class must override this function.

TypeChar

```
void TypeChar (char theChar, short theModifiers);
```

Processes a keystroke. This function does not need to set up for an **Undo** command and should handle the key directly. The task classes call this function to perform a keystroke. The default function does nothing. Your derived class must override this function.

SelectionChanged

```
void SelectionChanged ();
```

Called when the selection has just changed. This function calls the `BroadcastChange` function with `textSelectionChanged` as the reason. If this text pane has an editing task, it calls the `SelectionChanged` function.

MakeEditTask

```
CTextEditTask *MakeEditTask (long editCmd);
```

Creates a task to handle typing and the **Cut**, **Copy**, **Paste**, and **Clear** commands. This function creates a CTextEditTask object. If you override this function, your function must create a CTextEditTask-derived object. `MakeEditTask` is a protected member function.

MakeStyleTask

```
CTextStyleTask *MakeStyleTask (long editCmd);
```

Creates a task to handle font, style, size, alignment, and spacing commands. This function creates a CTextStyleTask object. If you override this function, your function must create a CTextStyleTask-derived task object. `MakeStyleTask` is a protected member function.

BecomeGopher

```
Boolean BecomeGopher (Boolean fBecoming);
```

Returns `TRUE` if requested action completed without error. If `fBecoming` is `TRUE`, this text pane is becoming the gopher. This function activates it and enables the **Font**, **Size**, and **Style** menus. If `fBecoming` is `FALSE`, this text pane is no longer the gopher. This function deactivates the pane and disables the **Font**, **Size**, and **Style** menus.

Display

ScrollToSelection

```
void ScrollToSelection ();
```

Scrolls so that the current selection is visible.

SetHorizontalScroll

```
void SetHorizontalScroll(Boolean doHoriz);
```

Enables automatic horizontal scrolling if `doHoriz` is `TRUE`; disables it if `doHoriz` is `FALSE`. By default, it's disabled. If automatic horizontal scrolling is enabled, the insertion point or selection is automatically scrolled into view if it moves beyond the right or left margin of the pane.

SetSelection

```
void SetSelection (long selStart, long selEnd,
   Boolean fRedraw);
```

Sets the selection to the range corresponding to character positions `selStart` through `selEnd`. The default function does nothing. Your derived class must override this function.

GetSelection

```
void GetSelection (long *selStart, long *selEnd);
```

Returns the start and end of the current selection. The default member function does nothing. Your derived class must override this function.

SelectAll

```
void SelectAll (Boolean fRedraw);
```

Selects all the text in this text pane. This function uses `GetLength` to determine the length of the text.

HideSelection

```
void HideSelection(Boolean hide, Boolean redraw);
```

Hides the current selection or insertion point if `hide` is `TRUE`. Otherwise, the current selection or insertion point is visible. The default function does nothing. Your derived class must override this function.

Paginate

```
void Paginate (struct CPrinter *aPrinter,
   short pageWidth, short pageHeight);
```

Splits the text pane into pages. This function ensures that lines aren't cut in half horizontally at the end of a page.

Text specification

SetTextString

```
void SetTextString (Str255 textStr);
```

Uses the specified string as the text pane's text. This function uses `SetTextPtr` to set the text.

SetTextHandle

```
void SetTextHandle (Handle textHand);
```

Set the text for this pane to the contents of `textHand`.. This function uses `SetTextPtr` to set the text.

SetTextPtr

```
void SetTextPtr (Ptr textPtr, long numChars);
```

Uses the first `numChars` characters that `textPtr` points to as the text for this abstract text object. The default function does nothing. Your derived class must override this function, which must make a copy of the text.

GetTextHandle

```
Handle GetTextHandle ();
```

Returns a handle to the text. The default function does nothing. Your derived class must override this function. In most cases, the text class should keep its text in a handle, and this function should return that handle—not a copy of it.

If your implementation requires the function to return a copy of the text in a handle, your derived class should override any functions that call `GetTextHandle` to dispose of the handle. In CAbstractText, the member functions `GetCharBefore` and `GetCharAfter` expect `GetTextHandle` to return a handle to the actual text.

CopyTextRange

```
Handle CopyTextRange (long start, long end);
```

Returns a copy of the text specified by `start` and `end`. The default member function does nothing. Your derived class must override this member function.

InsertTextPtr

```
void InsertTextPtr (Ptr text, long length,
   Boolean fRedraw);
```

Inserts a copy of the given text at the start of the selection. `text` is a pointer to the text. The default member function does nothing. Your derived class must override this member function.

InsertTextHandle

```
void InsertTextHandle (Handle text,
   Boolean fRedraw);
```

Inserts a copy of the given text at the start of the selection. `text` is a handle to the text.

GetCharBefore

```
void GetCharBefore (long *aPosition,
   tCharBuf charBuf);
```

Returns the character before `aPosition`. The character and its size are returned in `charBuf`, and `aPosition` is updated with the starting position of the character. If there is no character before `aPosition`, then this member function sets the length of the character to 0 and does not update `aPosition`.

GetCharAfter

```
void GetCharAfter (long *aPosition,
   tCharBuf charBuf);
```

Returns the character after `aPosition`. The character and its size are returned in `charBuf`, and `aPosition` is updated with the starting position of the character. If there is no character after `aPosition`, then this member function sets the length of the character to 0 and does not update `aPosition`.

Text characteristics

SetFontNumber

```
void SetFontNumber (short aFontNumber);
```

Sets the font by font number. The default member function does nothing. Your derived class must override this member function. If your derived class supports multiple fonts in a pane, this function should set the font for the current selection. Otherwise, it should set the font for the whole text pane.

SetFontName

```
void SetFontName (Str255 aFontName);
```

Sets the font by font name.

SetFontStyle

```
void SetFontStyle (short aStyle);
```

Sets the font style. `aStyle` may be one of the following: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, or `extend`. The default member function does nothing. Your derived class must override this member function. If your derived class supports multiple styles in a pane, this member function should set the style for the current selection. Otherwise, it should set the style for the whole text pane.

SetFontSize

```
void SetFontSize (short aSize);
```

Sets the font size to the specified size. The default member function does nothing. Your derived class must override this member function. If your derived class supports multiple styles in a pane, this member function should set the size for the current selection. Otherwise, it should set the size for the whole text pane.

SetTextMode

```
void SetTextMode (short aMode);
```

Sets the text mode to the specified mode. `aMode` can be one of the following: `srcOr`, `srcXor`, or `srcBic`. The default member function does nothing. Your derived class must override this member function.

SetAlignCmd

```
void SetAlignCmd (long anAlignment);
```

Sets the text alignment. `anAlignment` is one of the following: `cmdAlignLeft`, `cmdAlignRight`, or `cmdAlignCenter`. The default member function does nothing. Your derived class must override this member function. If your derived class supports multiple alignments in a pane, this member function should set the alignment for the current selection only. Otherwise, it should set the alignment for the whole text pane.

GetAlignCmd

```
long GetAlignCmd ();
```

Returns the current alignment. It can be one of the following: `cmdAlignLeft`, `cmdAlignRight`, `cmdAlignCenter`, or `cmdNull`. The default member function does nothing. Your derived class must override this member function. If your derived class allows different alignments within a text pane and the current selection contains more than one alignment, this member function should return `cmdNull`.

SetSpacingCmd

```
void SetSpacingCmd (long aSpacingCmd);
```

Sets the space between lines of text. `aSpacingCmd` can be one of the following: `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`. The default member function does nothing. Your derived class must override this member function.

GetSpacingCmd

```
long GetSpacingCmd ();
```

Returns the space between lines of text. It can be one of the following: `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`. The default function does nothing. Your derived class must override this function.

GetHeight

```
long GetHeight (long startLine, long endLine);
```

Returns the total height in pixels of the indicated lines of text. The default function does nothing. Your derived class must override this function.

Get1Height

```
short Get1Height (long aLineNum);
```

Returns the height in pixels of the indicated line of text.

GetCharOffset

```
long GetCharOffset (longPt *aPt);
```

Returns the character position nearest the coordinate `aPt`. `aPt` must be in frame coordinates. The default function does nothing. Your derived class must override this function.

GetCharPoint

```
void GetCharPoint( long offset, longPt *aPt);
```

Returns the coordinates of the character position `offset` in `*aPt`. `aPt` is in frame coordinates. The default function does nothing. Your derived class must override this function.

GetCharStyle

```
void GetCharStyle (long charOffset,
   TextStyle *theStyle);
```

Returns style information in `*theStyle` for the character at position `charOffset`.

GetTextStyle

```
void GetTextStyle (short *whichAttributes,
   TextStyle *aStyle);
```

Returns current text style information. `whichAttributes` is a flag that indicates which text attributes to report. The default function does nothing. Your derived class must override this function. If your derived class supports multiple styles in a text pane, this member function should report only those attributes that are continuous over the current selection, and should set `whichAttributes` to those attributes.

The attributes flags and `TextStyle` record are described in *Inside Macintosh Volume VI*, Chapter 15, "TextEdit."

Calibrating

SetWholeLines

```
void SetWholeLines (Boolean aWholeLines);
```

If `aWholeLines` is `TRUE`, this function resizes the frame so that it displays only whole lines. The text pane is not redrawn.

GetWholeLines

```
Boolean GetWholeLines();
```

Returns the value of the `wholeLines` data member. If `TRUE`, the frame is set to display only whole lines and not partial lines.

ResizeFrame

```
void ResizeFrame (Rect *delta);
```

Resizes the frame when the size of its pane changes. The `delta` rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left.

FindLine

```
long FindLine (long charPos);
```

Returns the line number containing the specified character position. The default function does nothing. Your derived class must override this function.

Both line and character numbering start at `0`. If the character position is before the start of the text (a negative number), this member function should return `0`. If the character position is beyond the end of the text, this member function should return the number of the last line.

GetLength

```
long GetLength ();
```

Returns the length in bytes of the text buffer. The default function does nothing. Your derived class must override this function.

GetNumLines

```
long GetNumLines ();
```

Returns the total number of lines in the text buffer. The default function does nothing. Your derived class must override this function.

Cursor

AdjustCursor

```
void AdjustCursor (Point where,
   RgnHandle mouseRgn);
```

Turns the cursor into an I-beam when the mouse is in the edit text pane.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

CAbstractTextX

```
void CAbstractTextX ();
```

Performs common constructor initialization.

14 CAbstractText

CAppleEvent 15

Introduction

CAppleEvent is a class whose objects contain an Apple event and its reply. The basic THINK Class Library handles the four required Apple events (Open Application, Open Documents, Print Documents, and Quit Application). If you make use of the Apple-event-aware functionality of CApplication, CDocument, and so on, your application can be scriptable and recordable using the Core, Miscellaneous, Text, and other Apple-event suites defined by the Apple Event Registry.

Heritage

Base Class	None
Derived Classes	None

Using CAppleEvent

When your application receives an Apple event, the THINK Class Library puts it in a CAppleEvent object. If the event is one of the four required events, it is passed to the application by calling the application's `DoRequiredEvent` member function. If the event is not one of the four required events and your application takes advantage of the Apple-event-aware features of the THINK Class Library, the event is dispatched directly to the appropriate CAppleEventObject-derived object. You don't need to write any code to handle the four required Apple events or any of the events handled by the CApplication, CDocument, CSwitchboard, CStyleText, or CWindow classes.

If you want your application to handle an Apple event other than the four required events or if you want your application to be scriptable and recordable—you must base your application on the Apple-event-aware classes. You also must write a `DoAppleEvent` member

function to handle the event. That function should be in the class of the object that is the target or direct object of the event: your derived class of CApplication, CDocument, CPane, and so on. You don't need to derive a new class from CAppleEvent to handle new Apple events.

CAppleEvent provides a number of functions that allow you to extract the parameters from an Apple event or add parameters to the reply. After you extract the parameters you know about, call the CAppleEvent's `FailMoreRequiredParams` function, which fails with an `errAEParamMissed` error code if you have not extracted all the required parameters.

If your Apple event needs to interact with the user (for example, if it displays a dialog), call the application's `RequestInteraction` function, specifying whether the interaction is required or optional. `RequestInteraction` fails if a required interaction is not permitted; otherwise, it returns `TRUE` if the requested interaction can take place, and `FALSE` if it is not allowed.

Apple events succeed by default. If `Failure` is called during handling of an Apple event, the event fails, returning the contents of the exception's error code as its result code and the message indicated by the exception's error message code, if any, as its error string.

How the THINK Class Library handles Apple events

When you initialize your application, it checks to see if the Apple Event Manager is installed. If it is, the switchboard installs a number of handlers and object accessors from the CAppleEventObject class.

When the switchboard receives a high-level event, it assumes the event is an Apple event and calls the Apple Event Manager's `AEProcessAppleEvent` function. `AEProcessAppleEvent` calls the handler installed for the event directly.

`AEProcessAppleEvent` calls one of the CAppleEventObject handler functions: `GenericResultHandler`, `GenericNoResultHandler`, `GenericAppHandler`, or `GenericGopherHandler`. The first three functions resolve the direct object or insertion location parameter of the event to determine which object should handle the event; then they dispatch the event directly by calling the object's `DoAppleEvent` function. `GenericGopherHandler` dispatches the event to the application

or to the object that is currently the gopher. In every case, the object that ultimately handles the event must be an instance of a derived class of CAppleEventObject.

Data Members

The following data members are protected:

Data member	Type	Description
`theEvent`	`const AppleEvent*`	Pointer to the Apple event.
`theReply`	`AppleEvent*`	Pointer to the default reply.
`theRefCon`	`long`	The reference value.
`eventClass`	`DescType`	The event class; for example, `'aevt'`.
`eventID`	`DescType`	The event ID; for example, `'oapp'`.
`canInteract`	`Boolean`	`TRUE` if interaction with the user has been previously requested and approved.
`directIsToken`	`Boolean`	`TRUE` if direct object is token.
`notificationRec`	`struct NMRec*`	The Notification Manager record for `AEInteractWith-User`. By default, it's `NULL`.
`idleProc`	`AEIdleUPP`	The idle procedure for `AEInteract–WithUser`. If `NULL`, uses CSwitchboard's idle procedure.
`directObject`	`AEDesc`	The cached token value of the direct object parameter of the event.
`insertionLoc`	`AEDesc`	The cached token value of the insertion loc parameter of the event.

Data member	Type	Description
insertionPos	DescType	Insertion location position.
currentEvent	static CAppleEvent*	Pointer to the event currently being handled.
previousEvent	CAppleEvent*	Pointer to the previous event. Non-NULL if an event was being handled when the current event was received.

Member Functions

Creation and destruction

CAppleEvent

```
CAppleEvent (const AppleEvent
   *theEvent = NULL,
   AppleEvent *theReply = NULL,
   long theRefCon = 0);
```

Constructor. Initializes a CAppleEvent object. `theEvent` is the Apple event that CSwitchboard received. `theReply` is the default reply that the Apple Event Manager generates for an Apple event. `theRefCon` is the reference value for this event's class and ID as set by CSwitchboard. For the default handlers, `theRefCon` is always `0`.

CAppleEvent

```
CAppleEvent ();
```

Default constructor. Implicitly called when this object is created by `new_by_name`. Can also be used in combination with `IAppleEvent` for backward compatibility.

~CAppleEvent

```
~CAppleEvent ();
```

Destructor. Calls CAppleEventObject's `EndEvent` function to delete any AEObjects created as tokens during processing of this event. Disposes of the direct object and insertion loc descriptors.

IAppleEvent

```
void IAppleEvent (const AppleEvent *theEvent,
   AppleEvent *theReply, long theRefCon,
   DescType anEventClass, DescType anEventID);
```

Initialization function provides for backward compatibility. If no constructor arguments are specified, a new object must be further initialized by calling `IAppleEvent`. If any constructor arguments are specified, `IAppleEvent` must not be called. See the non-default constructor for descriptions of the arguments.

Dispose

```
void Dispose ();
```

For backward compatibility. Calls `delete this`.

Accessing

GetDirectObject

```
Boolean GetDirectObject (AEDesc *objToken);
```

Returns in `objToken` the resolved token value of the direct object parameter of the event. Fails if there is no direct object parameter.

The direct object is cached by the event and should not be disposed by the caller. Subsequent calls to `GetDirectObject` return the same value with no further overhead.

The token is a descriptor whose descriptor type is the object class of the token and whose data handle contains a pointer to the `AEObject` that handles the event.

The token may also be a `typeAEList` descriptor, in which case it is an `AEDescList`. The Apple Event Manager's Object Support Library automatically forms a list of tokens as the result of a query with a test; certain `AEObject` accessors may also return lists of tokens. Lists of tokens are preprocessed by the `GenericResultHandler` and `GenericNoResultHandler` functions, which break down events directed to multiple objects into multiple events, each directed to a single object. `GenericResultHandler` also collects results from such events into result lists. In most cases, your application objects do not need to deal with token lists.

GetInsertionLoc

```
void GetInsertionLoc (Boolean locRequired,
   AEDesc *locToken, DescType *theClass);
```

Returns in `locToken` the resolved token value of the insertion loc parameter of the event, and `*theClass` to the element class of the insertion. Fails if there are no insertion loc or element class parameters. To handle events in which the insertion loc is not a required parameter, use `GetOptionalParamDesc` to obtain a descriptor of it.

The insertion loc is cached by the event and should not be disposed of by the caller. Subsequent calls to `GetInsertionLoc` return the same value with no further overhead.

DescListToArray

```
CArray *DescListToArray(const AEDesc *descList,
   DescType itemType, Size itemSize);
```

Converts a descriptor of `typeAEList` to a CArray whose contents are the items of the list. The items must be of uniform type and size. If the list has no items it returns an empty array. It fails if any error is encountered.

FailMoreRequiredParams

```
void FailMoreRequiredParams();
```

Fails if the event contains required parameters that have not yet been read by the handler.

InteractionPermitted

```
Boolean InteractionPermitted(Boolean required);
```

`required` is `TRUE` if the operation cannot be performed without user interaction (for example, in a dialog requesting the user to specify a file name), or `FALSE` if the interaction is optional (for example, in a dialog requesting the user to confirm a save). Returns `TRUE` for any interaction if the event specifies `kAEAlwaysInteract`, `TRUE` for required interactions if the event specifies `kAECanInteract`, and `FALSE` otherwise.

Objects should not call this function directly, but should call the application's `RequestInteraction` function instead.

GetCurrentEvent

```
static CAppleEvent *GetCurrentEvent();
```

Returns a pointer to the `AppleEvent` currently being handled.

GetOptionalParamDesc

```
Boolean GetOptionalParamDesc(
   AEKeyword theKeyword, DescType desiredType,
   AEDesc *result);
```

Returns in `result` a descriptor of an optional event parameter. `theKeyword` is the parameter keyword. `desiredType` is the parameter type; specify `typeAEWildCard` if the parameter can have more than one type. The function returns `TRUE` if the requested parameter is present in the event, and `FALSE` if not.

The function fails if any error other than `errAEDescNotFound` is returned by the Apple Event Manager.

GetOptionalParamPtr

```
Boolean GetOptionalParamPtr(
   AEKeyword theKeyword, DescType desiredType,
   Size requestedSize, void *data);
```

Returns in `data` the value of an optional event parameter. `theKeyword` is the parameter keyword. `desiredType` is the parameter type; specify `typeAEWildCard` if the parameter can have more than one type. `requestedSize` is the expected size of the value. The function returns `TRUE` if the requested parameter is present in the event, and `FALSE` if not.

The function fails if the size of the parameter data is not the requested size or if any error other than `errAEDescNotFound` is returned by the Apple Event Manager.

GetRequiredParamDesc

```
void GetRequiredParamDesc(
   AEKeyword theKeyword, DescType desiredType,
   AEDesc *result);
```

Returns in `result` a descriptor of a required event parameter. `theKeyword` is the parameter keyword. `desiredType` is the parameter type; specify `typeAEWildCard` if the parameter can have more than one type. The function returns `TRUE` if the requested parameter is present in the event, and `FALSE` if not.

The function fails if any error other than `errAEDescNotFound` is returned by the Apple Event Manager.

GetRequiredParamPtr

```
void GetRequiredParamPtr(
   AEKeyword theKeyword, DescType desiredType,
   Size requestedSize, void *data);
```

Returns in `data` the value of a required event parameter. `theKeyword` is the parameter keyword. `desiredType` is the parameter type; specify `typeAEWildCard` if the parameter can have more than one type. `requestedSize` is the expected size of the value.

The function fails if the size of the parameter data is not the requested size or if any error other than `errAEDescNotFound` is returned by the Apple Event Manager.

GetAttributeDesc

```
void GetAttributeDesc(AEKeyword theKeyword,
   DescType desiredType, AEDesc *result);
```

Returns in `result` a descriptor of an event attribute. `theKeyword` is the attribute keyword. `desiredType` is the attribute type.

The function fails if the attribute is not found or cannot be coerced to the requested type.

GetAttributePtr

```
void GetAttributePtr(AEKeyword theKeyword,
   DescType desiredType, Size requestedSize,
   void *data);
```

Returns in `data` the value of an event attribute. `theKeyword` is the attribute keyword. `desiredType` is the attribute type. `requestedSize` is the expected size of the returned data.

The function fails if the attribute data is not of the requested size, is not found, or cannot be coerced to the requested type.

GetEventClass

```
DescType GetEventClass ();
```

Returns the event class, for example `'aevt'`. The event class and event ID uniquely identify an `AppleEvent`.

GetEventID

```
DescType GetEventID ();
```

Returns the event ID, for example `'oapp'`. The event class and event ID uniquely identify an Apple event.

GetEvent

```
const AppleEvent *GetEvent ();
```

Returns a pointer to the Apple event.

GetReply

```
AppleEvent *GetReply ();
```

Returns a pointer to the Apple event reply.

GetRefCon

```
long GetRefCon ();
```

Returns the Apple event reference value for this event's class and ID. By default, this is always `0` in the THINK Class Library. To use other values, override the `ISwitchboard` member function in CSwitchboard, and change its call to `AEInstallEventHandler`.

GetDescList

```
void GetDescList (AEKeyword whichParam,
   AEDescList *descList);
```

Returns in `descList` the `AEDescList` associated with the keyword `whichParam`.

ExtractFromDescList

```
CArray *ExtractFromDescList (AEKeyword
   whichParam, DescType itemType, Size itemSize);
```

Gets the descriptor list associated with the parameter `whichParam`, and puts its items into a CArray. This member function is especially useful if all the items in the list are the same size and the data in the list is not keyword-separated. This member function does not use keywords.

GotRequiredParams

```
Boolean GotRequiredParams ();
```

Performs the recommended check to determine if all the required parameters of an Apple event have already been extracted. You should usually call `FailMoreRequiredParams` instead.

RequestInteraction

```
OSErr RequestInteraction (long timeOutTicks);
```

Called to indicate that you need to interact with the user (for example, to put up a dialog) to handle the event. If interaction is allowed and the application is brought forward before the timeout period has elapsed, then this member function returns `noErr` and you can proceed with the interaction. If any other value is returned, then your attempt to interact failed.

This member function calls `AEInteractWithUser`, with `timeOutTicks`, an idle procedure, and a pointer to a Notification Manager record. By default, the idle procedure calls the switchboard's `AppleEventIdle` function(see Chapter 112, "CSwitchboard"). To use a different idle procedure, assign it to the `idleproc` instance data member. Also, the Notification Manager record pointer is `NULL` by default, so `AEInteractWithUser` supplies its own default record. To use your own record, assign it to the `notificationRec` instance data member.

Objects should not call this function directly, but should call `CApplication::RequestInteraction` instead.

Member Functions: Private

CAppleEventX

```
void CAppleEventX();
```

Performs common initialization.

CAppleEventObject 16

Introduction

CAppleEventObject implements the behavior required of the Apple event Object Model cObject class. See the *Apple Event Registry* for an explanation of the Object Model and the cObject class.

Heritage

Base Class	None
Derived Classes	CApplication
	CClipboard
	CDocument
	CProperty
	CWindow
	CFileElement
	Your own Object Model classes

Using CAppleEventObject

CAppleEventObject can be used as a mixin or as a base class. Usually, CAppleEventObject is used as a mixin for classes that directly correspond to classes in the Object Model, as CApplication corresponds to cApplication or CDocument to cDocument. CAppleEventObject is used as the base class for classes that would not exist in the TCL were it not for Apple events, such as CProperty.

CAppleEventObject contains pure virtual functions for the basic operations of an Apple event handler and object accessor. See *Inside Macintosh: Interapplication Communication* for a complete description of these operations.

Event handlers

Each CAppleEventObject object acts as a handler for events directed to that object. An event is directed to an object if the object is specified as the direct object or insertion loc parameter of the event,

or if the object is implied by the event. For example: your current selection is implied by the Cut event from the Miscellaneous suite.

For most events, the handler that the Apple Event Manager is one of the generic handler functions defined by CAppleEventObject. The generic handler functions resolve object specifiers and identify the CAppleEventObject class objects that ultimately handle the event. They also gather results into a result list. To act as an event handler, a CAppleEventObject derived class must override the following pure virtual member functions:

Member function	Override needs to…
`DoAppleEvent`	Handles the event and, if necessary, return a result. Usually, `DoAppleEvent` calls a specific member function for each event, for instance, `HandleGetData`.
`GetContainer`	Returns a pointer to the CAppleEventObject class object that contains this object.
`GetClassID`	Returns the `AEDescType` for the class of the Apple event object, for example, cDocument.
`GetDefaultType`	Returns the `AEDescType` for the default type of data returned by a Get Data event directed to the object.
`MakeSelfDescriptor`	Returns an Apple event descriptor and key that describe this object.

Every object returns some sort of data. Objects such as documents simply return an object specifier that can be used to target the document; objects such as text paragraphs return the text. Therefore, most CAppleEventObject derived classes will override `DoGetDataEvent`.

Almost all Apple event objects have properties and will need to override the following functions to allow accessing and changing property data:

Member function	Override needs to…
`DoPropertyGetData`	Return the data for one of the object's properties.
`DoPropertySetData`	Change the data for one of the object's properties.

CAppleEventObject defines a number of other Do functions, described in the section "Event handling" later in this chapter. Your class should override each function corresponding to an event that objects of that class handle. Events for which no Do function is predefined can be handled by adding a case to your `DoAppleEvent` function.

Object accessors

Events that conform to the Apple event Object Model use object specifiers to describe application objects. Object specifiers are internal forms that might appear, for example, in AppleScript as: `title of first document of application "MyApp"`.

To interpret the event correctly, your application must convert each object specifier into a pointer to the corresponding application object. If there is no corresponding application object, as would be the case for the specifier above, the specifier must be converted into a pointer to an object that represents the Apple event object while the event is being processed.

The Apple Event Manager provides the Object Support Library (OSL) to assist with interpreting object specifiers. When an event processes an object specifier, the event handler calls the OSL's `AEResolve` function to convert the object specifier into a token. A token is an Apple event data descriptor whose `descriptorType` is the class of the Apple event object, and whose `dataHandle` is a handle of a four-byte location on the heap that contains a pointer to the CAppleEventObject corresponding to the specified Apple event object. In other words, a token descriptor is an indirect pointer to a CAppleEventObject, packaged with type information.

Beginning with the application object, the OSL works its way from the outermost container in the specifier to the innermost element or property (in the text form of the specifier, this would be from right to

left). It uses the `AccessObject` function to ask the CAppleEventObject container at each level of the specifier to return a token representing the next most specific object, until the specifier is fully resolved. The specifier `title of the first document of application "MyApp"` OSL would first ask the CApplication object to return a token designating the "first document" object. Then the OSL would ask the CDocument object pointed to by that token to return a token designating the title of the document. This second token would point to a CProperty object. This completely resolves the object specifier.

As each new token is returned, the OSL automatically frees the memory occupied by the previous token's Apple event descriptor. This releases the `dataHandle` memory, but has no effect on the CAppleEventObject class object pointed to indirectly by the `dataHandle`. All temporary Apple event objects created during the course of an event are automatically deleted when the event is finished.

As far as the OSL is concerned, a specifier is completely resolved once `AEResolve` returns a token representing the most specific object or objects specified. The generic handler that called `AEResolve` then uses that token to hand the event over to the objects for processing.

For the events in the Core and Miscellaneous suites, the CAppleEventObject class contains generic handlers that perform most of the actions described above. To act as an object accessor, a CAppleEventObject derived class needs to override only the following member functions:

Member function	Override needs to…
`AccessObject`	Returns a token designating an element or property of the object
`CountObjects`	Returns the number of elements of a particular type in the container
`CompareObjects`	Returns the result of comparing the CAppleEventObject class object with another object or constant data

The `AccessObject`, `CountObjects`, and `CompareObjects` member functions work the same as the `MyObjectAccessor`, `MyCountObjects`, and `MyCompareObjects` callback functions described in *Inside Macintosh: Interapplication Communication*.

Data Members

Static data members

CAppleEventObject defines the following public static data member:

Data member	Type	Description
`nullDesc`	`AEDesc`	Null descriptor

Any class that needs a null descriptor value can use `nullDesc`.

Protected data members

The following data members are protected:

Data member	Type	Description
`lastElementID`	`static long`	Used for generating unique IDs
`head`	`static CAppleEvent Object*`	Head of list of CAppleEventObjects (The Type must be entered with no breaks)
`callbackFlags`	`static short`	Flags used to call AEResolve
`stack`	`static CAppleEvent Object*[]`	Stack of CAppleEventObjects used by each active Apple event (The Type must be entered with no breaks)
`stackTop`	`static short`	Top of event stack

The stack depth, set by the `TCL_EVENTSTACKTOP` preprocessor variable defined in `CAppleEventObject.h`, determines the number of events that can be active at once. The default value is 3.

Object data members

The following data members are protected:

Data member	Type	Description
`elementID`	`long`	The object's unique ID
`next`	`CAppleEventObject*`	Next in the list of CAppleEventObjects used by active events

Data member	Type	Description
prev	CAppleEventObject*	Previous in the list of CAppleEventObjects
disposable	Boolean	TRUE if the object is automatically deleted at the end of the event that created it

Symbols

CAppleEventObject.h defines a preprocessor variable for a property not in the Apple event registry:

Symbol	Value	Description
pPosition	'ppos'	The upper-left corner of the object's *pBounds* rectangle

Member Functions

Creation and destruction

CAppleEventObject

```
CAppleEventObject (Boolean isDisposable);
```

Constructor. If isDisposable is TRUE, the CAppleEventObject is deleted when the currently active event terminates. isDisposable should be set to FALSE when CAppleEventObject is a mixin to existing application classes, and to TRUE for classes that are used only during an active event.

~CAppleEventObject

```
~CAppleEventObject ();
```

Destructor. Removes object from the list of objects associated with the currently active event.

Token memory management

DeleteObject

```
void DeleteObject ();
```

Deletes the Apple event object if it is disposable. During object specifier resolution, some event objects are created dynamically to serve as tokens. These objects are automatically deleted when the event is completely processed, as described in EndEvent later in

this chapter. Usually, only a small amount of memory is occupied by tokens that might be deleted earlier than the end of the event. However, your program can delete Apple event objects sooner, by calling `DeleteObject`. You should not call `operator delete` directly for an Apple event object used as a token, because it may be needed after the event is complete—as are, for instance, the CApplication and CDocument objects.

Event handling

DoAppleEvent

```
void DoAppleEvent (CAppleEvent *theEvent,
   AEDesc *result);
```

Processes an Apple event. Returns the result of the event, if any, in the `result` descriptor. Usually, an object's `DoAppleEvent` function consists of switch statements that identify the event by class and ID, then call another member function (for example, `DoGetDataEvent` for the Get Data event) to actually do the work. `DoAppleEvent` is a pure virtual function and must be overridden.

Note

In previous versions of the THINK Class Library, the only argument of `DoAppleEvent` was `theEvent`. The additional `result` argument is needed to efficiently handle events that address multiple objects and return multiple results, as explained below.

Overriding implementations of `DoAppleEvent` usually pass back the result to be returned as the event's reply in the `result` descriptor, rather than stuffing it directly into the `reply` Apple event descriptor within `theEvent`. This lets the `GenericResultHandler` collect results from multi-object events into a list. Otherwise, each application object would have to loop through token lists for any event that might specify more than one direct object.

Your `DoAppleEvent` override function should deal with each event it recognizes and call `CAppleEventObject::DoAppleEvent` (or the base class's function, if your class is not directly derived from CAppleEventObject) to handle events that can be done in a general-purpose way. If neither your override nor CAppleEventObject's

DoAppleEvent member function handles an event, the event fails with an errAEEventNotHandled error code.

CAppleEventObject's DoAppleEvent member function contains cases for events that can be handled in the same way by all objects, such as the CountElements and GetDataSize events. If you do not like CAppleEventObject's way of dealing with an event, add a case for it in your own DoAppleEvent function. CAppleEventObject also contains code to call specific "Do" functions for events that many objects recognize. This saves duplicating the dispatching code in each derived class. These functions are described below.

DoCloneEvent

```
void DoCloneEvent (CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Clone event from the Core suite. The default function fails. If your derived class handles the Clone event, you should override this function.

DoCountElementsEvent

```
void DoCountElementsEvent (
   CAppleEvent *theEvent, AEDesc *result);
```

Handles the Count Elements event from the Core suite. The default function calls the CountObjects member function of the event's direct object.

DoCreateElementEvent

```
void DoCreateElementEvent (
   CAppleEvent *theEvent, AEDesc *result);
```

Handles the Create Element event from the Core suite. The default function fails. If your derived class handles the Create Element event, you should override this function.

DoDeleteEvent

```
void DoDeleteEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Delete event from the Core suite. The default function fails. If your derived class handles the Delete event, you should override this function.

DoGetDataEvent

```
void DoGetDataEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Get Data event from the Core suite. The default function calls `MakeSelfSpecifier`. If your derived class handles the Get Data event, you should override this function.

DoGetDataSizeEvent

```
void DoGetDataSizeEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Get Data Size event from the Core suite. The default function calls `DoGetDataEvent`, gets the size of the returned data, then disposes the data.

DoMoveEvent

```
void DoMoveEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Move event from the Core suite. The default function fails. If your derived class handles the Move event, you should override this function.

DoSetDataEvent

```
void DoSetDataEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Set Data event from the Core suite. The default function fails. If your derived class handles the Set Data event, you should override this function.

DoCutEvent

```
void DoCutEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Cut event from the Miscellaneous suite. The default function fails. If your derived class handles the Cut event, you should override this function.

DoCopyEvent

```
void DoCopyEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Copy event from the Miscellaneous suite. The default function fails. If your derived class handles the Copy event, you should override this function.

DoPasteEvent

```
void DoPasteEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Paste event from the Miscellaneous suite. The default function fails. If your derived class handles the Paste event, you should override this function.

DoIsUniformEvent

```
void DoIsUniformEvent(CAppleEvent *theEvent,
   AEDesc *result);
```

Handles the Is Uniform event from the Miscellaneous suite. The default function fails. If your derived class handles the Is Uniform event, you should override this function.

DoPropertyGetDataEvent

```
void DoPropertyGetDataEvent(DescType property,
   const AEDesc *requestedType,
   AEDesc *data);
```

This function acts on a Get Data event for a property of the object and returns the value of the property in the `data` argument. Any object class that has properties must override this function. `requestedType` is a descriptor or a `typeAEList` list of descriptors containing the desired type of the returned data.

When an object specifier for a property is resolved in your `AccessObject` function, the usual technique is to allocate a CProperty object that identifies the property and contains a pointer to the object containing the property. Then, for a Get Data event, the property-containing object's `DoPropertyGetDataEvent` function is called by the CProperty object. Set Data events are handled similarly. Since most events directed to properties are Get Data or Set Data, this approach handles most events without requiring that a separate CAppleEventObject derived class be defined for each type of property.

The default member function handles the `pBestType`, `pDefaultType`, `pName`, `pIndex`, `pID`, and `pClass` properties, and fails for any other properties. Your `DoPropertyGetDataEvent` function should always call its base class's `DoPropertyGetDataEvent` function for any properties it does not recognize.

DoPropertySetDataEvent

```
void DoPropertySetDataEvent(DescType property,
   const AEDesc *data);
```

This function acts on a Set Data event for a property of the object and stores the contents of the `data` argument into the property. Any object class with properties that can be set must override this function. See `DoPropertyGetDataEvent` above for a discussion of how property events are handled.

The default member function fails. Your `DoPropertySetDataEvent` function should call its base class's `DoPropertySetDataEvent` function for any properties it does not recognize.

BecomeSelection

```
void BecomeSelection ();
```

When an object's `pSelection` property receives a Set Data event, it should resolve the object specifier data to a token and call the token object's `BecomeSelection` function.

Object information

GetSelection

```
CAppleEventObject *GetSelection ();
```

This member function returns a pointer to the Apple event object corresponding to the current selection relative to the called object.

The `gGopher` global pointer plays no role in identifying the current selection; the pointer is incorrect if the application is in the background. In any case, there is no guarantee that the class of the object pointed to by `gGopher` is derived from CAppleEventObject—for example, CDialogText is not. Each open window or document may have its own current selection, which corresponds to the gopher only for the active window.

The default function returns `NULL`. Classes that can contain the current selection must override this function. A returned `NULL` value indicates that the object has no current selection or that the selection is incapable of responding to an event.

CAppleEventObject's `GenericGopherHandler` function, which resolves gopher-specific events such as Cut or Paste, calls the application's `GetSelection` member function. If the application

returns NULL, the event fails; otherwise, the handler calls the selection's DoAppleEvent function.

CApplication overrides GetSelection. If there are no open documents, it returns NULL; otherwise, it returns the result of calling the active document's GetSelection member function. (If the application is in the background, the active document is the document that would be active if the application were in the foreground.)

CDocument does not override GetSelection. Your document-derived class must override it, usually to return the result of calling GetSelection for the next-most-specific container. For example, for a text document, your document class would likely call the GetSelection member function of itsMainPane.

GetContainer

```
CAppleEventObject *GetContainer ();
```

Returns a pointer to the Apple event object that contains this object. This is a pure virtual function; all derived classes must override it.

GetClassID

```
DescType GetClassID ();
```

Returns the Apple event Object Model class of this object. This is a pure virtual function; all derived classes must override it.

GetDefaultType

```
DescType GetDefaultType ();
```

Returns the default type of data returned by a Get Data event directed to this object. This is a pure virtual function; all derived classes must override it.

GetBestType

```
DescType GetBestType ();
```

Returns the type of data returned by a Get Data event directed to this object that most faithfully represents the object. The default function returns the result of calling GetDefaultType.

GetElementID

```
long GetElementID ();
```

Every CAppleEventObject has a unique `long` value for any single run of the application. This value serves as the ID of an Apple event object.

GetElementIndex

```
long GetElementIndex ();
```

Returns the index of the element in its container.

GetElementName

```
void GetElementName (Str255 name);
```

Returns the element's name in `name`. The default function fails. Element classes must override this function; properties need not.

GetElementNameDesc

```
void GetElementNameDesc (AEDesc *nameDesc);
```

Returns in `nameDesc` the name of the element as a `typeChar` descriptor. The default function calls `GetElementName`.

Accessors

AccessObject

```
void AccessObject(DescType desiredClass,
   const AEDesc *containerToken,
   DescType containerClass, DescType keyForm,
   const AEDesc *keyData, AEDesc *theToken,
   long theRefcon);
```

`AccessObject` returns a token representing the requested property or element. The object that gets the `AccessObject` call is always pointed to by the `containerToken` argument. The arguments are the same as for the object accessor callback function described in *Inside Macintosh: Interapplication Communication*.

AccessSelection

```
void AccessSelection(DescType desiredClass,
   const AEDesc *containerToken,
   DescType containerClass, DescType keyForm,
   const AEDesc *keyData, AEDesc *theToken,
   long theRefcon);
```

This function calls `GetSelection` to obtain the current selection. If there is a current selection, its `AccessObject` function is called.

GetElementByName

```
void GetElementByName (DescType desiredClass,
   Str255 name, AEDesc *token);
```

Returns a `token` for the element in the container with the specified `name`. `desiredClass` tells what kind of element to locate. The default function loops through the container, calling `GetElementByIndex` and `GetElementName` until it finds the first element with the same name; it fails if it does not find such an element.

GetElementByIndex

```
void GetElementByIndex (DescType desiredClass,
   long index, AEDesc *token);
```

Returns a `token` for the element in the container with the specified `index`. `desiredClass` tells what kind of element to locate. The default function fails; container classes must override this function.

GetElementByID

```
void GetElementByID (DescType desiredClass,
   long ID, AEDesc *token);
```

Returns a `token` for the element in the container with the specified `ID`. `desiredClass` tells what kind of element to locate. The default function loops through the container, calling `GetElementByIndex` and `GetElementID` until it finds the element; it fails if the element is not found.

CountObjects

```
void CountObjects(DescType desiredClass,
   long *result);
```

`CountObjects` returns in `result` the number of elements of the `desiredClass` in the container whose `CountObjects` function is called. Every container class must override this function.

The Object Support Library calls this function while resolving object specifiers that contain tests.

CompareObjects

```
void CompareObjects(DescType comparisonOperator,
   const AEDesc *tokenOrLiteral,
   Boolean isToken, Boolean *result);
```

`CompareObjects` compares the value of the called object to that of the `tokenOrLiteral` argument; it returns `TRUE` if the values are equal, `FALSE` if they are not. If `isToken` is `TRUE`, `tokenOrLiteral` must be a token; otherwise, `tokenOrLiteral` must be literal data. Every element class must override this function.

`CompareObjects` is called by the Object Support Library while resolving object specifiers containing tests.

EqualObject

```
Boolean EqualObject(CAppleEventObject *otherPtr);
```

Compare `this` to another object pointer. The default function just compares the pointers. This function may be overridden so that tokens which are not identically equal can to be treated as if they are the same.

Marking

Marking functions are only called by the OSL if your application calls `Resolve` (or `AEResolve`) with a `kAEIDoMarking` callback flag. `Resolve` uses the value of the `callbackFlags` static data member, which is initially set to `kAEIDoMinimum` (no marking). If objects in your application use mark tokens, call `SetCallbackFlags` with a value of `kAEIDoMarking`.

When the OSL collects multiple tokens into a list, it stuffs them in a `typeAEList` descriptor. The list representing, for example, every word in a long document can be much larger than the original document. Mark tokens let the application choose its own representation for lists of tokens. The name mark tokens comes from the idea that the original application objects might be marked in some way to indicate they are members of a token list. However, any implementation will do. In particular, CRunArray class objects make good mark tokens for elements, such as text, that are easily identified by integer offsets.

GetMarkToken

```
void GetMarkToken (AEDesc *result);
```

Returns a mark token as the `result`.

If a particular object cannot mark, it simply returns `errAEEventNotHandled`; the OSL will then collect tokens from this container in a `typeAEList` descriptor, just as if marking were not enabled.The default `GetMarkToken` function returns `errAEEventNotHandled`.

MarkObject

```
void MarkObject(const AEDesc *theToken,
   long markCount);
```

A mark token's `MarkObject` function is called by the OSL to add `theToken` to the list represented by the mark token. The `markCount` is the index of the value in the mark token. Mark counts start at 1 and increase by 1 each time `MarkObject` is called for a particular mark token. The count need not be explicitly stored in the mark token.

AdjustMarks

```
void AdjustMarks(long newStart, long newStop);
```

When the OSL needs to reduce the number of elements in the list represented by the mark token, it calls `AdjustMarks`, specifying the range of elements that are to remain in the list; all others are discarded. `AdjustMarks` is called only once for a given mark token.

Creating object specifiers

MakeSelfSpecifier

```
void MakeSelfSpecifier (AEDesc *specifier);
```

This function calls `MakeSelfDescriptor` and the container's `MakeSelfSpecifier` to construct an object specifier representing the called object. Only the application class needs to override this function.

MakeSelfDescriptor

```
void MakeSelfDescriptor (DescType *keyForm,
   AEDesc *keyData);
```

`MakeSelfDescriptor` returns the `keyForm` and `keyData` uniquely identifying this object within its container. There are usually several ways to identify an object; the values returned should be chosen so the object can be located most quickly. For elements kept in CPtrArray lists by the container, `MakeSelfDescriptor` usually returns a `keyForm` of `formAbsolutePosition` and a `keyData`

descriptor of `typeLongInteger` containing the index of the object in the container's list.

This function is pure virtual; all derived classes must override it.

Tokens

MakeToken

```
void MakeToken (AEDesc *token);
```

Returns a `token` descriptor pointing to this object. The token's `descriptorType` is the value returned by `GetClassID`.

DisposeToken

```
static void DisposeToken (AEDesc *token);
```

Gets rid of a token and its associated storage. The `dataHandle` storage is released. If the token points to a disposable object, the object is deleted.

Note

It is dangerous to call `DisposeToken` if there can be more than one pointer to the object the argument token points to. For example, do not call `DisposeToken` on tokens packed into an `AEDescList`; if you do, when they are unpacked they will point to objects which have been deleted. Instead, call `AEDisposeDesc` or `TCLForgetDesc`. Other than to conserve intermediate storage, you never actually need to call `DisposeToken`; all disposable tokens created during an event are deleted when the event is completed.

TokenToPtr

```
static CAppleEventObject *TokenToPtr (
   const AEDesc *token);
```

Returns a pointer to the object the token points to. The token must be a `typeNull` application token or have been created by a call to `MakeToken`; it must not be a `typeAEList` token.

IsListToken

```
static Boolean IsListToken (const AEDesc *token);
```

Returns `TRUE` if the token has `typeAEList`.

IsMarkToken

```
static Boolean IsMarkToken (const AEDesc *token);
```

Returns TRUE if the token is a mark token. Returns FALSE if the token is a list or if the token object's IsMarkObject function returns FALSE.

IsMarkObject

```
Boolean IsMarkObject ();
```

Member function returns TRUE if the object is a mark token. The default function returns FALSE. Mark token objects must override.

Factoring

SendEventToThis

```
void SendEventToThis (DescType eventClass,
   DescType eventID, Boolean directObject,
   long sendMode);
```

Creates and sends a simple event to this application. eventClass and eventID identify the event. The value of sendMode is OR'ed with kAENoReply and kAEAlwaysInteract to yield the mode used to send the event. If directObject is TRUE, add a keyAEDirectObject specifying this object.

MakeEventToThis

```
void MakeEventToThis (DescType eventClass,
   DescType eventID, Boolean directObject,
   AppleEvent *event);
```

Creates an event to be sent to this application. eventClass and eventID identify the event. If directObject is TRUE, add a keyAEDirectObject specifying this object.

SendEventNoReply

```
void SendEventNoReply (
   const AppleEvent *event,
   long sendMode);
```

Sends the event with kAENormalPriority and kAEDefaultTimeout. sendMode is OR'ed with kAENoReply to yield the mode used to send the event.

SendCreateElementToThis

```
void SendCreateElementToThis (
   DescType elementClass);
```

Sends a Create Element event to this application specifying this object as the `keyInsertHere` parameter and the `elementClass` as the `keyAEObjectClass` parameter.

SendSetPropertyToThis

```
void SendSetPropertyToThis (
   DescType property,
   DescType typeCode, void *dataPtr,
   Size dataSize, long sendMode);
```

Sends a Set Data event to this object's specified property. `typeCode` is the type of data to be sent, `dataPtr` points to the data, and `dataSize` is its length in bytes. `sendMode` is treated as described in `SendEventToThis` earlier in this chapter.

Static utility functions

The utility functions are THINK Class Library versions of Apple Event Manager functions commonly used by handlers and accessors. Utility functions throw appropriate exceptions if an Apple Event Manager function returns a non-zero error code or if data of incorrect type is encountered.

GetProperty

```
static DescType GetProperty(
   const AEDesc *propDesc);
```

Returns the property ID (for example, `pName`) from the `propDesc` descriptor.

CreateDesc

```
static void CreateDesc(DescType typeCode,
   void *dataPtr, Size dataSize,
   AEDesc *result);
```

Creates a `result` descriptor with its `descriptorType` set to `typeCode` and its `dataHandle` set to the value of `dataPtr`, which is `dataSize` bytes long.

CreateLongDesc

```
static void CreateLongDesc (DescType typeCode,
   long data, AEDesc *result);
```

Creates a `result` descriptor with `typeCode` as its descriptor type and `long` (4-byte) data.

CreateBooleanDesc

```
static void CreateBooleanDesc (Boolean data,
   AEDesc *result);
```

Creates a `result` descriptor with `typeBoolean` descriptor type and `Boolean` (1-byte) data.

CoerceDescList

```
static Boolean CoerceDescList (
   const AEDesc *toTypes, AEDesc *desc);
```

Coerces the `desc` argument to one of the types in the list `toTypes`, which are in priority order. `toTypes` may also contain a single `typeType` descriptor. Returns `TRUE` if the resulting `desc` satisfies at least one of the types in the list.

Unlike `AECoerceDesc`, `CoerceDescList` does not always make a copy of the coerced descriptor. If `desc` already satisfies `toTypes`, it is returned unchanged. Otherwise, the original `desc` is disposed and a new `desc` of the desired type is created.

CoerceDesc

```
static void CoerceDesc (DescType toType,
   AEDesc *desc);
```

Coerces `desc` to the desired `toType`. Unlike `AECoerceDesc`, `CoerceDesc` does not always make a copy of the coerced descriptor. If `desc` is already of type `toType`, it is returned unchanged. Otherwise, the original `desc` is disposed and a new `desc` of the desired type is created. Throws an exception with an error code of `errAECoercionFailed` if coercion fails.

MayCoerceDesc

```
static OSErr MayCoerceDesc (DescType toType,
   AEDesc *desc);
```

Same as `CoerceDesc`, except `MayCoerceDesc` returns an error code instead of throwing an exception if coercion fails.

GetDescData

```
static void GetDescData (const AEDesc *desc,
   void *data, Size dataSize);
```

Extracts the `data` from the `desc` descriptor. The descriptor's `dataHandle` must hold exactly `dataSize` bytes; if not, an exception is thrown with an `errAECorruptData` error code.

GetDescString

```
static void GetDescString (const AEDesc *desc,
   unsigned char *str, Size maxSize);
```

Returns the value of the `desc` descriptor in `str as` a Pascal string of no more than `maxSize` bytes.

GetDescStr

```
static void GetDescStr (const AEDesc *desc,
   char *str, Size maxSize);
```

Returns the value of the `desc` descriptor in `str` as a C string of no more than `maxSize` bytes.

GetDescLong

```
static long GetDescLong (const AEDesc *desc);
```

Returns the `long` value of the `desc` descriptor. The `dataHandle` must be exactly 4 bytes long.

GetDescBoolean

```
static Boolean GetDescBoolean (
   const AEDesc *desc);
```

Returns the `Boolean` value of the `desc` descriptor. The `descriptorType` must be `typeBoolean`, `typeTrue`, or `typeFalse`.

GetDescRect

```
static void GetDescRect (const AEDesc *desc,
   Rect *rect);
```

Returns the `rect` value of the `desc` descriptor. The `descriptorType` must be coerceable to `typeQDRect`.

GetDescPoint

```
static void GetDescPoint (const AEDesc *desc,
   Point *pt);
```

Returns the `pt` value of the `desc` descriptor. The `descriptorType` must be coerceable to `typeQDPoint`.

CreateList

```
static void CreateList (AEDescList *list);
```

Returns an empty descriptor list.

AppendDesc

```
static void AppendDesc (AEDescList *list,
   const AEDesc *item);
```

Appends the `item` descriptor to the `list`.

CountItems

```
static void CountItems (const AEDescList *list,
   long *count);
```

Returns in `count` the number of elements in the `list`.

GetNthDesc

```
static void GetNthDesc (const AEDescList *list,
   long index, DescType type, AEDesc *result);
```

Returns the descriptor at the `index` location in the `list` as the value of `result`. If the descriptor cannot be coerced to `type`, `GetNthDesc` fails.

ApplyDesc

```
static void ApplyDesc (const AEDescList *list,
   TCLApplyDescFun fun, void *param);
```

Calls the function `fun` once for each element in the `list` with the element descriptor as its first argument and `param` as its second argument. The function is declared as:

```
typedef void (*TCLApplyDescFun)(
   const AEDesc *desc, void *param);
```

MapDesc

```
static void MapDesc (const AEDescList *list,
   TCLMapDescFun fun, void *param,
   AEDesc *result);
```

Calls the function `fun` once for each element in the `list` with the element descriptor as its first argument and `param` as its second argument. `fun` must return a result as its third argument for each element in the list. The results are collected into a list and returned by `MapDesc` as the value of `result`.

The function is declared as:

```
typedef void (*TCLMapDescFun)(
   const AEDesc *desc, void *param,
   AEDesc *result);
```

SumDesc

```
static void SumDesc (const AEDescList *list,
   TCLMapDescFun fun, void *param,
   AEDesc *result);
```

Calls the function `fun once` for each element in the `list` with the element descriptor as its first argument and `param` as its second argument. `fun` must return a `typeLongInteger` descriptor result as its third argument for each element in the list. The results are collected into a list and returned by `SumDesc` as the `typeLongInteger` descriptor value of `result`.

Generic handling

The generic handler functions are called directly by the Apple Event Manager based on the event ID and class, as set up in CSwitchboard by the `InstallEventHandlers` function.

The arguments to the generic handler functions are:

Argument	Description
`theEvent`	The incoming event.
`theReply`	An Apple event descriptor used to hold the result of the event or the error code and message, if any.
`refCon`	A long value associated with the event class and ID set by the application when it calls `AEInstallEventHandler`. The THINK Class Library sets all `refCons` to `0`.

All generic handlers package `theEvent`, `theReply` and `refCon` as a CAppleEvent object, and call some CAppleEventObject's `DoAppleEvent` function.

GenericAppHandler

```
static pascal OSErr GenericAppHandler (
   const AppleEvent *theEvent,
   AppleEvent *theReply, long refCon);
```

Calls `gApplication`'s `DoAppleEvent` function.

GenericResultHandler

```
static pascal OSErr GenericResultHandler (
   const AppleEvent *theEvent,
   AppleEvent *theReply, long refCon);
```

This function resolves the event's `keyDirectObject` parameter to obtain the target object, then calls its `DoAppleEvent` function directly. The result returned by `DoAppleEvent` is stored in the reply as the value of the reply's `keyAEResult` parameter.

If `keyDirectObject` specifies a list of objects, each object is sent the event in turn and the function calls `MapDesc` to collect the multiple results returned by `DoAppleEvent`.

GenericNoResultHandler

```
static pascal OSErr GenericNoResultHandler (
   const AppleEvent *theEvent,
   AppleEvent *theReply, long refCon);
```

Same as `GenericResultHandler`, except that `GenericNoResultHandler` is called for events that do not return a result. Consequently, `ApplyDesc` is called for events with multiple objects instead of `MapDesc`.

GenericGopherHandler

```
static pascal OSErr GenericGopherHandler (
   const AppleEvent *theEvent,
   AppleEvent *theReply, long refCon);
```

Calls the `DoAppleEvent` function for the application's current user selection.

GenericHandler

```
static pascal OSErr GenericHandler (
   const AppleEvent *theEvent,
   AppleEvent *theReply, long refCon,
   GenericDispatcherFunction fun);
```

Internal function used to perform common operations for the generic handlers.

GenericInsertionHandler

```
static pascal OSErr GenericInsertionHandler (
   const AppleEvent *theEvent,
   AppleEvent *theReply, long refCon);
```

Calls the `DoAppleEvent` function of the object specifed as the `keyAEInsertHere` parameter.

Object specifier resolution

Resolve

```
static void Resolve (
   const AEDesc *objectSpecifier,
   AEDesc *theToken);
```

Uses the current `callbackFlags` to call `AEResolve`. Throws an exception if `AEResolve` returns a non-zero error code.

SetCallbackFlags

```
static short SetCallbackFlags(short flags);
```

Sets the value of the `callbackFlags` static data member and returns its former value.

Object Support Library callbacks

MyAccessObject

```
static pascal OSErr MyAccessObject (
   DescType desiredClass,
   const AEDesc *containerToken,
   DescType containerClass, DescType keyForm,
   const AEDesc *keyData, AEDesc *theToken,
   long theRefcon);
```

The object accessor callback function calls `AccessObject` for the object pointed to by the `containerToken`.

MyCountObjects

```
static pascal OSErr MyCountObjects (
   DescType desiredType, DescType containerClass,
   const AEDesc *containerToken, long *result);
```

The count objects callback function calls `CountObjects` for the object pointed to by the `containerToken`.

MyCompareObjects

```
static pascal OSErr MyCompareObjects (
   DescType comparisonOperator,
   const AEDesc *theObject,
   const AE3esc *descOrObj, Boolean *result);
```

The compare objects callback function resolves the `theObject` object specifier and calls `CompareObjects` for the object pointed to by the resulting token.

MyGetMarkToken

```
static pascal OSErr MyGetMarkToken (
   const AEDesc *containerToken,
   DescType containerClass, AEDesc *result);
```

The get mark token callback function calls `GetMarkToken` for the object pointed to by the `containerToken`.

MyMarkObject

```
static pascal OSErr MyMarkObject (
   const AEDesc *theToken,
   const AEDesc *markToken, long markCount);
```

The mark object callback function calls `MarkObject` for the object pointed to by the `markToken`.

MyAdjustMarks

```
static pascal OSErr MyAdjustMarks (
   long newStart, long newStop,
   const AEDesc *markToken);
```

The adjust marks callback function calls `AdjustMarks` for the object pointed to by the `markToken`.

MyGetErrDesc

```
static pascal OSErr MyGetErrDesc (
   DescPtr *appDescPtr);
```

The get error descriptor callback function returns a pointer to the global error descriptor.

Member Functions: Protected

LinkObject

```
void LinkObject ();
```

Adds this object to the list of objects for the current event.

DelinkObject

```
void DelinkObject ();
```

Removes this object from the event's list of objects.

DestackObject

```
void DestackObject ();
```

Adjusts `stackTop` if object being delinked is at `stackTop`.

SetDisposable

```
void SetDisposable (Boolean disp);
```

Sets the value of the `disposable` data member to `disp`. There are very few reasons to modify `disposable` once it is set by the constructor.

IsDisposable

```
Boolean IsDisposable ();
```

Returns the value of the `disposable` data member.

DoEndEvent

```
void DoEndEvent (CAppleEventObject *mark);
```

Does the work for `EndEvent`, popping all CAppleEventObjects created since `BeginEvent` off the stack, and calling `DeleteObject` for each of them. It is implemented as a member function so it can see the next object in the list. `mark` points to the stack bottom.

GetDescAnyString

```
static void GetDescAnyString (
   const AEDesc *desc, unsigned char *str,
   Size maxSize, Boolean pascalString);
```

Helper function called by `GetDescString` and `GetDescStr`.

BeginEvent

```
static void BeginEvent ();
```

Called only by the CAppleEvent constructor to mark the bottom of a stack of Apple event objects corresponding to the event. CAppleEvent is declared a friend class so that it can access `BeginEvent` and `EndEvent`.

EndEvent

```
static void EndEvent ();
```

Called only by ~CAppleEvent destructor to delete all disposable Apple event objects created since the corresponding `BeginEvent` call.

CAppleEventSender 17

Introduction

CAppleEventSender allows you to construct and send Apple events to other applications and to receive and interpret replies.

A factored application should use the interfaces in CAppleEventObject to send events to itself.

Heritage

Base Class	None
Derived Classes	None

Using CAppleEventSender

Each CAppleEventSender object represents a single Apple event to be sent and an optional reply Apple event. Creating a CAppleEventSender object automatically creates an event. Member functions allow you to add parameters to the event and send it. If there is a reply, additional member functions let you extract error information or parameters from the reply. Deleting the CAppleEventSender object disposes the output event and reply.

If you do not know the Process Serial Number (PSN) of the application to which you want to send an event, call CAppleEventSender's `FindProcess` static member function to find the PSN before creating the sender. The PSN becomes an attribute of the newly created output Apple event.

Much of the work of constructing Apple events and interpreting replies lies in creating and decoding Apple event descriptors and object specifiers. CAppleEventSender does not provide any functions for these purposes. However, the CAppleEventObject class has a

number of static member functions you will find useful including, among others:

CreateDesc	GetDescString	CreateList
CreateLongDesc	GetDescStr	AppendDesc
CreateBooleanDesc	GetDescLong	CountItems
CoerceDesc	GetDescRect	GetNthDesc
GetDescData	GetDescPoint	

Table 17-1 Selected CAppleEventObject static member functions

Each of these functions corresponds to one or more Apple Event Manager functions. Instead of returning error codes, however, the CAppleEventObject functions throw an exception if an error code other than `noErr` is returned. Several of the functions are more specialized and have fewer parameters than the corresponding Apple Event Manager functions. See the CAppleEventObject chapter for details.

Data Members

CAppleEventSender has the following protected data members:

Data member	Type	Description
`eventDesc`	`AEDesc`	Output event
`replyDesc`	`AEDesc`	Reply event

`replyDesc` is a null descriptor (`descriptorType` is `typeNull`) until a reply is received.

Member Functions

All CAppleEventSender member functions throw an exception if any operating system function returns an error code other than `noErr`.

Creation and destruction

CAppleEventSender

```
CAppleEventSender (DescType eventClass,
   DescType eventID, ProcessSerialNumber *psn);
```

Constructor. `eventClass` and `eventID` specify the class and ID of the newly created output event. `psn` is the Process Serial Number of the application to which the event is ultimately sent.

The Process Serial Number of an application can be obtained by calling the `FindProcess` static member function.

~CAppleEventSender

```
~CAppleEventSender ();
```

Destructor. Disposes the output Apple event and, if one exists, the reply event.

Accessing

CAppleEventSender does not mirror every Apple Event Manager function. If you want to call one of those functions whose capability is not provided by a member function, you can get direct access to the output event and reply descriptors.

GetEvent

```
AppleEvent *GetEvent ();
```

Returns a pointer to the output event descriptor for use as an argument to Apple Event Manager functions. Caller must not dispose the event.

GetReply

```
const AppleEvent *GetReply ();
```

Returns a pointer to the reply event descriptor for use as an argument to Apple Event Manager functions. Caller must not dispose the reply.

Putting event parameters

PutParamPtr

```
void PutParamPtr (DescType key, DescType type,
   void *param, Size paramSize);
```

Adds a parameter with the specified `key` to the output Apple event. `type` specifies the data type, `param` points to the data, and `paramSize` specifies the length of the data in bytes. Calls `AEPutParamPtr`.

PutParamDesc

```
void PutParamDesc (DescType key, AEDesc *desc,
   Boolean disposeArg);
```

Adds a parameter with the specified `key` to the output Apple event. `desc` points to the data descriptor. If `disposeArg` is `TRUE`, the descriptor is disposed after the parameter has been added to the event. Calls `AEPutParamDesc`.

PutParamInsertionLoc

```
void PutParamInsertionLoc (AEDesc *objSpecifier,
   DescType relPos, Boolean disposeArg);
```

Adds a `keyAEInsertHere` parameter to the output Apple event. `objSpecifier` points to the object relative to which the insertion is to take place; `relPos` indicates whether the location is before (`kAEBefore`) or after (`kAEAfter`) the object, or whether it is to replace (`kAEReplace`) the object. If `disposeArg` is `TRUE`, the descriptor is disposed after the parameter has been added to the event.

Sending

SendWait

```
long SendWait (long timeOutInTicks,
   AESendMode flags);
```

Sends the output event to the target application and waits for a reply, returning the value of the `keyErrorNumber` parameter from the reply, or zero if none. `timeOutInTicks` specifies how long the function will wait before returning. The send mode `flags` are OR'ed with `kAEWaitReply` and used as an argument to `AESend`.

SendNoWait

```
void SendNoWait (AESendMode flags);
```

Sends the output event to the target application and returns immediately without waiting for a reply. The send mode `flags` are OR'ed with `kAENoReply` and used as an argument to `AESend`.

SendWaitMsg

```
long SendWaitMsg (long timeOutInTicks,
   AESendMode flags, short anALRT,
   unsigned char *msg);
```

Sends the output event to the target application and waits for a reply. `timeOutInTicks` specifies how long the function will wait before returning. The send mode `flags` are OR'ed with `kAEWaitReply` and used as an argument to `AESend`.

If the reply contains a `keyErrorNumber` parameter with a non-zero value, this function displays an alert with resource ID `anALRT`. Before displaying the alert, `ParamText` is called setting the substitution variables `^0` through `^3` to the event ID, error string

returned in the reply, error number returned in the reply and `msg`, respectively. `msg` may be `NULL`. After displaying the alert, this function throws an exception with error code `kSilentErr`.

Getting reply parameters

GetErrorString

```
void GetErrorString (Str255 errorString);
```

Returns up to 255 characters of the reply event's `keyErrorString` parameter, or an empty string if that parameter is not present.

GetReplyParamPtr

```
void GetReplyParamPtr (DescType key,
   DescType dataType, void *dataPtr,
   Size dataSize);
```

Gets a parameter with the specified `key` from the reply Apple event. `dataType` specifies the desired data type, `dataPtr` points to the data area to receive the data, and `dataSize` specifies the maximum length of the data in bytes. Calls `AEGetParamPtr`.

GetReplyParamDesc

```
void GetReplyParamDesc (DescType key,
   DescType desiredType, AEDesc *result);
```

Gets a parameter with the specified `key` from the reply Apple event. `desiredType` specifies the desired data type, `result` points to the descriptor to receive the data. Calls `AEGetParamDesc`.

Utility

FindProcess

```
static OSErr FindProcess(OSType typeToFind,
   OSType creatorToFind,
   ProcessSerialNumber *psn);
```

Locates a process on the current machine. `typeToFind` is the application type and `creatorToFind` is the signature of the process to be found. The result of the search is returned as the value of `psn`.

Unlike most THINK Class Library functions, `FindProcess` returns an error code instead of failing if the error code is not `noErr`. It behaves as it does because a failed search is not necessarily an application error.

17 CAppleEventSender

CApplication 18

Introduction

Every program must create a derived class of CApplication and create only one instance of this class. The application is the highest level in the chain of command. It is the only bureaucrat without a supervisor.

Heritage

Base Classes	CDirectorOwner CAppleEventObject
Derived Classes	None

Using CApplication

CApplication is one of the classes you must override to implement an application with the THINK Class Library. An application usually has one or more subordinate directors that govern the interaction between commands and windows. The most common kind of director is the document, which manages the interaction between a window and a file.

The application and the chain of command

The application is the root of the chain of command. It is the only bureaucrat without a supervisor. If the first object in the chain of command (the bureaucrat known as the gopher, stored in the global variable `gGopher`) can't handle a command, it passes the command to its supervisor. If no bureaucrat with a supervisor can handle the command, it ends up with the application. If the application doesn't handle the command, it is ignored.

Writing the main program

Your main program must create an application object and assign it to the global variable `gApplication`. After initializing your

application, call its Run function to get it started. Finally, call your application's Exit function to give it a chance to clean up. Your main routine should look like this:

```
void main()
{
   gApplication = new
         CYourApp( /* ctor args */);
   gApplication->Run();
   gApplication->Exit();
}
```

The derived class of your application must define or override these five member functions:

Constructor (CYourApp in the example)
OpenDocument
SetUpFileParameters
DoCommand
CreateDocument

Of course, your derived class can override additional member functions if necessary.

Handling low-memory situations

The CApplication class provides several member functions that deal with low-memory situations. These functions use a memory reserve called the rainy day fund.

When you call CApplication, specify how much memory your application should allocate for the rainy day fund. You also specify how many bytes of that fund should be available to satisfy Toolbox routine memory requests and how many bytes should be available to satisfy critical operation requests. These values are stored in the data members toolboxBalance and criticalBalance.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function. The grow zone function in the THINK Class Library calls the application's GrowMemory function.

The GrowMemory function tries several strategies to free memory in the heap. First, it calls the application's MemoryShortage function. Your derived class should override this function to release memory that is not crucial to execution. For example, your

MemoryShortage function might release a code segment it's not using or free a buffer it no longer needs. If the MemoryShortage function isn't able to release enough memory, GrowMemory starts using the rainy day fund.

GrowMemory looks first at the inCriticalOperation data member to relase memory from the rainy day fund:

If inCriticalOperation is	Leave this much in reserve
TRUE	toolboxBalance
FALSE	criticalBalance

A critical operation is one that occurs in response to a single Macintosh event, that might require a large amount of memory, and that should not cause the application to crash. For example, saving a file is a critical operation. You can use the function SetCriticalOperation to set the inCriticalOperation flag.

If GrowMemory still isn't able to release enough memory and the canFail flag is TRUE, GrowMemory returns without trying to allocate any more memory. A TRUE setting for the canFail flag means the application can deal with a failing memory request.

If the canFail flag is FALSE, GrowMemory tries to release all the memory left in the rainy day fund because the application is not prepared to deal with a failing memory request. Without sufficient memory, even after using the rainy day fund, it is likely that the application will crash.

You can use the routine SetAllocation to set and reset the canFail flag. In general, your application should be prepared to handle failing memory requests.

Global Variables and Data Members

The global variable gApplication points to your application object. The CApplication constructor sets this variable.

The data members of the application class handle events, memory shortage situations, standard file parameters, and flow of control. Your derived class may define additional data members.

Global variables

Variable	Type	Description
`gApplication`	`CApplication*`	The single instance of your application object
`gSystem`	`tSystem`	A global record with information about the Macintosh and System that your program is running with

Static data members

Only member functions of CApplication and its derived classes should use these static data members.

Data member	Type	Description
`cMaxSleepTime`	`Static long`	default for `WaitNextEvent`
`SelfPsnDesc`	`Static AEDesc`	`PSN` for sending to self

Event-related data members

The event data members handle the interaction between the Macintosh Toolbox Event Manager and your application. Your derived class should not manipulate these data members.

Data member	Type	Description
`itsSwitchboard`	`CSwitchboard*`	Points to the event-processing object.
`itsIdleChores`	`CChoreList*`	Chores to perform at idle time.
`itsUrgentChores`	`CChoreList*`	Chores to perform after the current event.
`urgentsToDo`	`Boolean`	`TRUE` if any urgent chores are pending.
`running`	`Boolean`	If `TRUE`, the program is still running.

Phase-related data members

This data member tells you what phase your application is in.

Data member	Type	Description
`phase`	`short`	The phase the application is in

These are the possible values for `phase`:

Value	Description
`appInitializing`	The application is initializing.
`appRunning`	The application is running.
`appQuitting`	The application is quitting.

Memory-related data members

Memory data members are used when your application is running out of memory. The `rainyDayFund` data member specifies the number of bytes to set aside for use when your application runs into a critical memory situation. You specify this amount in your application's constructor.

Data member	Type	Description
`rainyDayFund`	`Size`	Bytes of memory to set aside for critical memory situations.
`criticalBalance`	`Size`	Portion of rainy day fund to use for critical operations.
`toolboxBalance`	`Size`	Portion of rainy day fund to use for Toolbox operations that must not fail.
`tempAllocation`	`Size`	Portion of rainy day fund that a routine has borrowed.
`rainyDay`	`Handle`	Handle to the reserve memory.
`rainyDayUsed`	`Boolean`	Has rainy day fund been used?
`memWarningIssued`	`Boolean`	Has the user been alerted?

Data member	Type	Description
canFail	Boolean	TRUE if it is OK for memory request to fail.
inCriticalOperation	Boolean	TRUE if it is OK to use up to criticalBalance bytes from reserve.
newWindowOnStartup	Boolean	TRUE if auto-new when launched.

Standard file data members

The standard file data members are used in the ChooseFile member function as parameters to the Macintosh Toolbox SFPGetFile routine. You set these data members in your SetUpFileParameters member function.

Data member	Type	Description
sfNumTypes	short	Number of file types recognized.
sfFileTypes	SFTypeList	File types that the application recognizes.
sfFileFilter	FileFilterUPP	Filter for files to display.
sfGetDLOGHook	DlgHookUPP	Hook for handling get dialog.
sfGetDLOGid	short	Dialog resource ID for get file. Default is getDlgID (-4000).
sfGetDLOGFilter	ModalFilterUPP	Filter for get dialog events.
appResFile	short	Application resource file.

Editing-related data members

These data members keep track of undo and redo information for the application.

Data member	Type	Description
`lastTask`	`CTask*`	Last task handled by application.
`undone`	`Boolean`	Has `lastTask` been undone?

Apple event data members

These protected data members store information on how the application deals with Apple events.

Data member	Type	Description
`factoring`	`TCLFactorChoice`	When to factor Apple events
`recording`	`Boolean`	`TRUE` if application is recording

Utility data members

This protected data member keeps track of the last CDirector added to the application.

Data member	Type	Description
`lastAdded`	`CDirector*`	Last director added

Member Functions

The member functions of the CApplication class deal with application initialization and document handling. Because the application is the ultimate supervisor for all command objects, it also serves as the end of the chain of command.

Creation and destruction

The initialization functions set up the application's memory and file parameters.

CApplication

```
CApplication(short extraMasters,
   Size aRainyDayFund, Size aCriticalBalance,
   Size aToolboxBalance);
```

Constructor. At least one of the arguments must be non-zero. The arguments are passed to the `InitMemory` function, which is described later in this section.

In your derived class, you should implement a constructor that calls the CApplication constructor with appropriate argument values for your application. For example, if your application class is named `MyAppClass`:

```
MyAppClass()
: CApplication(kExtraMasters,
kRainyDaySize, kCriticalBalance,
kToolboxBalance)
{
   // Initialize MyAppClass
   // data members here...
}
```

The CApplication constructor initializes the following global variables. All are described in detail in Chapter 126, “Global Variables.”

`gApplication`	`gSignature`
`gSleepTime`	`gIBeamCursor`
`gWatchCursor`	`gUtilRgn`
`gGopher`	`gLastViewHit`
`gClicks`	`gSystem`

Finally, the constructor calls the following application member functions. The rest of this section describes these and other functions in detail.

`InitToolbox`	`InitMemory`
`InspectSystem`	`InstallPatches`
`MakeSwitchboard`	`MakeError`
`MakeDesktop`	`MakeClipboard`
`MakeDecorator`	`SetUpFileParameters`
`SetUpMenus`	

CApplication

```
CApplication();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IApplication` for backward compatibility.

~CApplication

```
~CApplication();
```

Destructor. The `~CApplication` destructor should never be called, as the application object is never deleted. The system automatically purges your application from memory when it quits. If your application needs to clean up before quitting, override `Exit` instead.

IApplication

```
void IApplication (short extraMasters,
   Size aRainyDayFund, Size aCriticalBalance,
   Size aToolboxBalance);
```

Initialization function provided for backward compatibility. If you do not specify any arguments to the CApplication constructor, you must call `IApplication` immediately after calling the constructor. If you specify any arguments to the constructor, you must not call `IApplication`.

InitToolbox

```
void InitToolbox ();
```

Initializes all of the Macintosh Toolbox managers. Your derived class should not need to override this function.

InitMemory

```
void InitMemory (short extraMasters,
   Size aRainyDayFund, Size aCriticalBalance,
   Size aToolboxBalance);
```

Initializes the Macintosh memory manager and sets up the `GrowZone` function used in low-memory situations. `extraMasters` is the number of times to call the Toolbox routine `MoreMasters`. `aRainyDayFund` is the number of bytes to set aside in the application heap to deal with low-memory situations. `aCriticalBalance` is the number of bytes to use from the rainy day fund for critical operations. `aToolboxBalance` is the number of bytes to use from the rainy day fund for Toolbox operations that can't fail.

aRainyDayFund must be greater than or equal to aCriticalBalance, and aCriticalBalance must be greater than aToolboxBalance.

The values that you use as arguments to this function depend on the memory requirements of your application. You can use these values to get started:

Argument	Suggested value
extraMasters	10
aRainyDayFund	45000
aCriticalBalance	40000
aToolboxBalance	20000

MakeSwitchboard

```
void MakeSwitchboard ();
```

Creates the application's switchboard and stores it in the data member itsSwitchboard. If you create your own derived class of CBartender, you should override this function in your application's derived class.

MakeError

```
void MakeError ();
```

Creates the error handler and stores it in the global gError. If you create your own error handler, you can create a derived class of CError and override this function to use your error handler.

MakeDesktop

```
void MakeDesktop ();
```

Creates the desktop and stores it in the global gDesktop. The default function creates the standard CDesktop desktop. If your application uses a nonstandard desktop, you should override this function in your derived class.

MakeClipboard

```
void MakeClipboard ();
```

Creates the Clipboard and stores it in the global gClipboard. If you create your own derived class of CClipboard, you should override this function to create a clipboard of your class.

MakeDecorator

```
void MakeDecorator ();
```

Creates the window decorator and stores it in the global `gDecorator`. The window decorator is responsible for the default sizes of windows and arranges them neatly on the desktop. If you want to implement a different decorator, you can create a derived class of CDecorator and override this function to use your decorator.

SetUpFileParameters

```
void SetUpFileParameters ();
```

Sets up the parameters that specify what kinds of files your application works on. The THINK Class Library passes some of these parameters to the Toolbox `SFPGetFile` function, which displays the standard get file dialog. Your derived class should override this function and call the inherited function to set up the default values.

For example:

```
void MyAppClass::SetupFileParameters()
{
   CApplication::SetupFileParameters();
   sfNumTypes = 1;
   sfFileTypes[0] = 'MyFi';
   gSignature = 'MyAp';
}
```

Your own function should set the following data members and globals:

Data	Description
`sfNumTypes`	The number of different types of files your application deals with. If your application can open any kind of file, use –1.
`sfFileTypes`	The list of all file types that your application deals with.
`gSignature`	The four-character signature of your application. Although it's not a data member, this function is the place to set up this global.
`sfFileFilter`	Optional. A pointer to a function that filters the file names your application can deal with.

Data	Description
`sfGetDLOGHook`	Optional. A pointer to a dialog hook function for the standard get file dialog.
`sfGetDLOGid`	Optional. The resource ID of the standard get file dialog you're using. Do not change this data member if you want to use the default dialog.
`sfGetDLOGFilter`	Optional. A pointer to a dialog filter procedure.

To learn about file filter functions, dialog hook functions, and dialog filters, see *Inside Macintosh Volume I*, Chapter 20, "The Standard File Package."

SetUpMenus

```
void SetUpMenus ();
```

Calls `MakeBartender`, reads all the menu information from your application's `MBAR 1` resource, and builds the menus. This function also builds the desk accessory menu.

If your application uses menus that need to be built on the fly, you should override this function. For instance, if your application uses a **Font** menu, you would build it in this function. Be sure to call `inherited SetUpMenus` to make sure all the regular menus get built. See the description of CBartender, particularly the section "Resource-based menus" in Chapter 23, "CBartender," for more information about building these kinds of menu.

MakeBartender

```
void MakeBartender ();
```

Creates the bartender object, which handles all interactions with the menu bar and menu items, and stores it in the global `gBartender`. If you create your own derived class of `CBartender`, you should override this function in your derived class.

InspectSystem

```
void InspectSystem ();
```

Fills in the fields of the `gSystem` global, which gives you information about the Macintosh and the system software that your application is running under.

gSystem is declared as follows:

```
typedef struct
{
   Boolean  hasWNE         : 1;
   Boolean  hasColorQD     : 1;
   Boolean  hasGestalt     : 1;
   Boolean  hasAppleEvents : 1;
   Boolean  hasAliasMgr    : 1;
   Boolean  hasEditionMgr  : 1;
   Boolean  hasHelpMgr     : 1;
   Boolean  hasScriptMgr   : 1;
   Boolean  hasFPU         : 1;
   short    scriptsInstalled;
   short    systemVersion;
} gSystem;
```

The field scriptsInstalled tells you how many scripts are in use. The field systemVersion gives you the version of the Macintosh system that's running. The version number is given as two byte-long numbers. For example, if systemVersion is 0x0607, the System version number is 6.0.7.

Advanced initialization

You should not need to override these functions. If you are an advanced Macintosh programmer, you may want to take advantage of the fact that they're called in the CApplication constructor.

InstallPatches

```
void InstallPatches ();
```

Installs patches to Toolbox routines. The default function patches LoadSeg to make sure that CODE resources can be loaded. If a code resource can't be loaded, the patch raises an exception. The default function also patches ExitToShell to call RemovePatches.

RemovePatches

```
void RemovePatches ();
```

Removes the Toolbox patches made in InstallPatches.

Accessing

GetPhase

```
short GetPhase ();
```

Returns current application phase. The possible values are appInitializing, appRunning, and appQuitting.

Command

The command functions handle events that the application takes care of. Because calls frequently get passed up the chain of command—from the pane, to the document, to the application—the application is the last chance to handle commands. Most of the command functions don't do anything. They're empty functions that prevent the call from being passed on.

NotifyClean

```
void NotifyClean(CTask *theTask);
```

In CApplication, `NotifyClean` is the same as `Notify`. In CDocument, `NotifyClean` is the same as `Notify`, except that `NotifyClean` does not mark the document as dirty.

DoKeyDown

```
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

You should handle key-down events within a pane or a document. For applications, this function doesn't do anything. It exists in the event that the `DoKeyDown` function call gets passed up the chain of command. Your derived class should not override this function.

DoAutoKey

```
void DoAutoKey (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

You should handle auto-key events within a pane or a document. For applications, this function doesn't do anything. It exists in the event that the `DoAutoKey` function call gets passed up the chain of command. Your derived class should not override this function.

DoKeyUp

```
void DoKeyUp (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

You should handle key-up events within a pane or a document. For applications, this function doesn't do anything. It exists in the event that the `DoKeyUp` function call gets passed up the chain of command. Your derived class should not override this function.

Note

The Macintosh system event mask is usually set to mask out key-up events. If you need to get key-up events, be sure to reset the system event mask with the Toolbox routine `SetEventMask`.

DoCommand

```
void DoCommand (long theCommand);
```

This is the only application command function that really does more than just return. `DoCommand` handles application commands that the user chooses from the menu. These are the commands the default `DoCommand` function handles:

Command	Action
`cmdNew`	Calls `CreateDocument`.
`cmdOpen`	Calls `ChooseFile`. If the reply is good, calls `OpenDocument`. The default `ChooseFile` function calls `SFPGetFile` with the values you specified in the `SetUpFileParameters` function. Your application must override the `OpenDocument` function.
`cmdClose`	If the front window is a desk accessory, close it. Your document class should handle this command to close documents.
`cmdQuit`	Calls your application's `Quit` function. The default `Quit` function calls the `Quit` functions of the supervisors (directors) of each open window.

Table 18-1 Commands handled by DoCommand function

Command	Action
`cmdUndo`, `cmdCut`, `cmdCopy`, `cmdPaste`, `cmdClear`	If the front window is a desk accessory, always calls `SystemEdit`. Your document class should handle these commands to edit documents.
`cmdToggleClip`	Calls the clipboard's `Toggle` function.

Table 18-1 Commands handled by DoCommand function

If your application defines its own application-related events, your application class should override this function. If your `DoCommand` function gets a command that it does not handle, it should call the inherited `DoCommand`.

UpdateMenus

```
void UpdateMenus ();
```

This function enables the appropriate menu items just before a pop-up menu is displayed . The default function enables the **Quit** command. If the application is in the foreground, it enables the **Show/Hide Clipboard** command. If the application is in the background, this function enables the `cmdClose`, `cmdUndo`, `cmdCut`, `cmdCopy`, `cmdPaste`, and `cmdClear` commands for desk accessories.

Your document's `UpdateMenus` function should enable the appropriate **Edit** menu commands for the document.

PackageAppleEvent

```
CAppleEvent *PackageAppleEvent (
   const AppleEvent *theEvent,
   AppleEvent *theReply, long theRefCon);
```

Packages an incoming `AppleEvent` and its default reply into a `CAppleEvent` object. This function returns a `CAppleEvent` object. If you create a derived class of `CAppleEvent`, you need to override this function. `CAppleEvent` is described in Chapter 15, "CAppleEvent."

Memory management

These functions deal with critical memory situations in your application. The `InitMemory` function sets the function

GrowZoneFunc as the function to call in low-memory situations. This function calls your application's GrowMemory function to try to reclaim enough memory to continue.

Note

GrowZoneFunc is defined in CError.h.

RequestMemory

```
void RequestMemory (Boolean aCanFail);
```

Changes the canFail data member, which controls how GrowMemory releases memory from the rainy day fund. If aCanFail is FALSE, GrowMemory is more conservative about releasing memory for a failing memory request. If aCanFail is TRUE, GrowMemory will exhaust the rainy day fund to satisfy the request.

Instead of calling this function directly, you should use the utility routine SetAllocation (described in Chapter 127, "TCL Utilities"). SetAllocation calls the application's RequestMemory function and returns the previous value so you can restore it.

In addition to SetAllocation, TCLUtilities.h also defines two constants, kAllocCantFail (FALSE) and kAllocCanFail (TRUE). Because these constants are more self-documenting than TRUE and FALSE, you should use them as arguments to SetAllocation instead of the plain Boolean values.

Here is an example:

```
void AClass::AFunction()
{
   Boolean oldAlloc;

   oldAlloc = SetAllocation(kAllocCanFail);
   myHandle = NewHandleCanFail(50000L);
   SetAllocation(oldAlloc);

   if (myHandle == NULL)
   {
      couldn't get 50000 bytes
   }
   else
   {
      go on with this operation
   }
}
```

Remember that if the argument you pass to `SetAllocation` or `RequestMemory` is `kAllocCantFail`, `GrowMemory` will do whatever it can to get the memory. If there still isn't enough memory after the rainy day fund has been exhausted, your program will probably crash.

SetCriticalOperation

`void SetCriticalOperation (Boolean isCritical);`

Sets the `inCriticalOperation` data member, which controls how `GrowMemory` releases memory from the rainy day fund. If `inCriticalOperation` is `TRUE` and a memory shortage causes GrowMemory to be called, GrowMemory tries to release enough memory in the rainy day fund to leave `toolboxBalance` bytes free.

Your application should call the utility routine `SetCriticalOperation` (described in Chapter 127, "TCL Utilities") instead of calling the `SetCriticalOperation` function of `gApplication`.

Use this routine when there is an operation that must complete and for which your application might need to use some of the memory in the rainy day fund. The critical operation should release any memory that it allocated.

For example, when your application is saving a file, you may need to allocate an extraordinary amount of memory. In this case, you

would call SetCriticalOperation to let GrowMemory know that, if necessary, you're willing to use more of the rainy day fund. Once the operation is complete, you would dispose of the memory.

Note that the Run function resets inCriticalOperation to FALSE every time through the event loop.

GrowMemory

```
long GrowMemory (Size bytesNeeded);
```

The GrowZoneFunc function that the Memory Manager calls in low-memory situations invokes this function to try to reclaim memory. GrowMemory calls the application's MemoryShortage function.

If MemoryShortage isn't able to release enough memory, GrowMemory starts to use the rainy day fund. If inCriticalOperation is TRUE, GrowMemory uses as much of the rainy day fund as possible, while still leaving toolboxBalance bytes in the fund. If inCriticalOperation is FALSE, it uses as much as possible, while still leaving criticalBalance bytes in the fund.

If GrowMemory still wasn't able to release enough memory from the rainy day fund, and canFail is FALSE, the entire rainy day fund is released. If the entire rainy day fund has been exhausted, this function calls OutOfMemory.

MemoryShortage

```
void MemoryShortage (Size bytesNeeded);
```

The GrowMemory function calls your application's MemoryShortage function to try to free memory. The default MemoryShortage function does nothing, so your derived class should override this function. Your MemoryShortage function should try to free bytesNeeded bytes of memory, and it should disable menu commands that won't work in low-memory situations.

Warning

Your MemoryShortage function must not allocate any memory.

MemoryReplenished

```
void MemoryReplenished ();
```

This function is called when the memory situation is no longer critical. The default `MemoryReplenished` function does nothing, so your derived class should override this function. Your `MemoryReplenished` function should enable the commands you disabled in the `MemoryShortage` function. You might also want to reallocate memory that you released in that function.

OutOfMemory

```
long OutOfMemory (Size bytesNeeded);
```

A memory request cannot be satisfied and `canFail` was `FALSE`, so the application won't handle the memory allocation failure. This function calls `Failure(memFullError, 0)`. If any memory is left, the top-level handler displays an alert; it is likely that the application will crash.

You can override this function if you can free up more memory, or if you can exit the application gracefully without allocating memory. Normally, you would try to free enough memory in your `MemoryShortage` function. If there are more drastic ways to release memory, implement them here. This function should return `1` if it successfully releases memory, or `0` if it can't.

Warning

`OutOfMemory` should not allocate any memory or call any routines that allocate memory.

Execution

Most of these functions handle system-related events. The only execution function your application should override is `Exit`.

Run

```
void Run ();
```

Runs the application. This function calls the application's `Process1Event` function until the user chooses to quit.

Before running the program, this function calls your application's `Preload` function to open or print documents that the user selected and opened from the Finder.

Your application should not override this function.

Process1Event

```
void Process1Event ();
```

Processes and dispatch one event. This function calls the `ProcessEvent` function of the switchboard (stored in the data member `itsSwitchboard`). If you installed an urgent chore, this function calls its `Perform` function.

Preload

```
void Preload ();
```

If the user opened or chose to print files from the Finder, `Preload` calls the application's `OpenDocument` function for each document. If the user chose the **Print** command, `Preload` calls the gopher's `DoCommand(cmdPrint)` function. After processing all the files, this function calls the application's `StartUpAction` function.

StartUpAction

```
void StartUpAction (short numPreloads);
```

Offers an opportunity to perform any startup actions. `NumPreloads` is the number of files that the user selected from the Finder. If `numPreloads` is 0, the default function calls the `DoCommand(cmdNew)`function of the gopher (usually the application). The effect of the default function is to open an untitled document at startup.

Suspend

```
void Suspend ();
```

Your application is about to be suspended under MultiFinder or System 7. The default function calls the `Suspend` function of CDirectorOwner and sets the global `gInBackground` to `TRUE`.

Resume

```
void Resume ();
```

Your application has come back to the foreground under MultiFinder or System 7. The default function calls the `Resume` function of CDirectorOwner and sets the global `gInBackground` to `FALSE`.

SwitchToDA

```
void SwitchToDA ();
```

A desk accessory is becoming active. The default function calls the application's `Suspend` function. Your application should not override or use this function.

SwitchFromDA

```
void SwitchFromDA ();
```

Your application is becoming active after a desk accessory was active. The default function calls your application's `Resume` function. Your application should not override or use this function.

Idle

```
void Idle (EventRecord *macEvent);
```

Handles periodic tasks. It also checks to see if a critical memory situation is no longer critical. `Idle` calls the `Dawdle` function of the gopher and of each of the gopher's supervisors. It also calls the `Perform` functions of all chores. Your application should not override this function.

Quit

```
Boolean Quit ();
```

The user wants to quit the application. This function calls the `Quit` function of the supervisor (a director) of each open window. Any of the directors can cancel quitting. Your application should not override this function.

Note

`Quit` for directors returns a Boolean value. If it returns `FALSE`, quitting is cancelled.

Exit

```
void Exit ();
```

The application is about to exit. Your main program should call `Exit` before it terminates; the THINK Class Library will not call the function for you. The default function does nothing.

The `Exit` function is the last chance you have to clean up after your application. For example, this is a good place to delete any temporary files you've created.

JumpToEventLoop

```
void JumpToEventLoop ();
```

This function is for compatibility with earlier versions of the THINK Class Library. It calls `Failure(kSilentErr, 0)` to force a return to the top-level exception handler in `Run`.

Document

The document functions of an application deal with creating and opening documents. In your derived class, the only document functions you need to override are `CreateDocument` and `OpenDocument`. All the other document functions are used internally to implement standard behavior.

CreateDocument

```
void CreateDocument ();
```

Creates a new document. The default `DoCommand` function calls the application's `CreateDocument` function when the user chooses **New** from the **File** menu. Your application must override this function. Your `CreateDocument` function needs to create a new document, initialize it, and then call its `NewFile` function.

OpenDocument

```
void OpenDocument (SFReply *macSFReply);
```

Opens an existing document. The `macSFReply` record specifies the document to open. The default `DoCommand` function calls the application's `OpenDocument` function when the user chooses **Open** from the **File** menu.

Note

> `DoCommand` actually calls the application's `ChooseFile` function first and, if the reply is good, it calls the `OpenDocument` function. Your `OpenDocument` function can assume that the `macSFReply` record is valid.

Your derived class must override this function. Your `OpenDocument` function needs to create a new document, initialize it, and then call its `OpenFile(macSFReply)` function.

ChooseFile

```
void ChooseFile (SFReply *macSFReply);
```

Displays a standard get file dialog and places the reply in the `macSFReply` record. This function calls the Toolbox routine `SFPGetFile` with the file parameters you specified in the `SetUpFileParameters` function.

You can call `ChooseFile` from other objects to get the name and location of a file. Don't forget to check the `macSFReply.good` field to make sure that the information in the record is valid.

Your derived class should not override this function unless it uses a different technique to open a document.

Periodic task

AssignIdleChore

```
void AssignIdleChore (CChore *theChore);
```

Adds `theChore` to the application's list of chores. The application calls the `Perform` function of each chore at idle time. Your application should not override this function.

Note

If you want to remove the chore from the list later, you must keep a pointer to the chore object.

CancelIdleChore

```
void CancelIdleChore (CChore *theChore);
```

Removes `theChore` from the application's list of chores. Your application should not override this function.

AssignUrgentChore

```
void AssignUrgentChore (CChore *theChore);
```

Adds `theChore` to the application's list of urgent chores. The application calls the `Perform` function of each urgent chore after handling the current event, and then removes the chore. Your application should not override this function.

Class Resources

Resource	Description
`'MBAR' 1`	The application's menu bar
`'STR#' 129`	Low-memory warning strings

Apple Event Support

For information regarding Apple event member functions, please refer to the supplemental information provided in the `Apple event info` file in the `Apple Event Classes` folder.

18 CApplication

CArray 19

Introduction

CArray implements a resizable array.

Heritage

Base Class	CCollection
Derived Classes	CRunArray
	CVoidPtrArray

Using CArray

Use CArray when you want an array you can resize. The elements may be any size, as long as they're all the same size.

To insert a new element into the array, use `InsertAtIndex`. It moves the element in that slot and all elements below it down one slot. Its actions are illustrated in Figure 19-1.

Figure 19-1 Before and after InsertAtIndex

Use `SetItem` to update the value of an existing element. The function replaces the item at that slot with the new element, as illustrated in Figure 19-2.

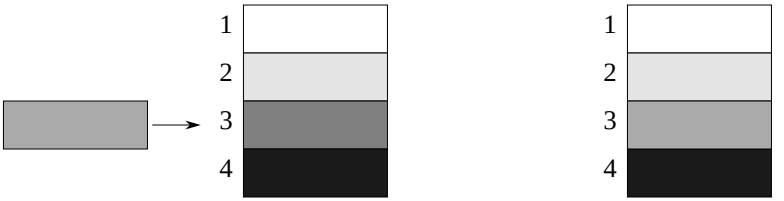


Figure 19-2 Before and after SetItem

To delete an element, use `DeleteItem`. The function moves all the elements below the deleted element up one slot.

CArray stores the elements in its array in one large piece of memory that it allocates dynamically. It uses the data member `blockSize` to control how to grow and shrink this memory. If there are no empty slots when you try to insert an element, CArray grows memory by `blockSize` slots. When you delete an element and there are `blockSize` empty slots left, CArray shrinks the memory by `blockSize` slots. By default, `blockSize` is 3.

CArray has two functions that move elements in the array: `Swap` and `MoveItemToIndex`. To store an element while moving it, CArray has a temporary storage slot at the end of the array. To make sure an element in temporary storage isn't accidentally overwritten, those member functions set the flag `usingTemporary` whenever they put something in the temporary storage slot. This slot is not counted in the `slots` data member, which is the total number of slots that can permanently store elements.

```
typedef struct tMovedElementInfo
{
   long originalIndex;
   long newIndex;
}  tMovedElementInfo;
```

The following table describes the correct usage of broadcast change messages.

Message	Info parameter usage
`arrayInsertElement`	Index of new element
`arrayDeleteElement`	Index of deleted element
`arrayMoveElement`	Pointer to a `tMovedElementInfo`
`arrayGoingAway`	`NULL`
`arrayElementChanged`	Index of changed array element

Table 19-1 Broadcast change messages

Data Members

CArray defines the following data members:

Data member	Type	Description
`blockSize`	`short`	Number of slots to allocate when more space is needed.
`slots`	`long`	Total number of slots allocated.
`hItems`	`Handle`	Items in the array.
`elementSize`	`long`	Size of each element in bytes.
`lockChanges`	`Boolean`	If `TRUE`, you can't insert or delete in the array.
`usingTemporary`	`Boolean`	If `TRUE`, temporary element storage buffer is in use.

Member Functions

Creation and destruction

CArray

```
CArray (long anElementSize = 0,
   short aBlockSize = 3);
```

Constructor. `anElementSize` is the size of each array element in bytes. `aBlockSize` is the number of elements to add to the array when CArray allocates more space.

~CArray

```
~CArray();
```

Destructor. Frees the memory occupied by the array elements.

IArray

```
void IArray(long anElementSize);
```

Initialization function provided for backward compatibility. If no constructor arguments are specified, the array must be further initialized by calling `IArray`. If any constructor arguments are specified, `IArray` must not be called. See constructor for a description of the `anElementSize` argument.

Copy

```
void *Copy();
```

Returns a copy of this array.

Accessing

SetBlockSize

```
void SetBlockSize (short aBlockSize);
```

Sets the number of elements to add to the array when CArray allocates more space. `IArray` initializes this value to 3.

SetLockChanges

```
Boolean SetLockChanges (Boolean fLockChanges);
```

If `fLockChanges` is `TRUE`, you can't use the `InsertAtIndex` and `DeleteItem` functions. This function sets `lockChanges` to `fLockChanges` and returns the previous value of `lockChanges`. `CPtrArray` uses this function to prevent you from corrupting the array when you iterate over its elements with `DoForEach`.

Insertion and deletion

InsertAtIndex

```
void InsertAtIndex (void *itemPtr, long index);
```

Inserts the element at `itemPtr` at the position `index`. If `index` is already occupied, moves the element in that slot and all the elements below it down one slot. If `index` is beyond the last position, adds the item to the end of the array. If necessary, this function allocates a larger block of memory for the array. This function calls the `BroadcastChange` function with `arrayInsertElement` as the reason.

DeleteItem

```
void DeleteItem (long index);
```

Deletes the element at index `index`. This function moves all the elements below `index` up one slot. If the number of free slots is greater than `blockSize`, this function reallocates a block of memory that's smaller by `blockSize` slots.

SetItem

```
void SetItem (void *itemPtr, long index);
```

Puts the element at `itemPtr` into the array at position `index`. `Index` must be within the array. This function calls the `BroadcastChange` function with `arrayElementChanged` as the reason.

Add

```
void Add (void *itemPtr);
```

Appends an item to the end of the array, extending the size by one element. Sends dependents an `arrayInsertElement` message.

SetArrayItem

```
void SetArrayItem (void *itemPtr, long index);
```

Sets the contents of the array element at `index`. `index` must be within the array. Sends dependents an `arrayElementChanged` message.

Membership

GetItem

```
void GetItem (void *itemPtr, long index);
```

Returns the element at position `index` in the block memory at `itemPtr`. `index` must be within the array. `itemPtr` must point to a block large enough to hold one element.

GetArrayItem

```
void GetArrayItem (void *itemPtr, long index);
```

Retrieves the contents of the array element. `index` must be within the array. `itemPtr` must point to storage large enough to hold one element.

Search

```
long Search (void *itemPtr,
   CompareFunc compare);
```

Finds an element that satisfies a condition. The function `compare` tests for the condition. It takes two arguments. The first is `itemPtr` and the second is a pointer to an element in the array. `compare` should return 0 when an element satisfies the condition. `itemPtr` is a pointer to some data that you can use in the comparison. It can be a pointer to data like that in the array, a pointer to another type of data, or `NULL`.

Returns the array index of the found item if the search succeeds; otherwise, returns `BAD_INDEX`, defined as `(long)(-1)`.

Declare `compare` like this:

```
int compare( void *itemPtr,
             void *arrayElt );
```

Moving

MoveItemToIndex

```
void MoveItemToIndex (long currentIndex,
   long newIndex);
```

Moves the element at `currentIndex` to `newIndex`. This function moves the elements between the old and new positions up or down a slot. This function calls the `BroadcastChange` function with `arrayMoveElement` as the reason.

Swap

```
void Swap (long index1, long index2);
```

Swaps the two items: Moves the item at `index1` to `index2`, and moves the item at `index2` to `index1`. For both items, this function calls the `BroadcastChange` function with `arrayElementChanged` as the reason.

Resizing

CArray uses these protected functions to resize its block of memory. You should need to use them only if you are creating a derived class of CArray. Only derived classes of CArray can use them.

Resize

```
void Resize (long numSlots);
```

Resizes the array so that it has `numSlots` slots. This function performs no error checking.

MoreSlots

```
void MoreSlots();
```

Adds `blockSize` slots to the array.

Temporary storage

CArray uses these functions to move elements into and out of the temporary storage slot. You should need to use them only if you are creating a derived class of CArray. In the THINK Class Library, only derived classes of CArray can use them.

CopyToTemporary

```
void CopyToTemporary (long index);
```

Places the element in slot `index` into temporary storage. This function also sets `usingTemporary` to `TRUE`.

CopyFromTemporary

```
void CopyFromTemporary (long index);
```

Places an item from temporary storage into slot `index`. This function also sets `usingTemporary` to `FALSE`.

Offset

CArray uses this function to find the offset into the array of an element. You should need to use it only if you are creating a derived class of CArray.

ItemOffset

```
long ItemOffset (long itemIndex);
```

Returns the offset into the array of the slot `itemIndex` in bytes.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the entire array to the stream.

GetFrom
```
void GetFrom (CStream& aStream);
```
Reads the entire array from the stream.

Member Functions: Protected

InsertItem
```
void InsertItem (long index);
```
Inserts space for an item into the array at the specified index. Any items at or above `index` are moved to the next higher position. If `index` is greater than the current size, then the item is appended to the end of the array.

PutItems
```
void PutItems (CStream& aStream);
```
Writes the elements of this array to the stream. Called by `PutTo`.

GetItems
```
void GetItems (CStream& aStream);
```
Reads the elements of this array from the stream. Called by `GetFrom`.

Store
```
void Store (void *itemPtr, long index);
```
Puts the element at `itemPtr` into the array at position `index`. Whenever possible, use `Set` instead. This function performs no error checking.

Retrieve
```
void Retrieve (void *itemPtr, long index);
```
Returns the element at position `index` in the block memory at `itemPtr`. Whenever possible, use `Get` instead. This function performs no error checking.

CArrayIterator 20

Introduction

CArrayIterator lets you iterate through the elements of a CArray or CList array.

Heritage

Base Class	CCollaborator
Derived Classes	CVoidPtrArrayIterator

Using CArrayIterator

CArrayIterator provides a straightforward and error-resistant way to loop through the items in a CArray. For example:

```
typedef struct { short i; ...} MyRec;
CArray *array(sizeof(MyRec));
...
CArrayIterator iter(array, kStartAtBeginning);
MyRec item;

while (iter.Next(&item))
{
   if (item.i == 0)
      ...
}
```

The `Next` function fetches the next record from the array and advances the cursor (see below). Successive calls to `Next` traverse the entire array. When the cursor is at the end of the array, `Next` returns `FALSE`.

You could simply loop through the array with an index, but that sort of code is more error-prone: if the loop inserts or deletes elements in the array, it usually fails.

More than one iterator can operate on the same array at the same time. For example, here is a loop that removes duplicate records from the array above:

```
CArrayIterator iter(array, kStartAtBeginning);
CArrayIterator check(array, 0);
MyRec r, dup;

while (iter.Next(&r))
{
   check.MoveTo(iter.GetCursor());
   while (check.Next(&dup))
      if (dup.i == r.i && ...)
         array->DeleteItem(check.GetCursor());
}
```

The cursor ranges from 0 to the number of elements in the array, and is logically positioned at the start of the array, at the end, or *between* two array items. For example, if an array has 5 items, a cursor with value 3 is positioned between items 3 and 4, as illustrated in Figure 20-1. `Next` returns item 4 and `Prev` returns item 3.

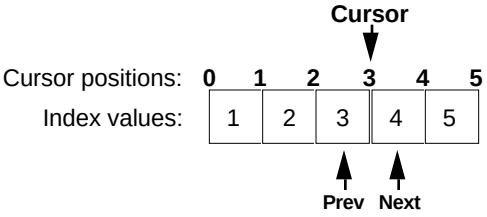


Figure 20-1 Relationship of cursor position to index

Unlike a loop index, an iterator is not required to go only forward or only backward; by calling `Next`, `Prev`, and `MoveTo`, you may go forward, backward, or jump around in an arbitrary way during the life of the iterator.

If the array associated with the iterator is deleted, the cursor is set to 0 and the `Next` and `Prev` functions return `FALSE`.

Data Members

CArrayIterator has the following protected data members:

Data member	Type	Description
`itsArray`	`CArray*`	Pointer to array that you iterate through
`curPosition`	`long`	Cursor position

Member Functions

Creation and destruction

CArrayIterator

```
CArrayIterator (CArray *array,
                CursorPosition start);
```

Constructor. `array` specifies the array to iterate through and `start` specifies where iteration begins. `start` may be specified as `kStartAtBeginning`, `kStartAtEnd`, `StartBefore(`*index*`)`, or `StartAfter(`*index*`)`, where *index* is the index of an array item, from 1 to the number of items in the array.

Advancing and retreating

Next

```
Boolean Next (void *itemptr);
```

If there are more items beyond the cursor, this function fetches the next item pointer, replacing the value of `itemptr`, increments the cursor, and returns `TRUE`. Otherwise, returns `FALSE`.

Prev

```
Boolean Prev (void *itemptr);
```

If there are items before the cursor, this function fetches the previous item pointer, replacing the value of `itemptr`, decrements the cursor, and returns `TRUE`. Otherwise, returns `FALSE`.

Positioning

GetCursor

```
CursorPosition GetCursor ();
```

Returns the current cursor position. The cursor is always in the range 0 to the number of items in the array.

MoveTo

```
void MoveTo (CursorPosition pos);
```

Resets the current cursor position. `pos` may be specified as `kStartAtBeginning`, `kStartAtEnd`, `StartBefore(`*index*`)` or `StartAfter(`*index*`)`, where *index* is the index of an array item, from 1 to the number of items in the array.

AtEnd

```
Boolean AtEnd ();
```

Returns `TRUE` if iterator is positioned at the end of the array or if the array has been deleted.

AtBeginning

```
Boolean AtBeginning ();
```

Returns `TRUE` if iterator is positioned at the beginning of the array or if the array has been deleted.

Member Functions: Protected

Collaboration

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

Receives `arrayDeleteElement` and `arrayInsertElement` reason codes as values of `reason` when called with `aProvider` equal to `itsArray`.

CArrayIterator 20

Introduction

CArrayIterator lets you iterate through the elements of a CArray or CList array.

Heritage

Base Class	CCollaborator
Derived Classes	CVoidPtrArrayIterator

Using CArrayIterator

CArrayIterator provides a straightforward and error-resistant way to loop through the items in a CArray. For example:

```
typedef struct { short i; ...} MyRec;
CArray *array(sizeof(MyRec));
...
CArrayIterator iter(array, kStartAtBeginning);
MyRec item;

while (iter.Next(&item))
{
   if (item.i == 0)
      ...
}
```

The `Next` function fetches the next record from the array and advances the cursor (see below). Successive calls to `Next` traverse the entire array. When the cursor is at the end of the array, `Next` returns `FALSE`.

You could simply loop through the array with an index, but that sort of code is more error-prone: if the loop inserts or deletes elements in the array, it usually fails.

More than one iterator can operate on the same array at the same time. For example, here is a loop that removes duplicate records from the array above:

```
CArrayIterator iter(array, kStartAtBeginning);
CArrayIterator check(array, 0);
MyRec r, dup;

while (iter.Next(&r))
{
   check.MoveTo(iter.GetCursor());
   while (check.Next(&dup))
      if (dup.i == r.i && ...)
         array->DeleteItem(check.GetCursor());
}
```

The cursor ranges from 0 to the number of elements in the array, and is logically positioned at the start of the array, at the end, or *between* two array items. For example, if an array has 5 items, a cursor with value 3 is positioned between items 3 and 4, as illustrated in Figure 20-1. `Next` returns item 4 and `Prev` returns item 3.

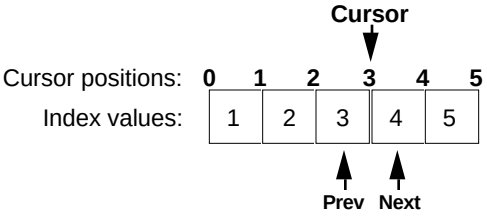


Figure 20-1 Relationship of cursor position to index

Unlike a loop index, an iterator is not required to go only forward or only backward; by calling `Next`, `Prev`, and `MoveTo`, you may go forward, backward, or jump around in an arbitrary way during the life of the iterator.

If the array associated with the iterator is deleted, the cursor is set to 0 and the `Next` and `Prev` functions return `FALSE`.

Data Members

CArrayIterator has the following protected data members:

Data member	Type	Description
`itsArray`	`CArray*`	Pointer to array that you iterate through
`curPosition`	`long`	Cursor position

Member Functions

Creation and destruction

CArrayIterator

```
CArrayIterator (CArray *array,
                CursorPosition start);
```

Constructor. `array` specifies the array to iterate through and `start` specifies where iteration begins. `start` may be specified as `kStartAtBeginning`, `kStartAtEnd`, `StartBefore(`*index*`)`, or `StartAfter(`*index*`)`, where *index* is the index of an array item, from 1 to the number of items in the array.

Advancing and retreating

Next

```
Boolean Next (void *itemptr);
```

If there are more items beyond the cursor, this function fetches the next item pointer, replacing the value of `itemptr`, increments the cursor, and returns `TRUE`. Otherwise, returns `FALSE`.

Prev

```
Boolean Prev (void *itemptr);
```

If there are items before the cursor, this function fetches the previous item pointer, replacing the value of `itemptr`, decrements the cursor, and returns `TRUE`. Otherwise, returns `FALSE`.

Positioning

GetCursor

```
CursorPosition GetCursor ();
```

Returns the current cursor position. The cursor is always in the range 0 to the number of items in the array.

MoveTo

```
void MoveTo (CursorPosition pos);
```

Resets the current cursor position. `pos` may be specified as `kStartAtBeginning`, `kStartAtEnd`, `StartBefore(`*index*`)` or `StartAfter(`*index*`)`, where *index* is the index of an array item, from 1 to the number of items in the array.

AtEnd

```
Boolean AtEnd ();
```

Returns `TRUE` if iterator is positioned at the end of the array or if the array has been deleted.

AtBeginning

```
Boolean AtBeginning ();
```

Returns `TRUE` if iterator is positioned at the beginning of the array or if the array has been deleted.

Member Functions: Protected

Collaboration

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

Receives `arrayDeleteElement` and `arrayInsertElement` reason codes as values of `reason` when called with `aProvider` equal to `itsArray`.

CArrayPane 21

Introduction

CArrayPane is an abstract class for displaying the contents of an array object in a scrolling list.

Heritage

Base Class	CTable
Derived Classes	None

Using CArrayPane

The dependency mechanism is described in Chapter 33, "CCollaborator."

CArrayPane is a derived class of CTable that displays the contents of a CArray object. The array pane is a table that has one row for each array element. By default, there's only one cell per row. CArrayPane uses the dependency mechanism to keep the table display synchronized with the array. Whenever you add, delete, or change an element, CArrayPane's `ProviderChanged` function is called, which automatically adds or deletes rows and redraws the appropriate areas.

To use CArrayPane, you must create a derived class that draws the contents of a cell. If your data can be represented as a string, override `GetCellText` to return a string. Otherwise, override `DrawCell` to draw the cell. These member functions are defined in Chapter 113, "CTable."

To associate a CArray with a CArrayPane, call the CArrayPane's `SetArray` function, with the CArray object and a flag indicating whether this CArrayPane owns the CArray. If the CArrayPane owns the array, it will automatically delete it. If you want to share a single array among multiple views, you should pass `FALSE` for the flag,

and make sure that your director- or document-derived class deletes the array.

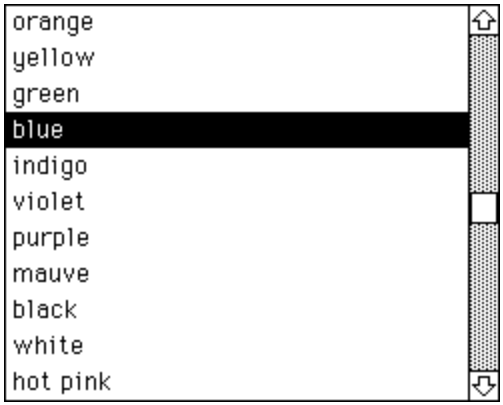


Figure 21-1 A CArrayPane

Data Members

CArrayPane defines these protected data members:

Data member	Type	Description
`itsArray`	`CArray*`	The array you're displaying
`ownsArray`	`Boolean`	`TRUE` if `destructor` should delete `itsArray`

Member Functions

Creation and destruction

CArrayPane

```
CArrayPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizELASTIC,
   SizingOption aVSizing);
```

Constructor. The arguments to this function are identical to those for the CPane constructor.

CArrayPane

```
CArrayPane ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IArrayPane` for backward compatability.

~CArrayPane

```
~CArrayPane();
```

Destructor. If the array is owned, this deletes it.

IArrayPane

```
void IArrayPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialization function provided for backward compatibility. If no constructor arguments are specified, the array pane must be further initialized by calling `IArrayPane`. If any constructor arguments are specified, `IArrayPane` must not be called. Arguments are the same as for the constructor, except there are no default values.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

Initializes an array pane from a resource template.

Accessing

SetArray

```
void SetArray (CArray *anArray,
   Boolean fOwnership);
```

Sets the array that this pane displays. If `fOwnership` is `TRUE`, this pane owns the array and deletes it when the pane is destroyed. Otherwise, the pane doesn't own the array and doesn't delete it.

If the pane was already displaying an array that it owned, it deletes the array.

GetArray

```
CArray *GetArray ();
```

Returns the array object this pane is displaying.

Size

ChangeSize

```
void ChangeSize (Rect *delta, Boolean redraw);
```

Changes the size of the array pane. Because an array table has only one column, this function sets the column's width to the frame's width.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

Change notification

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

The array has been changed and a `BroadcastChange` function has been called on it. This function handles these reasons:

Reason	Action
`arrayInsertElement`	Add a new row for the element
`arrayDeleteElement`	Delete the element's row
`arrayMoveElement`	Redraw all rows between the element's old and new rows
`arrayElementChanged`	Redraw the element's row

Table 21-1 Reasons that CArrayPane handles

CArrowPopupPane 22

Introduction

CArrowPopupPane is a pop-up menu pane that displays a downward-pointing arrow. A pop-up menu pane is a pane that owns a CPopupMenu and pops up the menu when clicked.

Heritage

Base Class	CPopupPane
Derived Classes	None

Using CArrowPopupPane

CArrowPopupPane follows Apple Human Interface Guidelines for a pop-up menu that displays no text when it is not popped up. It displays a downward-pointing arrow in a box with a drop shadow, as in Figure 22-1. It is a derived class of CPopupPane, described in Chapter 83, “CPopupPane.”

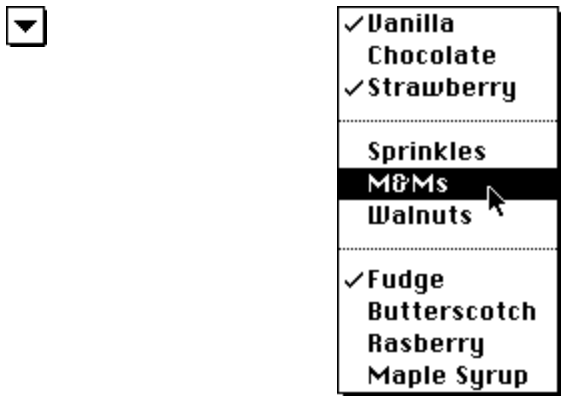


Figure 22-1 A CArrowPopupPane

Data Members

CArrowPopupPane defines no data members.

Member Functions

Creation and destruction

CArrowPopupPane

```
CArrowPopupPane(short menuID,
   Boolean fAutoSelect, Boolean fMultiSelect,
   CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHEncl, short aVEncl,
   Boolean fRadioStyle = FALSE);
```

Constructor. `menuID` is the resource ID of the MENU resource. `fAutoSelect` controls whether the menu automatically checks and unchecks menu items when selected. `fMultiSelect` controls whether multiple menu items may be selected. `fRadioStyle` controls whether the menu is to be used like a group of radio buttons, with only one item checked at a time.

Note

The rest of the arguments are described under CPane in Chapter 71. A CArrowPopupPane's sizing options are always `sizFIXEDSTICKY`.

CArrowPopupPane

```
CArrowPopupPane ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IArrowPopupPane` for backward compatability.

IArrowPopupPane

```
void IArrowPopupPane(short menuID,
Boolean fAutoSelect, Boolean fMultiSelect,
CView *anEnclosure, CBureaucrat *aSupervisor,
short aHEncl, short aVEncl,
Boolean fRadioStyle = FALSE);
```

Initialization function provided for backward compatibility. `IArrayPane` should only be called if the default constructor was used. `radioStyle` and `privateMenu` are set to `FALSE`.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally for initializing from a resource template. Each derived class of CArrowPopupPane overrides this function to use its own resource template.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the pane's pop-up menu when the menu isn't selected. Override this function if your derived class draws the menu differently.

Member Functions: Protected

IArrowPopupPaneX

```
void IArrowPopupPaneX ();
```

Performs common initialization.

CalcPopupPoint

```
point CalcPopupPoint (Point hitPt,
   Short modifierKeys, long when);
```

Calculates the popup position in global coordinates.

CBartender 23

Introduction

The bartender is the object that manages the menu bar, menus, and menu items. The bartender is an object of class CBartender. There is only one object of this class, stored in a global variable.

Note

Earlier versions of the THINK Class Library used CBarOwner, a derived class of CBartender, to prevent excessive menu bar flashing. This version of CBartender includes this improvement.

Heritage

Base Class	None
Derived Classes	None

Using CBartender

In the THINK Class Library, almost every menu item has a command number associated with it. The bartender maintains a table that maps menu items to command numbers. You should never try to access the bartender's data structures directly. Instead, use the access member functions.

You should not need to make a derived class of CBartender. If, for some reason, you do need to create a derived class, you'll also need to override the `SetUpMenus` member function in your application class to initialize your bartender.

To manipulate the menus or menu items, call the functions of the global bartender object stored in the global variable `gBartender`.

Creating standard menus

To associate a command number with a menu item, append the command number to the end of the menu item in your `'MENU'` resource. The menu item text and the command number are separated by the character #. Figure 23-1 shows what the **File** menu looks like in ResEdit:

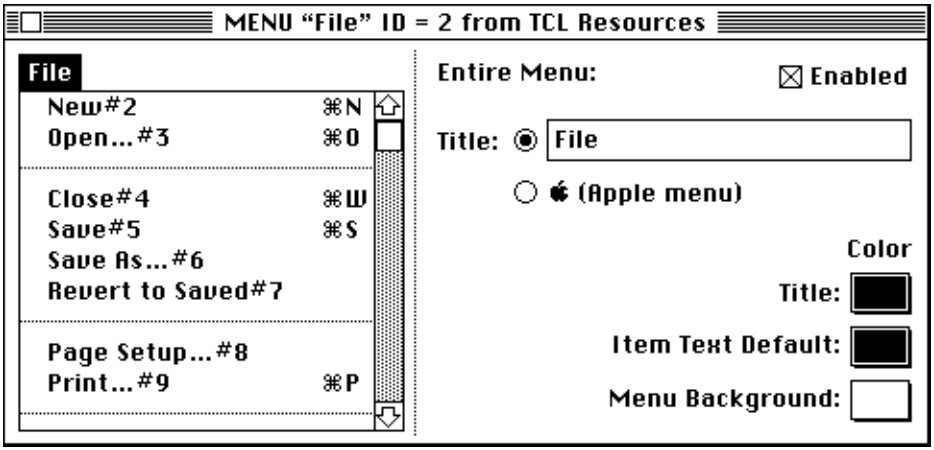


Figure 23-1 The File menu in ResEdit

If you don't append a command number to a menu item, the bartender automatically associates the command number `cmdNull` with it.

The bartender builds its tables from the information in the `'MBAR'` resource you pass to its constructor. Your application's `'MBAR'` resource should contain the menu IDs of all the menus that will appear in the menu bar.

Note

By default, the application's `SetUpMenus` member function uses the `'MBAR'` resource with an ID = 1.

The `'MENU'` *resources for these menus are in the file* `TCL Resources`*.*

You can use any menu ID for your application's menus. The THINK Class Library reserves the menu IDs in Table 23-1 for certain menus.

Menu title	Menu ID	Mnemonic
Apple	1	`MENUapple`
File	2	`MENUfile`
Edit	3	`MENUedit`
Font	10	`MENUfont`
Size	11	`MENUsize`

Table 23-1 Menu IDs

Below are the commands that the THINK Class Library defines. In Symantec C++, they are in the file `Commands.h`. Remember that the THINK Class Library reserves the commands in the range 1 to 1023.

Table 23-2 lists standard **File** menu commands.

Command	ID
`cmdQuit`	1
`cmdNew`	2
`cmdOpen`	3
`cmdClose`	4
`cmdSave`	5
`cmdSaveAs`	6
`cmdRevert`	7
`cmdPageSetup`	8
`cmdPrint`	9

Table 23-2 File menu command IDs

Table 23-3 lists the standard **Edit** menu commands.

Command	ID
`cmdUndo`	16
`cmdCut`	18
`cmdCopy`	19
`cmdPaste`	20
`cmdClear`	21
`cmdToggleClip`	22
`cmdSelectAll`	23

Table 23-3 Edit menu command IDs

Table 23-4 lists the text style commands.

Command	ID
cmdPlain	30
cmdBold	31
cmdItalic	32
cmdUnderline	33
cmdOutline	34
cmdShadow	35
cmdCondense	36
cmdExtend	37

Table 23-4 Text style command IDs

Table 23-5 lists the text alignment commands.

Command	ID
cmdAlignRight	40
cmdAlignLeft	41
cmdAlignCenter	42
cmdJustify	43

Table 23-5 Text alignment command IDs

Table 23-6 lists the line spacing commands.

Command	ID
cmdSingleSpace	50
cmd1HalfSpace	51
cmdDoubleSpace	52

Table 23-6 Line spacing command IDs

Table 23-7 lists miscellaneous commands.

Command	ID	Description
cmdNull	0	Command that does nothing
cmdOK	100	OK button in dialog box
cmdCancel	101	Cancel button in dialog box
cmdAbout	256	About Application request

Table 23-7 Miscellaneous command IDs

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command -1 in your `'MENU'` resource. The bartender returns the negative number of the menu ID

in the high word and the menu item number in the low word. This is the same return format used for resource-based menus, described next.

Resource-based menus

Your application may use menus that don't have menu commands associated with each item. The most common example is a **Font** menu. Use the reserved **Font** menu in your `'MBAR'` resource, and add the resources with the `AddResMenu` Toolbox routine. Here's how you might write your `SetUpMenus` function in Symantec C++ to include a font menu:

```
void CYourApp::SetUpMenus()
{
   MenuHandle macMenu;

   inherited::SetUpMenus();
   AddResMenu(GetMHandle(MENUfont),
      'FONT');
   SetDimOption(MENUfont, dimNONE);
   SetUnchecking(MENUfont, TRUE);
}
```

Note

To learn about `SetDimOption` and `SetUnchecking`, see "Dimming and checking menu items" later in this chapter.

When you build a menu this way, the bartender is unable to associate a command number with the menu items. In this case, the bartender's `FindCmdNumber` returns the negative of the menu ID in the high word and the item ID in the low word.

The values that `FindCmd` *returns in these cases are the negative of what* `MenuSelect` *would have returned.*

For example, if you choose the ninth font in the **Font** menu, `FindCmdNumber` returns -655369 (`0xFFF5FFF7`). Since the number is negative, you know that no command number is associated with the item. To get the menu ID and the menu item, negate the value and split it into two words. In this example, the return value becomes 655369 (`0x000A0009`), which means that the menu ID is 10 and the menu item is 9.

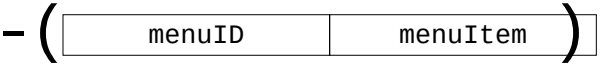


Figure 23-2 How FindCmdNumber builds command numbers

Note

There are functions to extract the high word and low word of a long integer. In Symantec C++, the functions are `HiShort` and `LoShort` (defined in `Global.h`).

You can add menu items to existing menus. You might add the **Font** menu to a general text-handling menu, or you might want to have a menu with the names of all the documents your application has opened. The important thing to remember is to add all these menu items at the end of the existing menu. Otherwise, the bartender will get confused.

Creating hierarchical menus

For a detailed description of hierarchical menus, see Inside Macintosh V, *Chapter 13, "The Menu Manager."*

Most of the time you'll be able to set up your hierarchical menus in your resource file. The key thing to remember is that the item to which the hierarchical menu is attached uses the command key equivalent and the item mark differently. The command key value must be `0x1B` (Control-[), and the value of the item mark is the resource ID of the hierarchical menu.

Sometimes, you can't specify a hierarchical menu in a resource. In these cases, use the `InsertHierMenu` member function. This function adds the hierarchical menu to an existing menu in the menu bar and to the bartender's table. Generally, you'll specify hierarchical menus in your resource file. Use this function to insert hierarchical menus from your program.

Note

For hierarchical menus to work correctly, you must set them up either in the resource file or with a call to `InsertHierMenu`. If you use only the Macintosh Toolbox routines, the bartender's item-checking functions will not work.

This example shows how to write a function to insert a hierarchical **Font** menu as the third item in a **Text** menu from your program:

```
void CYourApp::SetUpMenus()
{
   MenuHandle  macMenu;
   short       MENUtext, itemNO;

   CApplication::SetUpMenus();
   gBartender->InsertHierMenu(MENUfont,
      cmdNull, MENUtext, 2);
   AddResMenu(GetMHandle(MENUfont),
      'FONT');
   SetDimOption(MENUfont, dimNONE);
   SetUnchecking(MENUfont, TRUE);
}
```

Dimming and checking menu items

The dimming, undimming, and unchecking take very little time. You won't notice a delay between the time you click on the menu bar and when the menu is displayed.

The bartender includes member functions to let you enable and disable and check and uncheck menu items. When you click the menu bar, the bartender calls the `UpdateMenus` function of every bureaucrat in the chain of command. Generally, all menu items start out dimmed and unchecked. Each bureaucrat then enables the menu items that pertain to it. Once the appropriate items have been enabled and checked, the Toolbox routine `MenuSelect` displays all the menus.

Suppose you click the menu bar of a text-processing application. When you click the menu bar, the bartender disables all the menu items before the Toolbox displays the menu. Then, the application enables all the application-related menu items: **New**, **Open** and **Quit**, for example. The document enables all the document-related items: **Save**, **Save As**, **Revert** (if the document's been changed), and so on. A pane might check the current font and size in the **Font** menu. Finally, the menu appears on the screen with the correct items checked and enabled.

Your application, document, and pane each have a function called `UpdateMenus`, which needs to enable each menu item. To make sure that item enabling happens from the general (application) to the specific (pane), be sure to call inherited `UpdateMenus` first in your own `UpdateMenus` function.

You can use the bureaucrat member functions `SetDimOption` and `SetUnchecking` in your application `SetUpMenus` function to modify this behavior. `SetDimOption` lets you specify whether the

bartender should dim all, some, or none of the items when you click on the menu bar. For **Font** menus, for instance, dimming all the font names does not make sense if they are only going to be re-enabled again.

Global Variables and Data Members

The global variable `gBartender` points to the single instance of the bartender. When you want to manipulate menus, call this object's functions. The bartender uses its data members to maintain the mapping between command numbers and menu items. You should not need to access or alter the data members.

Global variable

Variable	Type	Description
`gBartender`	`CBartender`	The global bartender

Data members

Data member	Type	Description
`numMenus`	`short`	The number of menus available
`theMenus`	`MenuEntryH`	A table with an entry for each available menu
`choreAssigned`	`Boolean`	`TRUE` if there's a pending chore to redraw the menu bar
`forceMBarUpdate`	`Boolean`	`TRUE` if `UpdateMenuBar` always draws the menu bar

Member Functions

To use these functions to manipulate menus and menu items, call the functions of the bartender object stored in the global `gBartender`.

For example, to disable the **Open** command, you would write:

```
gBartender->DisableCmd(cmdOpen);
```

To change the name of the **Copy** command to **Copy Picture**, you might write:

```
gBartender->SetCmdText(cmdCopy,
   "\pCopy Picture");
```

In the rare case when you need the actual menu handle or other information about the menu, use the `FindMacMenu` or `FindMenuItem` functions.

Creation and destruction

CBartender

```
CBartender (short MBARid = 0);
```

Constructor. The application's `SetUpMenus` function creates the bartender and stores a pointer to it in the global variable `gBartender`. `MBARid` is the ID of the `'MBAR'` resource that the bartender uses to build its menu tables. The default `SetUpMenus` uses 1 as its `MBARid`.

CBartender

```
CBartender ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. It can also be used in combination with `IBartender` for backward compatibility.

~CBartender

```
~CBartender();
```

Destructor. The bartender's destructor doesn't do anything, as the bartender is never deleted.

IBartender

```
void IBartender (short MBARid);
```

Initialization function provided for backward compatibility. `IBartender` may only be called if no constructor arguments are specified.

Insertion and deletion

AddMenu

```
void AddMenu (short MENUid, Boolean install,
   short beforeID);
```

This function adds a menu to the bartender's list. `MENUid` is the resource ID of the menu. If `install` is `TRUE`, install it into the menu bar as well as into the bartender's list. `BeforeID` specifies the menu before which this menu is installed. If `beforeID` is 0, the menu is added to the end of the menu bar. A `beforeID` of -1 is a hierarchical or pop-up menu.

RemoveMenu

```
void RemoveMenu (short MENUid);
```

If nothing else is using the specified menu, remove it from the menu bar and from the bartender's list. Call `RemoveMenu` on all the hierarchical menus it uses. `MENUid` is the resource ID of the menu.

To know when it can remove a menu, the THINK Class Library keeps a use count for each menu. `AddMenu` increments a menu's use count each time it adds a menu. `RemoveMenu` decrements the use count and removes the menu only if the count is zero.

InsertInBar

```
void InsertInBar (short MENUid, short beforeID);
```

Inserts a menu in the menu bar and in the bartender's list. This function is the same as `AddMenu(MENUid, TRUE, beforeID)`.

DeleteFromBar

```
void DeleteFromBar (short MENUid);
```

Removes the specified menu from the menu bar, but leaves it in the list. After deleting the menu, this function creates a chore of class `CMBarChore` and sends it to `gApplication` as an urgent chore. The chore will redraw the menu bar the next time through the event loop.

InsertHierMenu

```
void InsertHierMenu (short hMENUid, long cmdNo,
   short inMENUid, short afterItem);
```

Inserts a hierarchical menu in another menu. `hMENUid` is the menu ID of the hierarchical menu. If you want to be able to dim a hierarchical menu, provide a command number in `cmdNo`; if you're

not going to dim the hierarchical menu, pass in `cmdNull` for `cmdNo`. `inMENUid` is the menu ID of the menu that will contain the hierarchical menu. `afterItem` is the item number after which the hierarchical menu should be inserted. The title of the menu is used as the item name for the hierarchical menu.

Item manipulation

EnableCmd

```
void EnableCmd (long cmdNo);
```

Enables the menu item associated with `cmdNo`.

DisableCmd

```
void DisableCmd (long cmdNo);
```

Disables the menu item associated with `cmdNo`.

EnableMenu

```
void EnableMenu (short MENUid);
```

Enables the menu with resource ID `MENUid`.

DisableMenu

```
void DisableMenu (short MENUid);
```

Disables the menu with resource ID `MENUid`.

EnableMenuBar

```
void EnableMenuBar();
```

Enables all the menus currently in the menu bar.

DisableMenuBar

```
void DisableMenuBar();
```

Disables all the menus currently in the menu bar.

SetCmdText

```
void SetCmdText (long cmdNo, Str255 theText);
```

Changes the text of the menu item associated with `cmdNo` to `theText`.

GetCmdText

```
void GetCmdText (long cmdNo, Str255 theText);
```

Stores the text of the menu item associated with `cmdNo` in `theText`. If `cmdNo` is not associated with a menu item, the contents of `theText` are unchanged.

CheckMarkCmd

```
void CheckMarkCmd (long cmdNo, Boolean checked);
```

Places (or removes) a check mark next to the menu item associated with `cmdNo`.

Item insertion and deletion

InsertMenuCmd

```
void InsertMenuCmd (long cmdNo, Str255 theText,
   short MENUid, short afterItem);
```

Inserts a new item in a menu. `cmdNo` is the command number of the new item. `theText` is the text of the new item. `MENUid` is the ID of the menu into which you're inserting the new item. `afterItem` specifies after which item to insert the new item.

RemoveMenuCmd

```
void RemoveMenuCmd (long cmdNo);
```

Removes the menu item associated with the command `cmdNo` from its menu.

Look-up

These functions let you know which command number is associated with which menu item and which menu contains the menu item associated with a particular command number.

FindCmdNumber

```
long FindCmdNumber (short MENUid, short itemNo);
```

Return the command number associated with the menu `MENUid` and `itemNo`. If the item has no command associated with it, this function returns `MENUid` in the high word and `itemNo` in the low word.

This is the function called by the desktop and switchboard on the bartender to convert a menu selection or a menu key into a command number. Some menu items, like font menus, don't have

commands associated with them. For these menus, `FindCmdNumber` returns the menu ID and item number.

FindMenuItem

```
void FindMenuItem (long cmdNo, short *MENUid,
   MenuHandle *macMenu, short *itemNo);
```

Returns the menu ID, the handle to the menu, and the item number associated with a given command number. If the command number isn't associated with a menu item, all three items are set to zero.

FindItemText

```
short FindItemText (short MENUid,
   Str255 itemStr);
```

Returns the item number in `MENUid` with the specified text. Return zero if the menu doesn't have an item called `itemStr`. This function is useful for keeping track of items that don't have command numbers associated with them.

FindMacMenu

```
MenuHandle FindMacMenu (short MENUid);
```

Returns a handle to the Macintosh menu with ID `MENUid`. If the menu is not in the bartender's table, return `NULL`.

FindMenuIndex

```
short FindMenuIndex (short MENUid);
```

Returns the index of the menu `MENUid` in the bartender's table. This is an internal function. You should not use this function.

Appearance

SetDimOption

```
void SetDimOption (short MENUid,
   DimOption aDimming);
```

Sets the dim option for the specified menu. By default, all the items in the menu are dimmed when you click the menu bar. The

UpdateMenus function of each of your bureaucrats needs to enable the appropriate items.

If aDimming is…	The bartender will…
dimNONE	Never dim any of the menu items.
dimSOME	Dim only the menu items that have command numbers associated with them.
dimALL	Dim all of the menu items. Each bureaucrat's UpdateMenus function must enable the items for the commands it handles. This is the default.

Table 23-8 Menu-dimming options

SetUnchecking

```
void SetUnchecking (short MENUid,
   Boolean anUnchecking);
```

Sets the checking option for the specified menu. By default, all items are checked when you click the menu bar. If you set this option to TRUE, all the items will be unchecked, and your UpdateMenus functions must check the appropriate menu items.

If anUnchecking is…	The bartender will…
TRUE	Uncheck all the menu items at menu selection. Your UpdateMenus function should check the appropriate items. Set this option for menus such as **Font** or **Style** menus.
FALSE	Not uncheck any menu items at menu selection. This is the default because most menu items never need to be checked.

Table 23-9 Menu-unchecking options

UpdateAllMenus

```
void UpdateAllMenus();
```

This function of the bartender is called before menu selection. First it disables and unchecks all the items in the menus according to the dimming and unchecking options. Then it calls the gopher's UpdateMenus function to enable the appropriate menu items. You should not use or override this function.

UpdateMenuBar

```
void UpdateMenuBar();
```

Call `DrawMenuBar` if the menu bar needs to be redrawn, . The menu bar needs to be redrawn if the `forceMBarUpdate` flag is `TRUE`, or if the enabled state of any menu in the menu bar doesn't match the last recorded state.

SetMenuBarState

```
long SetMenuBarState(long barState);
```

Enables or disables all the menus in the menu bar and returns the former state. If `barState` is 0, it disables all the menus. If `barState` is `-1`, it enables all the menus. This function has no effect on hierarchical menus.

Command extraction

These are three internal functions that the bartender uses to find commands associated with menu items and to build its internal data structures. You should not use these functions. These are THINK Class Library functions:

```
void ExtractCommands (MenuEntryP theEntry);
void ParseItemString (Str255 itemStr,
   long *cmdNo);
void ExtractHierMenus (MenuHandle macMenu,
   short index);
```

23 CBartender

CBitMap 24

Introduction

CBitMap is a class you use to work with Macintosh bitmaps. You should be familiar with QuickDraw bitmaps before using this class.

Heritage

Base Class	None
Derived Classes	None

Using CBitMap

CBitMap gives you member functions to draw into a bitmap, to stamp a bitmap on the current drawing port, and to get a bitmap from the current drawing port. This class doesn't provide a drawing function because it is only intended for manipulating bitmaps. The class CBitMapPane gives you a pane for drawing bitmaps.

To learn more about transfer modes, see Inside Macintosh Volume I, *Chapter 6, "QuickDraw."*

Use the `CopyFrom` member function to copy bits from your bitmap object to the bitmap of the current QuickDraw port. To copy bits from the current port to your bitmap object, use the `CopyTo` member function. The transfer mode determines how the bitmap is copied. Use the `GetXferMode` member function to examine the current transfer mode and the `SetXferMode` member function to set the transfer mode. The default transfer mode is `srcCopy`.

To draw into an off-screen bitmap, bracket your drawing routines with calls to `BeginDrawing` and `EndDrawing`. When you create the bitmap object, you have the choice of creating a drawing port for it. If you create a port, your bitmap will have a complete QuickDraw environment when you draw to it. If you don't, the drawing routines use the drawing environment of the current port.

In general, you should create a port for a bitmap whenever you intend to draw to a bitmap. If you use a bitmap only to hold an image that you copied from or want to copy to the current port, you don't need to create a port.

For example, this procedure draws a diagonal line in an off-screen bitmap:

```
void DrawIntoOffScreenBitMap(void)
{
   CBitMap *myBitMap;

   myBitMap = new CBitMap(300, 300, TRUE );
   myBitMap->BeginDrawing;
   MoveTo(10, 10);
   LineTo(200, 200);
   myBitMap->EndDrawing;
};
```

Once you finish drawing into your bitmap, you can use the `CopyFrom` member function to stamp it into the current port. To save the bitmap as a paint (`'PNTG'`) file, you use the CPNTGFile class.

Data Members

CBitMap defines these data members:

Data member	Type	Description
`macPort`	`GrafPtr`	Grafport for the bitmap if requested.
`savePort`	`GrafPtr`	Used internally to save the original drawing port if necessary.
`macBitMap`	`BitMap`	The bitmap.
`saveBitMap`	`BitMap`	Used internally to save the original drawing port's bitmap if necessary.
`xferMode`	`short`	Bitmap transfer mode. Initially `srcCopy`.

Member Functions

Creation and destruction

CBitMap

```
CBitMap (short width, short height,
   Boolean makePort);
```

Constructor. CBitMap allocates memory for the bitmap. The origin is set to (0,0) and the transfer mode is set to `srcCopy`. If `makePort` is `TRUE`, CBitMap creates a QuickDraw grafport whose port bits point to the bitmap. In general, you should pass `TRUE` for `makePort` if you intend to draw to the bitmap and `FALSE` if you use the bitmap only to hold an image.

CBitMap

```
CBitMap ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IBitMap` for backward compatibility.

~CBitMap

```
~CBitMap ();
```

Destructor. Releases memory allocated to the bitmap and closes `macPort` if it was opened.

IBitMap

```
void IBitMap (short width, short height,
   Boolean makePort);
```

Initialization function provided for backward compatibility. `IBitMap` may only be called if no constructor arguments are specified. Arguments are same as for the constructor.

Dispose

```
void Dispose ();
```

Disposal function provided for backward compatibility. `Dispose` simply calls operator delete. This function is only available if `TCL_USE_DISPOSE` is defined in the project.

Accessing

GetBounds

```
void GetBounds (theBounds *LongRect);
```

Stores the bounding rectangle of the bitmap in `theBounds`. This rectangle describes the size and coordinate system of the bitmap.

SetBoundsOrigin

```
void SetBoundsOrigin (short hOrigin,
   short vOrigin);
```

Sets the coordinates of the upper-left corner of the bitmap. This function changes the coordinate system of the bitmap.

SetXferMode

```
void SetXferMode (short aXferMode);
```

Sets the transfer mode of the bitmap.

GetXferMode

```
short GetXferMode ();
```

Returns the transfer mode of the bitmap.

PixelIsBlack

```
Boolean PixelIsBlack (LongPt pixelPos);
```

Returns `TRUE` if the pixel specified by `pixelPos` is black. Returns `FALSE` if it isn't or if the specified position is not in the bitmap.

Image copying

CopyTo

```
void CopyTo (LongRect *fromRect,
   LongRect *toRect, RgnHandle maskRgn);
```

Copies bits to this bitmap from the current QuickDraw port (`thePort`). `fromRect` is a rectangle in `thePort`'s bitmap, and `toRect` is a rectangle in this bitmap. If `maskRgn` is `NULL`, this function does no clipping. Otherwise, the transfer is clipped to `maskRgn`, a region in this bitmap's coordinates. The transfer mode is stored in the `xferMode` data member.

CopyFrom

```
void CopyFrom (LongRect *fromRect,
   LongRect *toRect, RgnHandle maskRgn);
```

Copies bits from this bitmap to the current QuickDraw port (`thePort`). `FromRect` is a rectangle in this bitmap, and `toRect` is a rectangle in `thePort`'s bitmap. If `maskRgn` is `NULL`, this function does no clipping. Otherwise, the transfer is clipped to `maskRgn`, a region in `thePort`'s coordinates. The transfer mode is stored in the `xferMode` data member.

Drawing preparation

BeginDrawing

```
void BeginDrawing ();
```

Prepares for drawing to this bitmap. If the bitmap doesn't have its own port, set the port bits of the current QuickDraw port to this bitmap. If the bitmap has its own port, this function saves the current port and sets the bitmap's port as the current port.

EndDrawing

```
void EndDrawing ();
```

Resets the port to the state before drawing. If the bitmap doesn't have its own port, restores `thePort`'s port bits from `saveBitMap`. Otherwise, resets the port to `savePort`.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Private

CBitMapX

```
void CBitMapX ()
```

Performs common constructor initialization.

Friend Functions

The following friend functions overload operator << and >> to provide alternatives to the Put and Get functions.

---

operator <<

```
friend CStream & operator << (CStream& s,
   CBitMap *p)
```

---

operator >>

```
friend CStream & operator >> (CStream& s,
   CBitMap *p)
```

CBitMapPane 25

Introduction

CBitMapPane is a derived class of CPanorama for drawing bitmaps in a pane.

Heritage

Base Class	CPanorama
Derived Classes	None

Using CBitMapPane

CBitMapPane is a pane with an associated bitmap. The drawing member function of a bitmap pane copies from the bitmap to the pane's drawing port. Your program should therefore draw into the bitmap pane's bitmap before the drawing member function gets called. The Art Class demonstration program, which uses a derived class of CBitMapPane, draws directly on the screen and on the associated bitmap in a mouse task. When the Macintosh generates an update event, the pane's drawing function copies the associated bitmap right on the screen, giving a very smooth update.

For static images, a picture pane is easier to use. See Chapter 78, "CPicture."

Though you can use CBitMapPane directly, you will probably want to use a derived class of it so you can add data members and member functions tailored for your application. Your program can use whatever technique is appropriate to fill the bitmap with an image. If the image doesn't change, your program can just read the bitmap from a file and rely on the update event to stamp the image on the screen. If it's an interactive image, you might use a mouse task as Art Class does. If it's a complex image and you want to give the illusion of a quick update, you do the drawing to the bitmap in your derived class's `Draw` function with CBitMap's `BeginDrawing` and `EndDrawing` functions, and then call the inherited `Draw` member function to stamp the image on the screen.

Data Members

CBitMapPane defines this data member:

Data member	Type	Description
itsBitMap	CBitMap*	The bitmap to stamp on the pane

Member Functions

Creation and destruction

CBitMapPane

```
void CBitMapPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   LongRect *aBounds,
   CBitMap *aBitMap, Boolean makePort);
```

Initializes a panorama. The first eight arguments to this routine are identical to those for CPane.

The aBounds rectangle is a long rectangle that defines the bounds of the panorama. The bounds define the height and width of aBitMap as well as its coordinate system.

If aBitMap is NULL, CBitMapPane creates a new bitmap object for the pane; otherwise, it uses aBitMap as the pane's bitmap. When you use an existing bitmap, be sure to pass the correct bounds to CBitMapPane. Call the bitmap's GetBounds function to get its bounds, and pass the results to CBitMapPane.

CBitMapPane passes the makePort parameter to CBitMap's constructor. If aBitMap is not NULL, this parameter is ignored. The makePort parameter specifies whether the bitmap should have a drawing environment. In general, you should always pass TRUE for this parameter.

Note

The other arguments are described in Chapter 71, "CPane."

CBitMapPane

```
CBitMapPane ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IBitMapPane` for backward compatibility.

IBitMapPane

```
void IBitMapPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   LongRect *aBounds,
   CBitMap *aBitMap, Boolean makePort);
```

Initialization function. Arguments are the same as in the CBitMapPane constructor. `IBitMapPane` may not be called if any arguments are specified in the constructor.

~CBitMapPane

```
~CBitMapPane ();
```

Destructor. Deletes the bitmap. Releases memory allocated to the bitmap and closes the `macPort` if one was opened.

Accessing

SetBitMap

```
void SetBitMap (CBitMap *aBitMap);
```

Stores the bounding rectangle of the bitmap in `aBitMap`. This rectangle describes the size and coordinate system of the bitmap.

GetBitMap

```
CBitMap *GetBitMap ();
```

Returns the bitmap associated with this pane.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the bitmap. Copies the bits in the bitmap to the bitmap of the current port.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CBufferedStream 26

Introduction

CBuffering Stream improves the performance of CStream by adding buffering. Buffering speeds up I/O by reducing the number of times the operating system is called and the number of times the disk is accessed for a given sequence of operations.

Heritage

Base Class	CStream
Derived Classes	CFileStream
	CHandleStream

Using CBufferedStream

Since CBufferedStream is an abstract class, you never use it directly. CBufferedStream does most of the work for the CHandleStream class and speeds up CFileStream I/O significantly. CBufferedStream implements the actual transfer of data to and from a stream.

Data Members

The following are protected data members that only derived classes may use.

Data member	Type	Description
`bufPosition`	`long`	Position in the stream of the first byte in the buffer.
`bufPos`	`long`	Current read/write position in the buffer.
`bufMax`	`long`	Size of the buffer.
`bufValid`	`long`	The highest valid byte in the buffer.
`itsBuffer`	`Handle`	Handle to the buffer.
`changed`	`Boolean`	`TRUE` if buffer has been put to since last written to disk.

Member Functions

Creation and destruction

CBufferedStream

```
CBufferedStream(long bufferSize =
   kTCLDefaultStreamBufferSize,
   Boolean check = TRUE);
```

Constructor. `bufferSize` specifies the number of bytes to be used for the buffer. `check` controls whether the stream checks for duplicate objects. The value of `kTCLDefaultStreamBufferSize` is 2048. See `CStream::CheckDuplicates`.

CBufferedStream

```
CBufferedStream();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IBufferedStream` for backward compatibility.

~CBufferedStream

```
~CBufferedStream();
```

Destructor. Frees the buffer memory.

SetBufferSize

```
void SetBufferSize(long aBufSize);
```

Sets the number of bytes to be used by the buffer. This function should be called before the stream is first opened.

Open and close

Open

```
void Open(TCLStreamMode mode);
```

Opens stream for `ReadStream`, `WriteStream`, or `ReadWriteStream` access.

Close

```
void Close ();
```

Closes stream, returning it to `ClosedStream` mode. The same stream may be opened and closed multiple times, such as for writing and then reading.

Positioning

Position

```
long Position ();
```

Returns the current stream position. The stream position is a long integer ranging from zero to the number of bytes in the stream minus one. The first byte of the next `Get` or `Put` will come from or go to the current position.

MoveTo

```
void MoveTo(long newPosition);
```

Changes the current stream position.

Size

```
long Size ();
```

Returns the number of bytes in the stream.

Truncate

```
void Truncate(long newSize);
```

Reduces the number of bytes in the stream.

Writing

Put

```
void Put(constvoid *bytes, long n);
```

Puts `n` bytes to the stream.

PutThru

```
void PutThru(constvoid *bytes, char c);
```

Puts data to the stream starting with the first character of bytes and ending with the first occurrence of the character `c`.

PutChar

```
void PutChar(char value);
```

Puts a single character to the stream.

Reading

Get

```
void Get(void *bytes, long n);
```

Gets `n` bytes from the stream.

GetThruN

```
long GetThru (void *bytes, char c, long n);
```

Gets data from the stream starting with the first character of `bytes` and ending with the first occurrence of the character `c`, the end of the stream or the first `n` characters, whichever is first.

`GetThruN`, unlike most `get` functions, does not fail if it is called at the end of a stream. Instead, it returns a length of zero. `GetStrAsLine`, which calls `GetThruN`, also works in this way.

GetChar

```
char GetChar();
```

Gets a single character from the stream.

Member Functions: Protected

Fill

```
void Fill ();
```

CBufferedStream calls this function when it reaches the end of the buffer while reading a stream. The derived class must refill the buffer or fail.

Drain

```
void Drain ();
```

CBufferedStream calls this function when it reaches the end of the buffer while writing a stream. The derived class must empty the buffer or fail.

Bracket

```
void Bracket(long aPosition);
```

CBufferedStream calls this function when it is asked to `Position` outside the data currently in the buffer. The derived class must drain the buffer, if necessary, and reposition it so that the requested position is inside the buffer. The resulting buffer may be empty.

GetPhysicalSize

```
long GetPhysicalSize();
```

CBufferedStream calls this function when it needs to know the actual physical size of the data in the stream, independent of buffering.

SetPhysicalSize

```
void SetPhysicalSize(long newSize);
```

CBufferedStream calls this function in response to `Truncate`. The derived class must reduce the actual physical size of the stream—for example, the size of a file—to the requested size.

CBureaucrat 27

Introduction

CBureaucrat is an abstract class that implements a link in the chain of command. Any object in the THINK Class Library that responds to a menu command or a mouse click is a bureaucrat.

A bureaucrat can either respond to a command, or it can pass the command to its supervisor. All of the default member functions for bureaucrats pass the command on to the supervisor. If a particular derived class cannot handle the command, therefore, calling the inherited member function causes it to be passed on up the chain of command.

Heritage

Base Class	CCollaborator
Derived Classes	CDirectorOwner
	CView
	CPopupMenu

Using CBureaucrat

You generally won't need to create a derived class of CBureaucrat because most of the objects you use are already descendants of this class. However, if you do need to create a class that is a link in the chain of command, CBureaucrat is the class to derive from.

The global variable `gGopher` points to the current bureaucrat, which is the first object in the chain of command. Whenever the switchboard responds to an event, it calls an appropriate member function of the gopher. For instance, in response to a key-down event, the switchboard calls the gopher's `DoKeyDown` member function.

If gGopher cannot "handle" the command, it passes the command up the chain of command until it reaches an object that does handle the command, or until it reaches the end of the chain of command—the application.

When no documents are open, gGopher points to the application. When you open a document, gGopher points to it. Your document derived class should use the BecomeGopher member function to change gGopher to point to the main pane.

Use the BecomeGopher member function to change the gopher instead of setting the gGopher variable yourself. BecomeGopher uses CCollaborator's provider/dependent mechanism to let dependent objects know that the gopher has changed.

Global Variables and Data Members

Global variable

CBureaucrat uses this global variable.

Variable	Type	Description
gGopher	CBureaucrat	The current bureaucrat

Data member

CBureaucrat defines only one data member.

Data member	Type	Description
itsSupervisor	CBureaucrat*	The supervisor

Member Functions

Creation and destruction

CBureaucrat

```
CBureaucrat (CBureaucrat *aSupervisor);
```

Constructor. Sets the bureaucrat's supervisor.

~CBureaucrat

```
~CBureaucrat();
```

Destructor. If this object is the gopher, the function changes the gopher to the object's supervisor.

IBureaucrat

```
void IBureaucrat (CBureaucrat *aSupervisor);
```

Initialization function provided for backward compatibility. `IBureaucrat` may be called only if no constructor argument is specified. The argument is the same as the constructor's.

Accessing

GetSupervisor

```
CBureaucrat *GetSupervisor();
```

Returns the bureaucrat's supervisor.

Command

The following are the commands that every bureaucrat recognizes. Unless a bureaucrat (or one of its ancestors) overrides one of these functions, it passes the command to its supervisor. If you follow the chain of supervisors—the chain of command—all the way to the end, you'll end up at your application class.

Notify

```
void Notify (CTask *theTask);
```

Calls the `Notify` function of the bureaucrat's supervisor to inform the supervisor that `theTask` has been completed.

NotifyClean

```
void NotifyClean (CTask *theTask);
```

In CBureaucrat, `NotifyClean` operates the same way as `Notify`. In CDocument, `NotifyClean` operates the same way as `Notify`, except that `NotifyClean` doesn't mark the document dirty.

IsTaskWindowInFront

```
Boolean IsTaskWindowInFront();
```

Returns `TRUE` if the current undo/redo task belongs to the frontmost window.

SetChanged

```
void SetChanged (Boolean isChanged);
```

Propagates the change notification up the chain of command. If there is no document in the chain of command, the notification is discarded.

GetChanged

```
Boolean GetChanged ();
```

Returns any changed state from the document in the chain of command.

DoKeyDown

```
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Calls the `DoKeyDown` function of the bureaucrat's supervisor to handle a key-down event.

DoAutoKey

```
void DoAutoKey (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Calls the `DoAutoKey` function of the bureaucrat's supervisor to handle an auto-key event.

DoKeyUp

```
void DoKeyUp (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Calls the `DoKeyUp` function of the bureaucrat's supervisor to handle a key-up event.

Note

By default, the Toolbox Event Manager masks out key-up events.

ToggleChanged

```
void ToggleChanged();
```

Toggles the flag that indicates that the document has changed. `ToggleChanged` is called when an Undo task is performed.

DoCommand

```
void DoCommand (long theCommand);
```

Calls the `DoCommand` function of the bureaucrat's supervisor to handle a command.

Dawdle

```
void Dawdle (long *maxSleep);
```

Performs periodic actions at idle time. The default function does nothing. CApplication's `Idle` function calls the `Dawdle` function of every bureaucrat in the chain of command.

If your bureaucrat requires periodic actions, perform them in the `Dawdle` function. Set the value of `maxSleep` to the largest number of ticks that your application can tolerate between calls to the `Dawdle` function. If your application's `Dawdle` function doesn't have any time constraints, you can ignore `maxSleep`.

The application class uses the value of `maxSleep` to let `WaitNextEvent` know that it should yield an event at least once every `maxSleep` ticks. For instance, the `Dawdle` function for CEditText calls `TEIdle` to blink the insertion point and sets `maxSleep` to `GetCaretTime`, which is the number of ticks between blinks.

Another use for the `Dawdle` function would be to update an on-screen clock. You would update the display from your document's `Dawdle` function. If your clock displays seconds, set `maxSleep` to 60, since you need to update the display at least every 60 ticks.

On a null event, CApplication's `Idle` function calls the `Dawdle` function of every bureaucrat in the chain of command. `Idle` sets the global variable `gSleepTime` to the smallest sleep time that all bureaucrats requested. The switchboard uses this global variable in its call to `WaitNextEvent`.

UpdateMenus

```
void UpdateMenus ();
```

Updates the menus that relate to this bureaucrat. Your bureaucrat (usually a document or a pane) should override this function to enable the menu items that pertain to it. When you write an `UpdateMenus` function, be sure to call inherited `UpdateMenus` before you enable your own menus. As a result, your supervisor's menus will be updated first. See Chapter 23, "CBartender," especially the section "Dimming and checking menu items," for a discussion of menu updating.

BecomeGopher

```
Boolean BecomeGopher (Boolean fBecoming);
```

If `fBecoming` is `TRUE`, make this bureaucrat the gopher and call `BroadcastChange` with `bureaucratIsGopher` as the reason. If `fBecoming` is `FALSE`, call `BroadcastChange` with `bureaucratIsNotGopher` as the reason.

If the current gopher refuses to relinquish control, this function returns `FALSE` and does not call `BroadcastChange` at all.

Change notification

The following member functions override functions in CCollaborator. To learn how to use them, see Chapter 33, "CCollaborator."

BroadcastChange

```
void BroadcastChange (long reason, void* info);
```

In addition to notifying this bureaucrat's dependents of a change, this function notifies its supervisor. `reason` is an integer that describes the type of change. You must define reasons for your derived class. `info` is a pointer to any additional information needed to respond to the change. If you want objects that aren't in a collaborator's list of dependents to know about a change, override this function.

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

One of this bureaucrat's providers has just changed. This function passes the change notification to the bureaucrat's supervisor. If you want your derived class to respond to change notifications, override this function. `aProvider` is the provider that changed. `reason` is an integer that describes the type of change. You must define reasons for your derived class. `info` is a pointer to any additional information that your derived class might need.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

27 CBureaucrat

CButton 28

Introduction

CButton implements a standard Macintosh push button. You can use a button in any pane or window.

Heritage

Base Class	CControl
Derived Classes	CCheckBox
	CRadioControl

Using CButton

CButton implements a standard Macintosh push button. You should not override this class unless you want to implement a push button that works differently from the default button. (The other kinds of Macintosh buttons, check boxes, and radio buttons are derived classes of this class.)

To create graphic elements that act like buttons but are not based on a Macintosh control, use CSwissArmyButton and its derived classes.

The CButton class handles all mouse tracking for you. When you click and release the mouse within a button, the button calls its supervisor's `DoCommand` function. To specify which command number the button passes to the `DoCommand` function, call its `SetClickCmd` function.

When you click a button, the `DoClick` member function that CButton inherits from CControl calls the Macintosh Toolbox routine `TrackControl` to highlight the button and track mouse movement. When you release the mouse within a button, the `DoClick` member function calls the control's `DoGoodClick` function. The `DoGoodClick` member function calls the `DoCommand` function of the button's supervisor.

Data Members

CButton defines these instance data members:

Data member	Type	Description
clickCmd	long	Command number to send after a click. The default is cmdNull.
procID	short	The procID used to create this button.

Member Functions

Although CButton implements only the member functions listed here, you can use the member functions it inherits from CControl and CView to manipulate a button's title and location.

Creation and destruction

CButton

```
CButton (short CNTLid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. CNTLid is the resource ID of the 'CNTL' resource. CButton places the control at the location specified in the 'CNTL' resource. If you want to position the button from your program, call its Place function (inherited from CPane). The last two arguments are the same as those used for CPane.

CButton

```
CButton (short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   StringPtr title, Boolean fVisible,
   short procID, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. The first four and last two arguments are the same as those used for CPane. title is the button title. fVisible controls the initial visibility of the button. procID specifies what type of button to create. Its possible values are defined in Controls.h and include pushButProc, checkBoxProc, and radioButProc.

Note

The other arguments are described in Chapter 71, "CPane."

CButton

```
CButton ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IButton` for backward compatibility.

IButton

```
void IButton (short CNTLid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialization function provided for backward compatibility. `IButton` may be called only if no constructor arguments are specified.

INewButton

```
void INewButton (short aWidth, short aHeight,
   short aHEncl, short aVEncl, StringPtr title,
   Boolean fVisible, short procID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialization function provided for backward compatibility. `INewButton` may be called only if no constructor arguments are specified.

Visual state and click response

GetClickCmd

```
long GetClickCmd ();
```

Returns the command that is sent to a button's supervisor when the user clicks a button.

SetClickCmd

```
void SetClickCmd (long aClickCmd);
```

Sets the command number to be sent when the user clicks a button.

SetDefault

```
void SetDefault (Boolean fDefault);
```

Specifies whether the button is the default button. The default button has a 3-pixel-thick rounded rectangle as a border.

SimulateClick

```
void SimulateClick ();
```

Simulates a click in this button. Use this function when you want to provide visual feedback for a Command key shortcut. This function highlights the button momentarily, and then calls the button's `DoGoodClick` function.

DoGoodClick

```
void DoGoodClick (short whichPart);
```

When you press and release the mouse within the button, this function calls its supervisor's `DoCommand` function.

Object I/O

PutTo

```
void PutTo; (CStream& Stream);
```

Writes the button to the stream.

GetFrom

```
void GetFrom; (CStream& Stream);
```

Reads the button from the stream.

Member Functions: Private

CButtonX

```
void CButtonX ();
```

Performs common initialization.

IButtonX

```
void IButtonX (short CNTLid);
```

Completes initialization from a `'CNTL'` resource.

INewButtonX

```
void INewButtonX (StringPtr title,
   Boolean fVisible, short procID);
```

Completes initialization from a `procID`.

CCharGrid 29

Introduction

CCharGrid is a derived class of CGridSelector that displays characters in a table and lets you choose one. CCharGrid is useful for implementing tool palettes like those in MacPaint™ and HyperCard™.

Heritage

Base Class	CGridSelector
Derived Classes	None

Using CCharGrid

The CCharGrid class lets you create panes that display characters in a table. The most common use for this kind of table is a tool palette. You can use a CCharGrid as a tool palette that's part of a window or as a custom tear-off menu. The Art Class demonstration program uses CCharGrid to display its Tools tear-off menu.

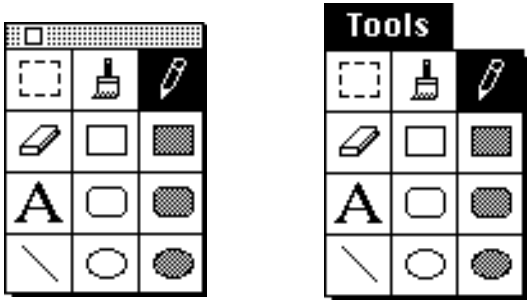


Figure 29-1 Art Class uses CCharGrid as a palette and as a menu

To use a CCharGrid, you create a special font that contains your tools. You can use ResEdit to create the font, or you can use a

commercial bitmap font editor. Apple recommends that the family ID for these kinds of fonts be in the range 32256 to 32767. Check the Font Manager chapters in *Inside Macintosh I* and *Inside Macintosh IV* for detailed information about working with fonts.

Unlike other derived classes of CPane, which give you the option of initializing an object from a resource, you *must* use a resource to initialize a CCharGrid. The first argument to the constructor, `CCharGrid`, is the resource ID of a `'ChGd'` resource. That resource contains the values that `CCharGrid` passes to CGridSelector's constructor.

CCharGrid defines this data member:

Data member	Type	Description
`theCharacters`	`Handle`	Handle to a Pascal string of the characters to display in the grid

Member Functions

Creation and destruction

CCharGrid

```
CCharGrid (short ChGdId, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. `ChGdId` is the resource ID of the `'ChGd'` resource that describes the location of the pane and the size and name of the font to use for the grid. See the section "Using CCharGrid" earlier in this chapter for details on `ChGd` resources. CCharGrid creates a text environment object of class CTextEnvirons to maintain the text characteristics whenever the characters are drawn. The text transfer mode is initially `srcOr`. The last two arguments are the same as those used with CPane.

CCharGrid

```
CCharGrid ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `ICharGrid` for backward compatibility.

~CCharGrid

```
~CCharGrid ();
```

Destructor. Deletes `theCharacters` handle.

ICharGrid

```
void ICharGrid (short ChGdid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialization function provided for backward compatibility. `ICharGrid` may be called only if no constructor arguments are specified.

Drawing

DrawItem

```
void DrawItem (short theItem, Rect *theBox);
```

Draws character numbered `theItem` so that it is centered within `theBox`. Characters are numbered starting at 1.

29 CCharGrid

CCheckBox 30

Introduction

CCheckBox implements a standard Macintosh check box. You can use a check box in any pane or window.

Heritage

Base Classes	CButton
	CGroupButton
Derived Classes	None

Using CCheckBox

You should not override this class unless you want to implement a check box that works differently from the default. To implement a graphic element with the behavior of a check box, see Chapter 111, "CSwissArmyButton."

The CCheckBox class handles all mouse tracking. When you click and release the mouse button within a check box, it toggles the value of the check box and calls its supervisor's `DoCommand` function. To specify which command number the check box calls in the `DoCommand` function, call its `SetClickCmd` function.

Note

CCheckBox inherits the `SetClickCmd` member function from CButton.

When you click a check box, the `DoClick` member function that CCheckBox inherits from CControl calls the Macintosh Toolbox routine `TrackControl`, highlights the check box, and tracks mouse movement. When you release the mouse within a check box, the `DoClick` function calls the control's `DoGoodClick` function. The

DoGoodClick function toggles the value of the control and calls the check box's supervisor's DoCommand function.

To find out whether the check box is checked, call its IsChecked function. You can set the value of the check box from your program by calling its SetValue function. Use the values BUTTON_ON and BUTTON_OFF to specify whether to check or uncheck the check box.

Note

Setting the value with SetValue will not call the check box's supervisor's DoCommand function.

A CCheckBox object can participate in a button group along with CRadioControl, CSwissArmyButton, and CIconButton objects. Setting the value of a CCheckBox control to 1 sets all radio-style buttons in the group to 0. Other check box-style buttons are unaffected. As a result, you can construct button groups that behave like the following button group:

Figure 30-1 Example of a button group

Data Members

This class has no data members.

Member Functions

Although CCheckBox implements only the functions listed here, you can use the functions it inherits from CControl and CView to manipulate a check box's title and location.

You should use the SetClickCmd function that CCheckBox inherits from CButton to specify the command that the check box sends to its supervisor when you click it.

Creation and destruction

CCheckBox

```
CCheckBox (short CNTLid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. `CNTLid` is the resource ID of the `'CNTL'` resource. CCheckBox places the control at the location specified in the `'CNTL'` resource. If you want to position the check box from your program, call its `Place` function (inherited from CPane). The last two arguments are the same as those used for CPane.

CCheckBox

```
CCheckBox (short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   StringPtr title, Boolean fVisible,
   short procID, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. The first four and last two arguments are the same as those used for CPane. `title` is the check box title. `fVisible` controls the initial visibility of the check box. `procID` specifies what type of check box to create. Its possible values are defined in `Controls.h` and include `pushButProc`, `checkBoxProc`, and `radioButProc`.

CCheckBox

```
CCheckBox ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `ICheckBox` for backward compatibility.

ICheckBox

```
void ICheckBox (short CNTLid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialization function provided for backward compatibility. `ICheckBox` may be called only if no constructor arguments are specified.

INewCheckBox

```
INewCheckBox (short aWidth,
   short aHeight, short aHEncl, short aVEncl,
   StringPtr title, Boolean fVisible,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialization function provided for backward compatibility. `INewCheckBox` may be called only if no constructor arguments are specified.

Mouse

DoGoodClick

```
void DoGoodClick (short whichPart);
```

When you press and release the mouse within the check box, this function toggles the state of the check box and calls its supervisor's `DoCommand` function.

Accessing

IsChecked

```
Boolean IsChecked ();
```

Returns `TRUE` if the check box is checked.

TurnOff

```
void TurnOff ();
```

Called by `CGroupButton::TurningOn` when a radio button in the same button group is turning on. `TurnOff` calls `SetValue(0)`.

SetValue

```
void SetValue (short aValue);
```

Sets the value of the button to `aValue`, which is one of `BUTTON_OFF`, `BUTTON_ON`. This function will also notify other CGroupButton controls in the same button group if the check box is being turned on.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CChore 31

Introduction

CChore is an abstract class for implementing periodic or urgent actions. The THINK Class Library executes periodic chores at idle time and urgent chores after processing the current event.

Heritage

Base Class	None
Derived Classes	CMBarChore
	CTearChore

Using CChore

Use this class to create chores that run at idle time or at the next possible opportunity. Your application maintains a list of chores and implements the member functions to install and remove them.

Using chores

Chores are attached to the application and not to any particular document. The tasks that a chore performs, then, should pertain to the entire application.

To implement a chore, define a derived class of this class. In your derived class, override the `Perform` member function. That function implements the action you want your chore to take.

Note

If you need your chore to run at a specific interval, use the Macintosh Toolbox routine `Ticks` to store the last time the chore was performed.

For periodic tasks that apply to a particular document or pane, use the `Dawdle` member function. All bureaucrats, including documents and panes, inherit a `Dawdle` function. The application calls the `Dawdle` function of each bureaucrat in the chain of command. For example, you would use a `Dawdle` function to blink an insertion point in a pane.

Idle chores

To install an idle chore, call the application's `AssignIdleChore` function. Your application calls the `Perform` function of all idle chores at idle time. To remove an idle chore, call the chore you want to remove within the application's `CancelIdleChore` function. This example shows how to install an idle chore:

```
gApplication->AssignIdleChore(theChore);
```

Note

To remove an idle chore, keep it in a data member, because `CancelIdleChore` requires a reference to a CChore object.

Urgent chores

An urgent chore is one that gets executed immediately after the application processes the current event. Typically, it's the same event that caused the urgent chore to be installed. Urgent chores are executed only once. They're removed from the urgent chore list immediately after they're executed. To install an urgent chore, call the application's `AssignUrgentChore` function.

Data Members

This class has no data members. Your CChore derived class may define its own data members. For example, if you want an idle chore to run at specified intervals, use a data member to store the time period.

Member Functions

CChore implements only one member function, which you must override. Your CChore derived class may implement additional member functions.

Perform

```
void Perform (long *maxSleep);
```

This is the function that executes your chore. You must override this function in your CChore derived class. The default function does nothing.

The `maxSleep` parameter specifies how many ticks your chore can tolerate passing before it gets a `Perform` function call. For example, if you need to do a chore at least once every tenth of a second, you would set `maxSleep` to 6. (Each tick is 1/60 of a second.)

For a more detailed discussion of `maxSleep`, see the description of the `Dawdle` member function “CBureaucrat” in Chapter 27.

31 CChore

CClipboard 32

Introduction

CClipboard implements a standard Macintosh clipboard, or scrap. The default clipboard handles `TEXT` and `PICT` scraps. To deal with other kinds of scraps or to implement a private scrap, create a derived class of this class.

Heritage

Base Class	CDirector
Derived Classes	CStyleTEClipboard

Using CClipboard

This class implements a standard Macintosh clipboard. CClipboard uses the desk scrap to store its data and supports a window to display the contents of the scrap. This implementation supports only `TEXT` and `PICT` data.

When you initialize your application, its constructor calls the `MakeClipboard` function of your application class. That function creates an instance of CClipboard and stores it in the global variable `gClipboard`.

The default application `DoCommand` member function handles the `cmdToggleClip` command, which shows and hides the Clipboard window. When you're running under System 7 or MultiFinder, the Clipboard window is hidden when your application is suspended. When the application resumes, the Clipboard window becomes visible again.

Note

In the THINK Class Library, desk accessories behave as if they were in their own layer, even if your application isn't running under System 7 or MultiFinder. When a desk accessory window is frontmost, the Clipboard window is hidden. When one of the application windows is frontmost, the Clipboard window becomes visible again.

Implementing a CClipboard derived class

If you want your application to support a private scrap, or if you want to display other types of data, you need to derive a class from CClipboard. You also need to override the `MakeClipboard` member function in your application class.

Note

If you create a derived class of this class, be sure you understand how the Macintosh scrap mechanism works. See *Inside Macintosh: More Macintosh Toolbox*, Chapter 2, "The Scrap Manager."

In your CClipboard-derived class, you need to override these member functions:

Member Function	Action
`PutData`	Place data in the private scrap.
`GetData`	Retrieve data from the private scrap.
`ConvertGlobal`	Place the contents of the desk scrap into the private scrap.
`ConvertPrivate`	Place the contents of the private scrap into the desk scrap.
`MakeClipView`	Create a view to display data other than plain text of pictures.

Table 32-1 CClipboard member functions to override

Global Variables and Data Members

Global variables

The global `gClipboard` contains a pointer to the single instance of the clipboard object. The value of this variable is set in the application member function `MakeClipboard`.

Data member	Type	Description
`gClipboard`	`CClipboard`	The global clipboard

Data members

CClipboard defines these data members:

Data member	Type	Description
`itsContents`	`CPanorama*`	Pane for displaying contents
`itsScrollPane`	`CScrollPane*`	Contents can be scrolled
`theLength`	`long`	Length from last get operation
`theOffset`	`long`	Offset from last get operation
`lastScrapCount`	`short`	Count at the last conversion between global and private scraps
`privateNewer`	`Boolean`	`TRUE` if the private scrap has changed since the last conversion to the desk scrap
`windowVisible`	`Boolean`	`TRUE` if the Clipboard window is visible

Member Functions

Creation and destruction

CClipboard

```
CClipboard (Boolean hasWindow = TRUE);
```

Constructor. `hasWindow` controls whether the Clipboard displays its contents in a window. The application's `MakeClipboard` member function creates the clipboard and stores a pointer to it in the global `gClipboard`.

Note

If you derive from CClipboard, be sure to override the `MakeClipboard` function in your application class so it creates an object of the derived clipboard derived class. See the implementation of `MakeClipboard` in CApplication for an example.

IClipboard

```
void IClipboard (CApplication *aSupervisor,
   Boolean hasWindow);
```

Initialization function provided for compatibility with previous release. `IClipboard` may only be called if no constructor arguments are specified.

Suspend and resume

The following functions handle scrap conversion when your application moves from foreground to background under System 7 or MultiFinder. In most cases, you won't need to use or override these functions.

Suspend

```
void Suspend ();
```

The application is about to be switched into the background under System 7 or MultiFinder. If the application's private scrap is newer than the desk scrap, this function calls the clipboard's `ConvertPrivate` and `ScrapConverted` functions, then calls the clipboard window's `HideSuspend` function.

Resume

```
void Resume ();
```

The application is about to be brought to the foreground under System 7 or MultiFinder. If the desk scrap is newer than the private scrap, this member function calls the clipboard's `ConvertGlobal`, `ScrapConverted`, and `UpdateDisplay` functions to update the contents of the clipboard. Then it calls the window's `ShowResume` function to make the window visible.

Appearance

The following functions handle the appearance of the clipboard window and how the contents of the clipboard appear in the

window. If you implement your own clipboard class, you'll need to override the `UpdateDisplay` function.

Close

```
Boolean Close (Boolean quitting);
```

The user chose **Close** from the **File** menu. `Close` calls the `CloseWind` function and returns `TRUE`.

CloseWind

```
void CloseWind (CWindow *theWindow);
```

The user clicked in the window's close box. This function hides the Clipboard window and changes the text in the menu item.

Toggle

```
void Toggle ();
```

This function opens the Clipboard window if it's closed or closes it if it's open. The application's default `DoCommand` function calls this function.

UpdateDisplay

```
void UpdateDisplay ();
```

Displays the contents of the scrap in the Clipboard window. This function supports only `TEXT` and `PICT` types of data. It works correctly with a private scrap, but if you want to display other kinds of data, you need to override this function.

Accessing

Uses the following functions to place data in and retrieve data from the desk scrap. If your application implements a private scrap, use the `PutData` and `GetData` member functions instead.

PutGlobalScrap

```
void PutGlobalScrap (ResType theType,
   Handle theData);
```

Puts `theData` of type `theType` into the desk scrap. Be sure to call the `EmptyGlobalScrap` function before you call this function.

GetGlobalScrap

```
Boolean GetGlobalScrap (ResType theType,
   Handle theData);
```

Gets data of type `theType` from the desk scrap and puts it in the block to which `theData` is a handle. Returns `TRUE` if it was able to fulfill the request, `FALSE` otherwise.

`theData` must be an allocated handle. The size of the allocated memory grows as needed to fit the data.You get `TEXT` data from the desk scrap in the following manner:

```
Handle myData;
…
   /* Create a zero-sized handle */
myData = NewHandle(0);
gClipboard->GetGlobalData('TEXT', myData);
```

DataSize

```
long DataSize (ResType theType);
```

Returns the number of bytes of data in the Clipboard of type `theType`. If the Clipboard doesn't contain any data of the specified type, this function returns zero.

Status

```
ScrapStatus Status ();
```

Returns the status of the scrap. This function returns a value that describes the relationship between the private scrap and the desk scrap.

Value	Meaning
`PRIVATE_SCRAP_NEWER`	The information in the private scrap is newer than the information in the desk scrap.
`GLOBAL_SCRAP_NEWER`	The information in the desk scrap is newer than the information in the private scrap.
`SCRAPS_THE_SAME`	The information in both scraps is the same.

Table 32-2 Scrap status values

ScrapConverted

```
void ScrapConverted ();
```

Sets the internal flags after the scrap has been converted.

Scrap conversion

Uses the `PutData` and `GetData` functions to implement **Cut**, **Copy**, and **Paste** commands for your application. If your application uses only the desk scrap, you can use the `PutGlobalScrap` and `GetGlobalScrap` functions.

PutData

```
void PutData (ResType theType, Handle theData);
```

Put `theData` of type `theType` into the scrap. The default function puts the data in the desk scrap. If your derived class supports a private scrap, you must override this function. After you store your data in the private scrap, call the `PrivateChanged` function. Before you call this function, be sure to call an `EmptyScrap` function to clear your scrap.

GetData

```
Boolean GetData (ResType theType,
   Handle *theData);
```

Gets `theData` of type `theType` and puts it into a newly created handle. This function returns `TRUE` if the request was successful, `FALSE` otherwise. If your application supports a private scrap, you must override this function.

Unlike with `GetGlobalScrap`, you should not allocate the handle to `theData`. This function will allocate the memory. You get data of type `'TEXT'` in this way:

```
Handle myData;
…
gClipboard->GetData('TEXT', &myData);
```

ConvertGlobal

```
void ConvertGlobal ();
```

Converts data in the desk scrap and puts it in the private scrap. The default function does nothing. If your application supports a private scrap, you must override this function.

ConvertPrivate

```
void ConvertPrivate ();
```

Converts data in the private scrap and puts it into the desk scrap. The default function does nothing. If your application supports a private scrap, you must override this function.

EmptyGlobalScrap

```
void EmptyGlobalScrap ()
```

Clears the contents of the desk scrap.

EmptyScrap

```
void EmptyScrap ()
```

Clears the contents of the scrap that is the destination of `PutData`. You should call this function before you call `PutData`. The default function clears the global scrap.

PrivateChanged

```
void PrivateChanged ();
```

The data in the private scrap has changed. If you implement a private scrap, be sure to call this function of the clipboard object after you put new data in your private scrap. This function sets the internal flags and calls the clipboard's `UpdateDisplay` function. Your derived class should not override this function.

MakeClipView

```
CPanorama *MakeClipView (long dataType,
   Handle dataHandle)
```

Makes a view to display the Clipboard data `dataHandle` of type `dataType`. This function disposes `dataHandle` when appropriate. `UpdateDisplay` calls this function to create the right kind of pane for the data you want to display in the Clipboard window.

If you create a derived class of CClipboard, override this function to create the kind of pane needed to display your data. If `dataType` is `PICT` or `TEXT`, you can call the inherited function.

Member Functions: Protected

IClipboardX

```
voidIClipboardX (CApplication *aSupervisor,
   Boolean hasWindow);
```

Performs common construction.

Class Resources

The `'WIND'` resource for the Clipboard window is in the file `TCL Resources`, which contains all the resources the THINK Class Library requires.

Resource	Description
`'WIND' 200`	Window template for Clipboard window

Apple Event Support

For information regarding Apple event member functions, please refer to the supplemental information provided in the file `Apple event info` in the folder `Apple Event Classes`.

32 CClipboard

CCollaborator 33

Introduction

CCollaborator is an abstract class that implements objects that depend on one another. This class provides a mechanism for an object to announce changes to other objects.

Heritage

Base class	None
Derived classes	CBureaucrat
	CCollection

Using CCollaborator

This class implements a mechanism that lets an object announce changes to other objects. The object that announces the changes is called the provider. The objects that depend upon announcements from the provider are called dependents.

For example, suppose that your program displays the contents of a file as different kinds of graphs in different windows. Whenever the data in the file changes, the windows that display the data need to change also. The data file is the provider. The graph display windows are the dependents.

To specify a dependency, call `DependUpon` for the dependent object with the provider as the argument. When the provider changes, call the provider's `BroadcastChange` function. That member function calls each of the dependents' `ProviderChanged` functions.

The `BroadcastChange` function takes a parameter called `reason` that lets the dependents know why they're being notified. Reasons are implemented as long integers. You can use `const` values, `#define` directives, or `enum` types.

Reason names follow three conventions. First, reason names begin with the name of their class, without the C. For example,the names of CArray's reasons are entered as “array”. Second, the reason codes for each THINK Class Library class begin with a unique four-character `OSType` value, for example, '`TAB0`' for the CTable class. Additional reason codes in the same class are formed by adding 1 to the preceding code, such as '`TAB0`', '`TAB1`', etc. Third, each collaborator subclass has a reason named `classnameLastChange`. For example, CArray defines a reason named `arrayLastChange`.

Using these conventions, you can ensure that the reason codes you define are unique. You can define your own four-character `OSTypes` as reason codes. Reason codes consisting entirely of uppercase letters and numbers are reserved for the THINK Class Library. Your own reason codes should contain at least one lowercase letter or special character, such as `Myn0`, to ensure that all reason codes are globally unique. The reason codes currently in use by the THINK Class Library are:

Reason	Class
ABS0-1	CAbstractText
ARR0-1	CArray
BUR0-1	CBureaucrat
CON0	CControl
DIA0	CDialogText
PIC0	CPICTGrid
POP0	CPopupMenu
RAD0	CRadioControl
RUN0-1	CRunArray
TAB0	CTable

Note that the fourth character of all codes is the digit '0'.

Alternatively, in each of your derived classes you can set the derived class's first reason code to be equal to its base class's `LastChange` plus 1. For example, if you derive a class CMyArray from CArray, you might define reason codes in CMyArray as:

```
enum
{
   myarraySwapElements = arrayLastChange +
1,
   myarrayScrambled,
   myarrayLastChange = myArrayScrambled
};
```

This technique ensures that your reason codes are unique in a particular branch of the class hierarchy, but not unique overall. With multiple inheritance, a class can inherit reason codes from multiple branches of the class hierarchy. Thus, this technique does not, in general, guarantee unique reason codes even with respect to a single object.

Either approach, using `OSType` or `LastChange` values, is compatible with previous releases of the THINK Class Library.

The following examples illustrate how you might implement the graphing program described earlier in this chapter. When the data in the data file changes, the data file's `UpdateData` member function calls the `BroadcastChange` function with the reason `dataUpdated`. The `BroadcastChange` function calls a `ProviderChanged` function on each of the data file's graphs, the data file's dependents. This updates the picture of the data in each graph.

This is what the implementation might look like:

```
enum
{  myFileUpdated = MyF0,
   myFilePointAdded,
   myFilePointDeleted,

   myFileLastChange = myFilePointDeleted
};
. . .
```

```
void CMyFile::UpdateData (
   MyDataPtr oldData,
   MyDataPtr newData)
{
   . . .
   BroadcastChange(myFileUpdated,
                   (void *) newData );
}

void CMyGraphPane::ProviderChanged (
   CCollaborator *aProvider,
   long reason, void *info)
{
   if (aProvider == itsDataFile)
      switch (reason)
      {
         case myFileUpdated:
            UpdateGraphData(info);
            break;
         case myFilePointAdded:
            AddGraphData(info);
            break;
         case myFilePointDeleted:
            DeleteGraphData(info);
            break;
      }
}
```

Usually, you should check that the provider is a known object before acting on a reason code. If you use unique reason codes, checking the provider for class membership is never necessary, as any one class can broadcast the code.

Data Members

CCollaborator defines the following protected data members:

Data Member	Type	Description
`itsProviders`	`CCollaboratorList*`	The objects this collaborator depends upon
`itsDependents`	`CCollaboratorList*`	The objects that depend upon this collaborator

Member Functions

Creation and destruction

CCollaborator

```
CCollaborator();
```

Constructor. This function sets both `itsProviders` and `itsDependents` to `NULL`.

~CCollaborator

```
~CCollaborator();
```

Destructor. Remove this object from each of its provider's list of dependents and from each of its dependent's list of providers. Then dispose of its own lists of providers and dependents.

Copy

```
void *Copy ();
```

Provided for backward compatibility.

ICollaborator

```
void ICollaborator ();
```

Initialization function provided for compatibility with previous releases. Does nothing.

Dispose

```
void Dispose ();
```

Provided for compatibility with previous release. Calls `operator delete`. `TCL_USE_DISPOSE` must be defined to use this member function.

Creating dependency

DependUpon

```
void DependUpon (CCollaborator *aProvider);
```

This collaborator depends on `aProvider`. Add this collaborator to `aProvider`'s list of dependents, and add `aProvider` to this collaborator's list of providers. If either of those lists is `NULL`, this function creates them before adding to them.

CancelDependency

```
void CancelDependency (CCollaborator
   *aProvider);
```

This collaborator no longer depends upon `aProvider`. Remove this collaborator from the provider's dependent list and remove the provider from this collaborator's provider list.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

Change notification

BroadcastChange

```
void BroadcastChange (long reason, void* info);
```

Notifies this object's dependents that this object has changed. `reason` describes the type of change. You must define reasons for your derived class. `info` is a pointer to any additional information needed to respond to the change.

If you want objects that aren't in a collaborator's list of dependents to know about a change, override this protected member function. For example, CBureaucrat overrides this function to notify the bureaucrat's supervisor, in addition to its dependents, of the change.

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void *info);
```

This informs you that one of this object's providers has just changed. Your derived class must override this protected member function to respond to each reason it can handle. The default function does nothing. `aProvider` is the provider that changed. `reason` describes the type of change. You must define reasons for your derived class. `info` is a pointer to any additional information that your derived class might need.

Dependent_Update

```
static void Dependent_Update
   (CCollaborator *aDependent, long aProvider);
```

Internal function. Called by `BroadcastChange`.

Member Functions: Private

CCollaborator uses the following member functions internally to maintain the lists of dependents and providers, and they may be made private or protected members. Whenever possible, use the `DependUpon` member function instead of these. When a dependent is added or removed, `DependUpon` updates both the dependent's list of providers and the provider's list of dependents. Similarly, it updates both lists when a provider is added or removed. If you use the functions below, you might damage the collaborators' lists so that they don't match. If these functions are made private, they will not be accessible.

AddDependent

```
void AddDependent (CCollaborator *aDependent);
```

Adds `aDependent` to this collaborator's list of dependents.

RemoveDependent

```
void RemoveDependent (CCollaborator *aDependent);
```

Removes `aDependent` from this collaborator's list of dependents.

AddProvider

```
void AddProvider (CCollaborator *aProvider);
```

Adds `aProvider` to this collaborator's list of providers.

RemoveProvider

```
void RemoveProvider (CCollaborator *aProvider);
```

Removes `aProvider` from this collaborator's list of providers..

Provider_Remove

```
static void Provider_Remove
   (CCollaborator *aProvider, long aDependent);
```

This internal function removes dependencies.

Dependent_Remove

```
static void Dependent_Remove
   (CCollaborator *aDependent, long infoAddr);
```

This internal function removes dependencies.

Friend Functions

The following friend functions overload `operators` << and >>.

operator <<

```
CStream & operator <<
   (CStream& s, CCollaborator* p);
```

Puts a CCollaborator object.

operator >>

```
CStream & operator >>
   (CStream& s, CCollaborator* p);
```

Gets a CCollaborator object.

CCollection 34

Introduction

CCollection is an abstract class for implementing collections.

Heritage

Base Class	CCollaborator
Derived Classes	CArray

Using CCollection

This class doesn't provide the implementation of the collection. In your derived class, you specify the data structures that implement the list.

Data Members

The only data member in this abstract class is the number of items in the collection. Your derived class needs to add at least a data member that contains or points to the memory which holds the items in your collection.

Data member	Type	Description
`numItems`	`long`	The number of items in the collection

Member Functions

Creation and Destruction

CCollection
```
CCollection();
```
Constructor. Sets the initial number of items to 0. Also, implicitly called when an object is created by `new_by_name`. Can also be used in combination with `ICollection` for backward compatibility.

~CCollection
```
~CCollection ();
```
Destructor.

Copy
```
void *Copy ();
```
Provided for backward compatibility.

ICollection
```
ICollection ();
```
Initialization function included for backward compatibility. Should not be called if arguments are passed to the constructor.

Accessing

GetNumItems
```
long GetNumItems ();
```
Returns the number of items in a collection.

IsEmpty
```
Boolean IsEmpty ();
```
Returns `TRUE` if the collection has no items.

Object I/O

PutTo
```
void PutTo (CStream& aStream);
```
Writes to the stream.

GetFrom
```
void GetFrom (CStream& aStream);
```
Reads from the stream.

CColorTextEnvirons 35

Introduction

CColorTextEnvirons extends CTextEnvirons by remembering the foreground and background color, the pen mode, height and width, and the foreground and background pattern of an object.

Heritage

Base Class	CTextEnvirons
Derived Classes	None

Using CColorTextEnvirons

Use CColorTextEnvirons to control more completely the drawing environment for panes. Consistent use of CColorTextEnvirons ensures that drawing styles do not “bleed” from one object to the next, and it eliminates the need to set up and restore the drawing environment in your own code.

Data Members

Data member	Type	Description
penSize	Point	The pen height and width.
penMode	short	The pen drawing mode.
forePatID	short	The foreground pattern 'PAT#' resource ID.
backPatID	short	The background pattern 'PAT#' resource ID.
forePatInd	short	The foreground pattern index. If non-zero, the foreground pattern is obtained from the resource. If zero, the black pattern is used.
backPatInd	short	The background pattern index. If non-zero, the background pattern is obtained from the resource. If zero, the black pattern is used.
changeForePattern	Boolean	TRUE if foreground pattern has been changed.
changeBackPattern	Boolean	TRUE if background pattern has been changed.
foreColor	RGBColor	The foreground color.
backColor	RGBColor	The background color.
changeForeColor	Boolean	TRUE if restore foreground color from foreColor. FALSE if set foreground color to black.
changeBackColor	Boolean	TRUE if restore background color from backColor. FALSE if set background color to white.

Member Functions

Creation and destruction

IColorTextEnvirons

```
IColorTextEnvirons();
```

Initialization function. Calls the `ITextEnvirons` function of CTextEnvirons. Sets `forePatID`, `backPatID`, `forePatInd`, and `backPatInd` to `0`. Sets `changeForePattern`, `changeBackPattern`, `changeForeColor`, and `changeBackColor` to `FALSE`. Sets `penSize.v` and `penSize.h` to `1`, and `penMode` to `patCopy`. Sets `foreColor` to pure black, and `backColor` to pure white.

Restore and capture

Restore

```
void Restore ();
```

Restores the environment. Called by the THINK Class Library before the pane's `Draw` member function is called.

Capture

```
void Capture ();
```

Replaces the contents of the environment with the settings from the current grafport. This function is similar to the eyedropper tool of a paint program.

CaptureText

```
static void CaptureText (TextInfoRec *rec);
```

Fills in the `TextInfoRec` argument with the text settings from the current grafport. This is a static member function.

Access

SetColorInfo

```
void SetColorInfo (RGBColor *fore,
   RGBColor *back);
```

Sets the foreground and background color in the environment. If either pointer is `NULL`, the corresponding Boolean, `changeForeColor` or `changeBackColor` is set to `FALSE`.

GetColorInfo

```
void GetColorInfo (Boolean *changeFore,
   Boolean *changeBack,
   RGBColor *fore, RGBColor *back);
```

Gets the foreground and background color settings from the environment.

SetForePatRes

```
void SetForePatRes (short resID, short index);
```

Sets the resource ID and index of a pattern from a `'PAT#'` resource to be used as the foreground pattern. If the index is `0`, `changeForePat` is set to `FALSE`.

SetBackPatRes

```
void SetBackPatRes (short resID, short index);
```

Sets the resource ID and index of a pattern from a `'PAT#'` resource to be used as the background pattern. If the index is `0`, `changeBackPat` is set to `FALSE`.

GetPatInfo

```
void GetPatInfo (short *foreID,
   short *foreIndex,
   short *backID, short *backIndex);
```

Gets the foreground and background resource ID and index from the environment.

SetPenInfo

```
void SetPenInfo (short penMode,
   short penHeight, short penWidth);
```

Sets the pen mode, height, and width in the environment.

GetPenInfo

```
void GetPenInfo (short *penMode,
   short *penHeight, short *penWidth);
```

Gets the pen mode, height, and width from the environment.

GetChangeBack

```
void GetChangeBack ();
```

Returns `changeBackColor`.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CControl 36

Introduction

CControl is an abstract base class for implementing Macintosh controls. The two built-in derived classes, CButton and CScrollBar, implement the standard Macintosh controls.

Heritage

Base Class	CPane
Derived Classes	CButton
	CScrollBar

Using CControl

This class describes the common functions for all controls. Scroll bars, buttons, check boxes, and radio buttons are derived from CControl. Although you won't be creating derived classes of CControl (unless you define your own kind of control), you should become familiar with these functions.

Data Members

The only data member for this class is a handle to the actual Macintosh control.

Data member	Type	Description
`macControl`	`ControlHandle`	Handle to the Macintosh control

Member Functions

You can use these member functions with any kind of control, but some functions don't really apply to all controls. For example, it's possible to set the value of a push button or to give a title to a scroll bar, but in most cases these actions don't make sense.

Creation and destruction

CControl

```
CControl(CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth = 0, short aHeight = 0,
   short aHEncl = 0, short aVEncl = 0,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY);
```

Constructor. Sets `macControl` to `NULL`. Most initialization is done by classes derived from CControl.

CControl

```
CControl ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`.

~CControl

```
~CControl();
```

Destructor. Disposes `macControl`.

Accessing

SetValue

```
void SetValue (short aValue);
```

Sets the value of a control. This member function calls the `BroadcastChange` function of all of this object's dependents, with `controlValueChanged` as the reason.

GetValue

```
short GetValue ();
```

Gets the value of a control.

SetMaxValue

```
void SetMaxValue (short aMaxValue);
```

Sets the maximum value for a control. For push buttons, check boxes, and radio buttons, this value should be 1.

GetMaxValue

```
short GetMaxValue ();
```

Gets the maximum value a control can take on.

SetMinValue

```
void SetMinValue (short aMinValue);
```

Sets the minimum value for a control. Note that for push buttons, check boxes, and radio buttons, this value should be 0.

GetMinValue

```
short GetMinValue ();
```

Gets the minimum value a control can take on.

SetTitle

```
void SetTitle (Str255 aTitle);
```

Sets the title of a control. Scroll bars can have titles, but the title is not displayed.

GetTitle

```
void GetTitle (Str255 aTitle);
```

Gets the title of a control.

SetActionProc

```
void SetActionProc (ActionProcPtr anActionProc);
```

Sets the action proc of a control. The action proc is called repeatedly while the mouse is held down in a control. The control manager distinguishes between an action proc called when the mouse goes down in a moving indicator (like a scroll box) and one called when the mouse goes down in a stationary part of a control. The default `DoClick` member function uses the action proc only for mouse hits that are not in a moving indicator. In the THINK Class Library, you must declare the action proc as follows:

```
void pascal MyAction (
   ControlHandle macControl,
   short whichPart);
```

If you want an action proc called when the mouse is in a moving indicator, you'll need to override the `DoClick` function.

Note

Be sure you understand action procs and the control manager before you use this function.

Appearance

The following functions affect the appearance of controls on the screen. They hide and show the control, draw it, move it, and change its size.

Hide

```
void Hide ();
```

Hides the control.

Show

```
void Show ();
```

Shows the control.

Activate

```
void Activate ();
```

Makes the control active. The control's `Activate` function is called automatically when its enclosure's `Activate` function is called. In CControl, this function calls `Refresh` so that the control will be redrawn as active on the next Update event.

Deactivate

```
void Deactivate ();
```

Makes the control inactive. The control's `Deactivate` function is called automatically when its enclosure's `Deactivate` function is called. In CControl, this function calls `DrawAll` to immediately redraw the control as inactive.

Offset

```
void Offset (long hOffset, long vOffset,
   Boolean redraw);
```

Moves the control by `hOffset`, `vOffset` pixels. If `redraw` is `TRUE`, redraws the control after the move.

ChangeSize

```
void ChangeSize (Rect *delta, Boolean redraw);
```

Changes the size of a control. Each component of the rectangle specifies the number of pixels by which to offset each point. Positive numbers indicate down and to the right. Negative numbers indicate up and to the left. If `redraw` is `TRUE`, the function redraws the control after the change.

GetHelpBalloonState

```
short GetHelpBalloonState ();
```

Returns one of the balloon help states:

kHMEnabledItem (0)	Enabled (or, for buttons, off)
kHMDisabledItem (1)	Disabled (dimmed)
kHMCheckedItem (2)	Enabled and checked (or, for buttons, on)
kHMMarkedItem (3)	Enabled and marked (for any other purpose)

Controls typically implement only the first two or three of the above states.

There is not necessarily a one-to-one correspondence between balloon help states and THINK Class Library variable states. The dimmed state is the most complex. Dimmed means that the view would normally accept user interaction but currently does not. Note that a view must normally want clicks or allow editing in order to be considered dimmed when it does not want clicks or allow editing. Classes for which dimmed or marked states are meaningful, or that want to use the marked state, must override this function.

Draw

```
void Draw (Rect *area);
```

Draws the control. The `area` parameter is ignored.

DrawAll

```
void DrawAll (Rect *area);
```

Draws the control and all its subviews.

Prepare

```
void Prepare ();
```

Prepares the port and the coordinate system before drawing. Generally, you don't need to use or override this function.

PrepareToPrint

```
void PrepareToPrint ();
```

Sets up the coordinate system and the clipping region for printing. The origin is set to (0,0). The clipping region is set as specified by the `printClip` data member, inherited from CPane, whose possible values are `clipAPERTURE`, `clipFRAME`, and `clipPAGE`.

RefreshLongRect

```
void RefreshLongRect (LongRect *area);
```

Adds the intersection of `area` and the pane's aperture to the update region of the pane. Redrawing does not actually take place until the next update event is processed.

Click response

The CControl class takes care of all the mouse tracking within a control.

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handles a click in a control. `hitPt` is the point in frame coordinates. `modifierKeys` is the same as the modifier field of an event record. The argument `when` is the time in ticks that the mouse went down.

The default function handles both simple controls and controls with moving indicators. It behaves a little differently in each case.

If the mouse goes down in a part other than a moving indicator:

- The default function calls the Toolbox routine `TrackControl` with the action proc you specified with the `SetActionProc` function.
- If `TrackControl` returns `TRUE` (the user clicks and releases the mouse in the same part of the control), this function calls the control's `DoGoodClick` function.

If the mouse goes down in a moving indicator, such as the thumb of a scroll bar:

- The default function calls the Toolbox routine `TrackControl` with no action proc.

- If the value of the control has changed (the user moved the indicator to a new place), this function calls the control's `DoThumbDragged` function.

DoThumbDragged

```
void DoThumbDragged (short delta);
```

Called when an indicator in a control has been moved. The `DoClick` function calls the control's `DoThumbDragged` function when the user changes the position of a control's indicator.

The default function does nothing. Controls with indicators should override this function. For an example, see Chapter 95, "CScrollBar."

DoGoodClick

```
void DoGoodClick (short whichPart);
```

Called when the mouse went down and up in the same part of a control.

The default function does nothing. Derived classes should override this member function. See Chapter 28, "CButton," for an example.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetControl

```
void GetControl (CStream& aStream,
   short controlDef);
```

Reads and initializes a CControl from the stream. Derived classes use this function to implement their `GetFrom` functions.

CCountingStream 37

Introduction

CCountingStream is a stream class whose instances—counting streams—count the number of bytes put to them, but don't actually write anything. If the same sequence of Put operations is done to a handle or file stream, the number of bytes written will be the same as determined by the counting stream.

Get operations have no effect.

Heritage

Base Class	CStream
Derived Classes	None

Using CCountingStream

Counting streams are used to preflight stream output operations to determine the number of bytes that will be written. Before performing a sequence of Put operations to a handle stream or a file stream, perform the same sequence to a counting stream, then call the counting stream's member function `Size` to determine the number of contiguous bytes the output will occupy.

Data Members

Although these data members are public, they should not be directly manipulated.

Data member	Type	Description
`count`	`long`	The number of bytes put to the stream
`pos`	`long`	The current position in the stream

Member Functions

Creation and destruction

CCountingStream

```
CCountingStream (Boolean check = TRUE);
```

Constructor. Calls the CStream constructor with argument `check`, then initializes `count` and `pos` to `0`. `check` controls whether the stream checks for duplicate objects. See the description of `CheckDuplicates` in Chapter 104, "CStream."

Open

Open

```
void Open (TCLStreamMode mode);
```

Opens stream for writing. `mode` must equal `WriteStream`.

Positioning

Position

```
long Position ();
```

Returns `pos`, the current stream position. The stream position is a long integer ranging from zero to the number of bytes in the stream minus one. The first byte of the next Put operation will go to the current position.

MoveTo

```
void MoveTo (long newPosition);
```

Changes the current stream position. Sets `pos` to `newPosition`, which must not be greater than `count`.

Size

```
long Size ();
```

Returns `count`, the number of bytes in the stream.

Truncate

```
void Truncate (long newSize);
```

Reduces the number of bytes in the stream by setting `count` equal to `newSize`, which must not be greater than the existing value of `count`. If `pos` is greater than the new value of `count`, it is set equal to that value.

Writing

Put

```
void Put (void *bytes, long n);
```

Puts `n` bytes to the stream, starting at position `pos`. `pos` is increased by `n`; if `pos` then exceeds `count`, `count` is set equal to `pos`.

PutThru

```
void PutThru (void *bytes, char c);
```

Puts data to the stream starting with the first character of `bytes` and ending with the first occurrence of the character `c`. `pos` is increased by the number of bytes written; if `pos` then exceeds `count`, `count` is set equal to `pos`.

Reading

Get

```
void Get (void *bytes, long n);
```

Does nothing.

GetThruN

```
long GetThruN (void *bytes, char c, long n);
```

Returns `0`.

37 CCountingStream

CDataFile 38

Introduction

This class implements the functions you need to work with a Macintosh data file. You can use this class without creating a derived class if you need to read or write raw bytes. If you require a structured file, you should create a derived class of CDataFile.

Heritage

Base Class	CFile
Derived Classes	CPNTGFile
	CPictFile

Using CDataFile

You can use this class without creating a derived class to read and write the data fork of a Macintosh file. After creating an instance of CDataFile, use one of the specification member functions inherited from CFile to indicate which file you're working with. Then use the `Open` member function to open the file, and use the reading and writing member functions to read and write the file.

Some CDataFile functions use the THINK Class Library's Exception handling library if something goes wrong in a file operation. For more information on exception handling, see Chapter 9, "Exception Handling and RTTI."

Note

This version of CDataFile handles errors differently than the original version which has been renamed ODataFile and moved to the `Compatibility` folder.

Data Members

The only data member for this class is the `refNum` of the file. The Macintosh file system uses this number to identify the volume, directory, and file on disk. To learn more about the Macintosh file system, see *Inside Macintosh IV*, Chapter 19, "The File Manager."

Data member	Type	Description
`refNum`	`short`	The file's `refNum` when the file is open

Member Functions

These functions let you open and close data files, read and write data, and get information about a file. You need to use the functions inherited from CFile to specify which file you want to work with.

Creation and destruction

CDataFile
```
CDataFile();
```
Initializes the data file. This function sets `refNum` to 0.

IDataFile
```
void IDataFile();
```
Provided for backward compatibility.

Accessing

SetLength
```
void SetLength(long aLength);
```
Sets the end-of-file marker for this file at `aLength`.

GetLength
```
long GetLength();
```
Returns the length of the file. The file must be open already.

SetMark
```
void SetMark(long howFar, short fromWhere);
```
Specifies where subsequent read/write operations will take place. This position is called the "mark."

howFar specifies how far in bytes from the fromWhere parameter to set the mark. Positive values are offsets toward the end of the file. Negative values are offsets toward the beginning of the file. The argument fromWhere is one of the following:

FromWhere value	Meaning
fsFromStart	From the beginning of the file
fsFromLEOF	From the end of the file
fsFromMark	From the current position

Table 38-1 Values for FromWhere in CDataFile::SetMark

GetMark

```
long GetMark();
```

Returns the current position of the mark in markPos.

Open and close

Open

```
void Open(SignedByte permission);
```

Opens the file with the given permission. The permission argument can be one of the following:

Permission	Meaning
fsCurPerm	Same as the current permission
fsRdPerm	Read permission
fsWrPerm	Write permission
fsRdWrPerm	Read/write permission
fsRdWrShPerm	Shared read/write permission

Table 38-2 Values for permission in CDataFile::Open

See *Inside Macintosh IV*, Chapter 19, "The File Manager," for a discussion of permissions.

Close

```
void Close();
```

Closes the file and calls FlushVol to cause any buffered data to be written to disk.

IsOpen

```
Boolean IsOpen ();
```

Returns TRUE if the file is open, FALSE otherwise.

Read and write

ReadAll

```
Handle ReadAll();
```

Reads the entire contents of the file into a handle. This function allocates the handle. Here's an example of how to use `ReadAll`:

```
Handle theData;
…
theData = myDataFile->ReadAll();
```

ReadSome

```
void ReadSome(Ptr info, long howMuch);
```

Reads `howMuch` bytes into a buffer `info`. You must allocate the buffer prior to calling this function. Here's an example of how to use `ReadSome`:

```
char myBuffer[128];
…
myDataFile->ReadSome(myBuffer, 128L);
```

WriteAll

```
void WriteAll(Handle contents);
```

Writes the contents of the handle to the file.

WriteSome

```
void WriteSome(Ptr info, long howMuch);
```

Writes `howMuch` bytes from the buffer pointed to by `info`.

CDecorator 39

Introduction

CDecorator implements an object that arranges windows on the screen. There is only one instance of this class.

Heritage

Base Class	None
Derived Classes	None

Using CDecorator

CDecorator gives you member functions for arranging windows on the screen. There is only one instance of CDecorator which is stored in the global variable `gDecorator.`

After creating and initializing a new window, call CDecorator's `PlaceNewWindow` function. The decorator makes the window somewhat smaller than the main screen and offsets it down and to the right of any previous window. Using the decorator is optional.

You can derive from CDecorator if you like. For instance, you might want a decorator that implements window tiling. To do this, override the `MakeDecorator` function in your application class. Your override should create and initialize an object of your decorator derived class, and set the global variable `gDecorator` to point to it.

Data Members

Global variable

The global decorator object, created in the application member function `MakeDecorator`, is stored in the global variable `gDecorator`.

Data member	Type	Description
`gDecorator`	CDecorator	The window-dressing object

Instance data members

CDecorator defines these data members.

Data member	Type	Description
`wCount`	`short`	The number of new windows placed
`index`	`short`	Index for offsetting windows
`wWidth`	`short`	Width of a window in pixels
`wHeight`	`short`	Height of a window in pixels
`hLocation`	`short`	Horizontal location for next window
`vLocation`	`short`	Vertical location for next window

Member Functions

Construction and Destruction

CDecorator

```
CDecorator();
```

Constructor. The application member function `MakeDecorator` creates the global decorator.

~CDecorator

```
~CDecorator
```

An empty destructor.

PlaceNewWindow

```
void PlaceNewWindow (CWindow *theWindow);
```

Places a window specified by `theWindow` on the screen. The default function makes it fill up most of the screen and calls `StaggerWindow` to position it.

StaggerWindow

```
void StaggerWindow (CWindow *theWindow);
```

Positions a new window specified by the window on the screen. Positions it offset down and to the right of the last window the decorator placed.

CenterWindow

```
void CenterWindow (CWindow *theWindow);
```

Centers the window specified by `theWindow` on the main screen. If `theWindow` is a modal dialog, this function tries to put it in the top third of the main screen. This function does not increment `wCount` or change any of the data members.

GetWCount

```
short GetWCount();
```

Returns the number of windows the decorator has put on the screen. You can use the returned value to give your untitled windows sequential numbers such as Untitled-1.

IDecorator

```
void IDecorator();
```

Provides for backward compatibility. This function is not needed.

39 CDecorator

CDesktop 40

Introduction

CDesktop implements a view that occupies the entire screen. This view serves as the top of the visual hierarchy and as the enclosure for all windows. Normally, there is only one instance of CDesktop.

Heritage

Base Class	CView
Derived Classes	CFWDesktop

Using CDesktop

The most common way you'll use the desktop is as the enclosure for your windows. Your application should not call desktop functions. Instead, you should rely on higher level objects like windows and directors to call desktop functions.

The desktop is stored in the global variable `gDesktop`. The CApplication function `MakeDesktop` creates an instance of the CDesktop and stores a pointer to it in that variable.

Global Variables and Data Members

Global variable

The global `gDesktop` holds a pointer to the single instance of the desktop. Use this variable to specify the enclosure for your application's windows.

Global variable	Type	Description
`gDesktop`	`CDesktop`	The single instance of the desktop

Data members

CDesktop defines these data members:

Data member	Type	Description
bounds	Rect	Boundaries of the desktop
itsWindows	CWindowList*	List of windows
topWindow	CWindow*	Topmost application window
itsFloats	CWindowList*	Floating application windows
topFloat	CWindow*	Topmost floating window

Member Functions

Creation and destruction

CDesktop

```
CDesktop();
```

Constructor. This function opens a grafport that your application uses as its desktop. The desktop's supervisor is set to the application, stored in the global variable gApplication.

~CDesktop

```
~CDesktop();
```

Destructor. It destroys all of the desktop's windows.

IDesktop

```
void IDesktop (CBureaucrat *aSupervisor);
```

Provides for compatibility with previous releases. This function may only be called if the constructor was called with no arguments.

Appearance

Show

```
void Show();
```

Makes a desktop visible by showing all of its windows. This function calls the Show function of all of the desktop's windows.

Hide

```
void Hide();
```

Hides a desktop by hiding all of its windows. This function calls the Hide function of all of the desktop's windows.

Activate

```
void Activate();
```

Activates the desktop by activating the top window. This function calls the `Activate` function of the desktop's top window.

Deactivate

```
void Deactivate();
```

Deactivates the desktop by deactivating its top window. This function calls the `Deactivate` function of the desktop's top window.

ReallyVisible

```
Boolean ReallyVisible();
```

Returns `TRUE` if the desktop is visible.

Mouse

DispatchClick

```
void DispatchClick (EventRecord *macEvent);
```

Processes a mouse click. If the user clicks the mouse, `DispatchClick` determines where the mouse went down, and call, the appropriate function of the object the click is intended for. If the top window is modal, and the mouse did not go down in the menu bar, this function beeps and does not process the click.

This function uses what is returned by the Macintosh Toolbox routine `FindWindow` returns to determine what function to call on which object.

Part	Action
`inDesk`	If the mouse goes down in the desktop, this member function calls the desktop's `DoClick` function. The default `DoClick` function does nothing.
`inSysWindow`	If the mouse goes down in a system window, this function calls the Toolbox routine `SystemClick.`

Table 40-1 How CDesktop dispatches events

Part	Action
`inMenuBar`	If the mouse goes down in the menu bar, this function calls the Toolbox routine `MenuSelect`. This function then calls the bartender's `FindCmdNumber` function to convert the menu selection to a command number. This command number is sent to the gopher via a `DoCommand` function call.
`inContent`	If the mouse goes down in the content region of a window and the window is not active, this function calls the window's `Select` function. If the window is active and can receive clicks, this function calls the window's `DispatchClick` function, which eventually calls the `DoClick` function of a pane or the window itself.
`inDrag`	If the mouse goes down in the drag region (the title bar), this function calls the window's `Drag` function.
`inGrow`	If the mouse goes down in the window's grow region (the lower right corner), this function calls the window's `Resize` function.
`inGoAway`	If the mouse goes down and up in the window's close box, this function calls its `Close` function.
`inZoomIn`, `inZoomOut`	If the mouse goes down and up in the window's zoom box, this function calls its `Zoom` function.

Table 40-1 How CDesktop dispatches events *(Continued)*

DoMouseUp

```
void DoMouseUp(EventRecord *macEvent);
```

Handles a mouse up event in the desktop. The default function does nothing.

DispatchCursor

```
void DispatchCursor (Point where,
   RgnHandle mouseRgn);
```

Sets the cursor for the view the cursor is in. If the cursor is over an active window, this function calls the window's `DispatchCursor` function. If the cursor is in the menu bar, this function sets it to the arrow. If the cursor is not in the menu bar or in an active window, this function calls the desktop's `AdjustCursor` function. You should not override this function.

AdjustCursor

```
void AdjustCursor (Point where,
   RgnHandle mouseRgn);
```

Adjusts the cursor shape when the mouse is in an insignificant part of the desktop. The default function sets the cursor to an arrow.

Contains

```
Boolean Contains(Point thePoint);
```

Returns `TRUE` if `thePoint` is within the bounds of the desktop.

HitSamePart

```
Boolean HitSamePart(Point pointA, Point pointB);
```

Returns `TRUE` if `pointA` and `pointB` are reasonably close together. Reasonably close means that the two points are within `REASONABLY_CLOSE` pixels.

Window

You do not have to use any of the following functions yourself. Directors, the objects that manage the interaction between windows and the desktop, take care of most of these function calls.

AddWind

```
void AddWind (CWindow *theWindow);
```

Adds `theWindow` to the desktop's window list.

RemoveWind

```
void RemoveWind (CWindow *theWindow);
```

Removes `theWindow` from the desktop's window list.

SelectWind

```
void SelectWind (CWindow *theWindow);
```

Selects `theWindow` and brings it to the front.

ShowWind

```
void ShowWind (CWindow *theWindow);
```

Makes `theWindow` visible on the desktop.

HideWind

```
void HideWind (CWindow *theWindow);
```

Hides `theWindow`.

DragWind

```
void DragWind (CWindow *theWindow,
   EventRecord *macEvent);
```

Drags `theWindow` on the desktop.

UpdateWindows

```
void UpdateWindows ();
```

Calls `Update` for each of the desktop's windows.

Accessing

GetTopWindow

```
CWindow* GetTopWindow ();
```

Gets the frontmost window.

NthWindow

```
CWindow* NthWindow(long index);
```

Returns the non-floating window at the specified index.

GetWindowIndex

```
long GetWindowIndex(CWindow *wind);
```

Returns the index number of the specified index.

GetNumWindows

```
long GetNumWindows();
```

Returns the number of non-floating windows.

DoForEachWindow

```
void DoForEachWindow(wEachFunc1 fun, long param);
```

Applies the function pointed to by `fun` to each non-floating window by calling `DoForEach1(fun, param)`. For each such window, `param` is passed as a single parameter to `fun`.

FirstSuccessWindow

```
CWindow* FirstSuccessWindow(wTestFunc1 fun,
   long param);
```

Applies the test function pointed to by `fun` to each window by calling `FindItem1(fun, param)`. For each window, `param` is passed as a single parameter to `fun`. This function returns a pointer to the first window that passes the test, or it returns `NULL` if none do.

SetIndex

```
Boolean SetIndex(CWindow *wind, long index);
```

Changes the index of a non-floating window. This function allows you to move a window up or down in the non-floating layer. Returns `FALSE` if the window can't be moved as specified by `index`. For example, trying to move a non-modal window in front of a modal one would make `SetIndex` return `FALSE`.

GetLastModal

```
CWindow* GetLastModal (Boolean *anyVisible);
```

Returns the bottommost modal window if there is one; if there are no modal windows, returns `NULL`. Also, if any modal window is visible, sets `anyVisible` to `TRUE`; otherwise, sets `anyVisible` to `FALSE`.

GetBounds

```
void GetBounds(Rect *theBounds);
```

Gets the bounds of the desktop.

GetAperture

```
void GetAperture (LongRect *theAperture);
```

Gets the visible portion of the desktop. This function returns the bounds of the desktop since all of the desktop is always visible.

CalcTopFloat

```
void CalcTopFloat();
```

Sets `topFloat` to the first visible window in `itsfloats` if the list is nonempty.

Calibration

Prepare

```
void Prepare ();
```

Sets the port to the desktop port. You should not use or override this function.

Member Functions: Protected

IDesktopX

```
void IDesktopX ();
```

Internal initialization member function.

CDialog 41

Introduction

CDialog is an abstract class for modal and modeless dialog windows.

Heritage

Base Class	CWindow
Derived Classes	CDLOGDialog

Using CDialog

CDialog is the fundamental dialog class. It is a derived class of CWindow that has additional member functions to provide typical dialog services. You can use CDialog anywhere you would use CWindow. A dialog is usually supervised by a dialog director, which takes care of creating the dialog, handling its commands, and closing it.

CDialog defines behavior that's useful for most dialogs, including:

- Validation. Before the dialog is closed, the dialog director calls `Validate` for all its CDialogText items. The items return `TRUE` only if the item is valid according to the constraints you set. The dialog is closed only if all items return `TRUE`.
- Default button. You can set a default button, which is highlighted with a bold border.
- Keyboard shortcuts. Press Command-Period or Escape to select the Cancel button. Press Enter or Return to select the default button if there is one.
- Tabbing. Use Tab to move though the fields in your dialog, and use Shift-Tab to move backward.

- Scrolling. If you create a large dialog, the THINK Class Library handles scrolling for you.

CDialog can be the basis of a modal or modeless dialog. It can even be a document window; for example, the document window for a database program may need the validation and tabbing support that CDialog provides.

Unless you explicitly define a CDialog derived class, code generated by Visual Architect will use the CDialog class itself as the dialog window class. CDialog can be used out of the box because all initialization for the window and its panes is performed by `GetFrom` member functions called to read the dialog from the `'CVue'` resource created by Visual Architect. Supervisory functions are performed by your CDialogDirector derived class, which Visual Architect always generates.

If you define a dialog without using Visual Architect or CDLOGDialog, you'll probably want to create a derived class of CDialog. Your derived class needs to have a constructor that builds the dialog and all the elements inside it. Unless you need to override standard dialog behavior, your derived class doesn't need to define any other member functions.

The THINK Class Library also includes a derived class, CDLOGDialog, which builds a dialog from `'DLOG'` and `'DITL'` resources, which can be defined using ResEdit or THINK Rez.

A dialog usually belongs to a dialog director, which takes care of creating the dialog, handling its commands, and closing it. Your program doesn't generally need to manipulate the dialog directly, but instead sends commands to the dialog's director. To create a director for a CDialog, you should derive CDialogDirector. To create a director for a CDLOGDialog, you should derive from CDLOGDirector.

If the dialog is a documents only window, its director should be the document. If the dialog is one window for a multi-window document, its director should be a dialog director.

Every item in a dialog is a derived class of CPane. You can use any pane in a dialog plus any of your own pane derived classes. There are two special derived classes of CEditText — CDialogText and CIntegerText — that you should use for text items. CDialogText and

CIntegerText cooperate with the dialog to ensure that validation and tabbing work correctly.

In most windows, the document or director is the supervisor of a window's panes. However, in dialogs, the dialog itself must be the supervisor of the dialog's panes, because the CDialog class handles special keys, such as Return, Enter, Command-Period, and Escape. Since the dialog's director is the direct supervisor of the dialog, the director still receives commands and events from the dialog's panes. The dialog just gets them first.

Data Members

CDialog defines the following protected data members.

Data member	Type	Description
`defaultBtn`	`CButton*`	Current default button.
`itsPanorama`	`CPanorama*`	Panorama that encloses all dialog items.
`scrollable`	`Boolean`	`TRUE` if auto-scrolling is enabled. Set `TRUE` if `itsPanorama` is enclosed in a scroll pane.

Member Functions

Creation

CDialog

```
CDialog (short WINDid,
   CDirector *aSupervisor);
```

Constructor. Initializes a dialog window from a `'WIND'` resource. `WINDid` is the ID of the `'WIND'` resource. `aSupervisor` is the window's supervisor. A window's enclosure is always `gDesktop`.

CDialog

```
CDialog (Rect *bounds, Boolean fVisible,
   short procID, Boolean fHasGoAway,
   CDirector *aSupervisor);
```

Constructor. Initializes a dialog from the parameters in the argument list.

`Bounds` is a rectangle that describes the size and position of the window. `aProcID` is a `short` that describes the type of the window. If `fVisible` is `TRUE`, the window is drawn immediately

after it's created. If `fHasGoAway` is `TRUE`, the window has a close box. `aSupervisor` is the window's supervisor.

CDialog

```
CDialog ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. It also can be used in combination with IDialog for backward compatibility.

IDialog

```
void IDialog (short WINDid,
   CDesktop *anEnclosure,
   CDirector *aSupervisor);
```

Initialization function for backward compatibility with previous release. May not be called if constructor has arguments. Arguments are the same as the constructor's with the addition of `anEnclosure`.

INewDialog

```
void INewDialog (Rect *bounds, Boolean fVisible,
   short procID, Boolean fHasGoAway,
   CDesktop *anEnclosure,
   CDirector *aSupervisor);
```

Initialization function for backward compatibility with previous release. May not be called if constructor has arguments. Arguments are the same as constructor with the addition of `anEnclosure`.

Close

```
void Close ();
```

Closes the dialog, if all the fields have valid entries. The function calls `Validate` to check the fields.

Validate

```
Boolean Validate ();
```

Returns `TRUE` if all the fields in the dialog have valid entries. Returns `FALSE` otherwise. This member function validates the values in the CDialogText fields.

Accessing

FindButton

```
CButton *FindButton (long aCmd);
```

Returns the first button in the view hierarchy associated with the command `aCmd`.

SetDefaultCmd

```
void SetDefaultCmd (long aCmd);
```

Sets the default button to the one associated with `aCmd`. This member function finds the button associated with `aCmd`, and then calls `SetDefaultButton` with the button.

SetDefaultButton

```
void SetDefaultButton (CButton *aButton);
```

Sets the default button to `aButton`. CDialog draws the button's border by calling its `SetDefault` function with `TRUE`. If there is already a default button, CDialog removes the border by calling its `SetDefault` function with `FALSE`.

Commands

DoKeyDown

```
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handles a key-down event that none of the dialog's panes can handle. It takes care of Tab, Return, Enter, Escape, and Command-Period, as in Table 41-1

Key	Description
Tab	Moves to the next dialog item.
Shift-Tab	Moves to the previous dialog item.
Return or Enter	Selects the default command, usually `cmdOK` or `cmdCancel`.
Command-Period or Escape	Selects `cmdCancel`.

Table 41-1 How CDialog handles keys

CDialog handles Tab and Shift-Tab by calling `DoTab`.

DoTab

```
void DoTab (Boolean fForward);
```

Handles a Tab. If `fForward` is `TRUE`, it moves to the next dialog item. Otherwise, it moves to the previous dialog item. CDialog calls `FindGophers` to determine the item to which to move.

FindGophers

```
void FindGophers (tGopherInfo *gopherInfo);
```

Returns what the next, previous, last, and first potential gophers are, by setting the fields in `gopherInfo`. You must set the `currentGopher` field to the current gopher if it's a dialog item; if not a dialog item, set it to `NULL`.

Table 41-2 lists the fields in `gopherInfo` that you use. They are all of type CView*.

Field	Description
`currentGopher`	The current gopher, if it's a dialog item. `NULL`, otherwise. You must set this item before calling `FindGophers`.
`firstGopher`	The first dialog item that can be a gopher.
`lastGopher`	The last dialog item that can be a gopher.
`nextGopher`	The dialog item after `currentGopher` that can be a gopher. If `currentGopher` is `NULL` or there is no next gopher, then `nextGopher` is set to `firstGopher.`
`prevGopher`	The dialog item before `currentGopher` that can be a gopher. If `currentGopher` is `NULL`; or there is no previous gopher, then `prevGopher` is set to `lastGopher`.

Table 41-2 What FindGophers returns in gopherInfo

This function uses the same order in which you created the panes as the tabbing order. It scans through all the panes within the dialog. Panes that return `TRUE` to both the `ReallyVisible` and `CanBeGopher` functions are considered part of the tabbing sequence for the dialog.

Appearance

SetCmdEnable

```
void SetCmdEnable (long aCmd, Boolean fEnable);
```

Activates the button associated with `aCmd`, if `fEnable` is `TRUE`. Otherwise, it deactivates the button.

Scrolling

ScrollToPane

```
void ScrollToPane (CPane *aPane);
```

Ensures that the pane `aPane` is visible by scrolling it into view if necessary. This function does nothing if the dialog is not scrollable.

Change notification

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

Responds to a change in a provider. This function attempts to scroll a dialog item into view by calling `ScrollToPane`, if `reason` is `bureaucratIsGopher` and `aProvider` is a dialog item.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

MakePanorama

```
void MakePanorama (Boolean fHasHScroll,
   Boolean fHasVScroll, Boolean fHasSizeBox);
```

Creates a panorama object to act as a background for all the dialog items. If either `fHasHScroll` or `fHasVScroll` is `TRUE`, this function also creates a scroll pane for this dialog.

Set `fHasHScroll` to `TRUE` if you want horizontal scroll bars. Set `fHasVSroll` to `TRUE` if you want vertical scroll bars. Set `fHasSizeBox` to `TRUE` if you want a size box.

You should not have to call this member function.

Member Functions: Private

CDialogX

```
void CDialogX():
```

Performs common initialization.

CDialogDirector 42

Introduction

CDialogDirector is a base class for a director that manages a dialog window.

Heritage

Base Class	CDirector
Derived Classes	CDLOGDirector

Using CDialogDirector

CDialogDirector is an abstract class that manages a dialog, which can be modal or modeless.

Even though CDialogDirector has no pure virtual functions, you need to create or use a derived class of CDialogDirector. At a minimum, you need to write a constructor that creates your dialog window. You may also want to override `DoCommand` to handle the commands your dialog produces and override `ProviderChanged` to handle changes to dialog panes.

Visual Architect generates a CDialogDirector derived class for each dialog you define. The dialog window is initialized at run-time by reading the `'CVue'` resource for the dialog. Any run-time initialization of the dialog or its panes should be done by overriding the `GetFrom` member function for the appropriate pane or dialog class, or by overriding the `SetData` function of the generated dialog director. A pointer to the window created from the `'CVue'` resource is set in the `itsWindow` data member after initialization from the resource is complete. The generated code also initializes a pointer to each pane in the dialog.

The THINK Class Library includes a derived class, CDLOGDirector, which creates and manages a dialog object of type CDLOGDialog, which is initialized from a `'DLOG'` resource.

A dialog director validates its dialog contents before closing the dialog. It calls its dialog's `Validate` member function, which then calls `Validate` for all CDialogText panes. If a pane doesn't meet all the constraints you've set, `Validate` returns `FALSE`. If any pane returns `FALSE`, the director doesn't close the dialog. To provide other types of validation, such as checking for conflicting combinations of control settings, you can define a derived class of CDialogDirector and override `Validate`. Make sure that the inherited `Validate` function succeeds, then perform your own validation.

If your dialog is a modal dialog, read the next section, "Modal dialog loop," to find out how to use the modal dialog in your program. If your dialog is modeless or is a document window, create it and use it like any other window. You can take advantage of tabbing and validation support. When the user tries to close the dialog, either by clicking the close box or by quitting the application, it's closed only if validation succeeds. For a modeless dialog, if you need to do something after the window is closed but before it's disposed, override `EndDialog`, and call the inherited `EndDialog` function. If it returns `TRUE`, you know that the dialog will close and your `EndDialog` function can take the appropriate action before returning.

Modal dialog loop

To use a modal dialog, you need to create a derived class of CDialogDirector. If you're using `'DLOG'` resources, you need to derive from CDLOGDirector. Extremely simple dialogs can even be handled by instantiating CDLOGDirector directly, because it creates the window automatically.

Here's an example:

```
void CMyDialogDirector::TalkToUser(void)
{
   long dismissCommand;

      // Before calling
      // BeginModalDialog
      // you should setup the dialog
      // with the desired settings

   dismissCommand=DoModalDialog(cmdOK);

   if (dismissCommand == cmdOK)
   {
      // The dialog was confirmed and
      // validated. Process the
      // input.
   }
}
```

The dialog director's `DoModalDialog` function sets the dialog to modal and processes events until the user issues a command to dismiss the dialog. Usually the command is `cmdOK` or `cmdCancel`, but you can have other commands exit the modal dialog loop. At this point, you can exit the dialog or call `DoModalDialog` again to continue the dialog.

The dialog director's `EndDialog` function sets the data member `dismissCmd` to the given command. This signals `DoModalDialog` to return after processing the current event. If the `fValidate` flag is `TRUE`, it also validates the dialog. If validation fails, `dismissCmd` is not set, so `DoModalDialog` does not return.

When `DoModalDialog` exits, it restores the changed state of any supervising document by calling `SetChanged` with the value returned by `GetChanged` on entry to `DoModalDialog`.

During the call to `DoModalDialog`, your application is running its event loop and processing events. However, most of the menus are disabled and the dialog window is the only window available. Any buttons or panes that call `DoCommand` functions continue to work. CDialogDirector recognizes only `cmdOK` and `cmdCancel`. To handle other commands, you must derive a class from CDialogDirector (or CDLOGDirector), override `DoCommand`, and check for your own command numbers. If one of your own command numbers dismisses the dialog, call the dialog director's `EndDialog` function.

Pass it the command number and a flag indicating whether or not to validate. You don't need to perform validation if the user cancels the dialog.

Here is an example of `DoCommand` for a CDialogDirector-derived class:

```
void CMyDialogDirector::DoCommand (long
   theCommand)
{
   switch (theCommand)
   {
      case cmdOtherCmd1:
         DoOtherCmd1();
            // This one doesn't dismiss
            // the dialog
         break;

      case cmdOtherCmd2:
         EndDialog(cmdOtherCmd2, TRUE);
            // This one does dismiss it
         break;

      default:
         CDialogDirector::DoCommand(
                           theCommand);
         break;
   }
}
```

Data Members

CDialogDirector defines these protected data members:

Data member	Type	Description
`dismissCmd`	`long`	The command that dismissed the dialog
`barState`	`long`	The state of the menu bar (enabled or disabled) when `DoModalDialog` began

Member Functions

Creation

CDialogDirector

```
CDialogDirector (CDirectorOwner
   *aSupervisor = NULL);
```

Constructor. `aSupervisor` is the director, document, or application that owns the dialog.

CDialogDirector

```
CDialogDirector ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with IDialogDirector for backward compatibility.

IDialogDirector

```
void IDialogDirector (CDirectorOwner
   *aSupervisor);
```

Initialization function compatible with previous release. Must not be called if constructor has an argument.

Command

DoCommand

```
void DoCommand (long aCmd);
```

Handles the standard OK and Cancel commands. OK closes the dialog if `Validate` returns `TRUE`; Cancel always closes the dialog.

Close

```
Boolean Close (Boolean fQuitting);
```

Handles the Close command. This function closes the dialog if `Validate` returns `TRUE`. Returns `TRUE` if the dialog was closed; returns `FALSE` otherwise.

BeginDialog

```
void BeginDialog ();
```

Starts using the dialog. This function selects the window, activates the first dialog item that's a gopher and, if the item is a text field, selects all its text. `BeginDialog` should not be called for a modal dialog.

BeginModalDialog

```
void BeginModalDialog();
```

Provides backward compatibility. Use `DoModalDialog` instead.

DoModalDialog

```
long DoModalDialog (long defaultCmd);
```

Enters a modal loop until the user dismisses the dialog. It returns the command that dismissed the dialog, usually `cmdOK` or `cmdCancel`. The dialog encloses `defaultCmd`, the default command button, in a thick border. It usually is `cmdOK` or `cmdCancel`.

DoChangeableModalDialog

```
long DoChangeableModalDialog (long defaultCmd,
   Boolean changeDoc);
```

Similar to `DoModalDialog`, except that changes in the dialog may dirty the document. The changed state is reset to its original condition if the command that ends the dialog is `cmdCancel`.

EndDialog

```
Boolean EndDialog (long withCmd,
   Boolean fValidate);
```

Tries to stop using the dialog. If `fValidate` is `TRUE`, the function validates the dialog's items by calling `Validate`. If `Validate` returns `TRUE` or `fValidate` is `FALSE`, it sets `dismissCmd` to `withCmd`, which is the command the user selected to close the dialog. This function does not close the dialog.

Validation

Validate

```
Boolean Validate ();
```

Returns `TRUE` if this director's dialog is valid, according to the constraints you set; otherwise returns `FALSE` . This function calls the `Validate` function of the director's dialog.

Member Functions: Protected

DisableTheMenus

```
void DisableTheMenus ();
```

Disables all menus except the **Edit** menu. Called just before bringing up a modal dialog.

EnableTheMenus

```
void EnableTheMenus ();
```

Enables all menus. Called after a modal dialog is dismissed.

CDialogText 43

Introduction

CDialogText is a derived class of CEditText for text fields in dialogs.

Heritage

Base Class	CEditText
Derived Classes	CIntegerText

Using CDialogText

This class adds some enhancements to text panes that are especially useful in dialogs. The features include:

- Validating a field before closing the dialog.
- Drawing a rectangular border around the text field.
- Passing special keys to the dialog such as Tab, Return, Enter, Command-Period (Command-.) and Escape.
- Enabling the **Edit** menu in a modal dialog when a text field is active.
- Visually disabling by graying out the border and suppressing the text.
- Making a field uneditable by removing the border and making it not want clicks.

The most useful feature is validation. After you create the field, you specify constraints; for example, the field must be filled in and contain fewer than 25 characters. Before the dialog closes, it calls the `Validate` function of all its CDialogText items. If a constraint isn't met in a field, it returns FALSE. The dialog isn't closed if any field returns FALSE, and it displays an alert describing the problem.

This class lets you specify these constraints, with `SetConstraints`:

- Required. A required field is valid if it's not empty.
- Maximum length. A field is valid if it contains no more than the maximum number of characters.

If you want to add move constraints, define a derived class of CDialogText and override `Validate` to specify your new constraints. Add strings to the `'STR#'` resource `STRdlgValidation` to describe the new errors.

If the text is changed, a `dialogTextChanged` is broadcast to its dependents. The info parameter is a `short*` to the Pane ID.

CIntegerText is a derived class for numeric fields in dialogs. It lets you specify a minimum and maximum value.

Data Members

CDialogText defines these protected data members:

Data member	Type	Description
`isRequired`	`Boolean`	`TRUE` if must contain some text
`maxValidLength`	`long`	Maximum number of characters
`validateOnResign`	`Boolean`	`TRUE` if validate on resigning gopher

Member Functions

Creation

CDialogText

```
CDialogText ();
```

Default constructor.

CDialogText

```
CDialogText (CView *anEnclosure,
   CView *aSupervisor, short aWidth,
   short aHeight, short aHEncl, short aVEncl,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY,
   short aLineWidth = -1, Boolean
   aScrollHorizv = 0, Boolean aIsRequired,
```

```
   long aMaxValidLength = MAXLONG,
   Boolean aValidateDnResign = TRUE);
```

Constructor. Most of the arguments to this member function are identical to those for `CEditText`. `aIsRequired` `TRUE` makes entry required; the default is `FALSE`. `aMaxValidLength` specifies the maximum number of characters allowed; the default is `MAXLONG` (no limit). If aValidateOnResign is TRUE, the text is validated when it is no longer the gopher. This function creates a non-stylable border (`stylable` is `FALSE`) around the field:

To change whether the field is stylable, use `Specify`, described in "Specify" in Chapter 14, "CAbstractText." To change whether the field is validated when it's no longer the gopher, your derived class must set `validateOnResign`.

~CDialog Text

```
~CDialogText();
```

Destructor.

IDialogText

```
void IDialogText (CView *anEnclosure,
   CView *aSupervisor, short aWidth,
   short aHeight, short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initialization function compatible with the previous release. Must not be called if constructor arguments are used.

IViewTemp

```
void IViewTemp(CView *onEnclosure, CBureaucrate
   *aSupervisor, Ptr *viewData);
```

Used to initialize a CDialogText from a resource template.

Accessing

SetConstraints

```
void SetConstraints (Boolean fRequired,
   long aMaxChars);
```

Specifies how to validate this field. If `fRequired` is `TRUE`, the field is required and must contain at least one character. `aMaxChars` is the maximum number of characters the field may hold.

GetTextString

```
void GetTextString (StringPtr aString);
```

Returns the contents of the dialog text field as a string. If the text is longer than 255 characters, only the first 255 will be returned.

GetHelpBallonState

```
short GetHelpState();
```

Returns the balloon help state:

kHMEnabledItem (0) -	Enable (or, for buttons, off)
kHMDisabledItem (1) -	Disabled (dimmed)
KHMCheckITem (2) -	Enabled and checked (or, for buttons, on)
kHMMarkedItem (3) -	Enabled and marked (for any other purpose)

Controls and edit panes typically implement only the first two of three of the above states.

There is not necessarily a one-to-one correspondence between balloon help states and THINK Class Library variable states. The dimmed state is the most complex. Dimmed means the view would normally accept user interaction but currently does not. Note that a view must normally want clicks or allow editing to be considered dimmed when it does not want clicks or allow editing. Classes for which dimmed or marked states are meaningful, or that want to use the marked state, must override this function.

Command

DoKeyDown

```
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handles a key-down event in a dialog text field. It passes keys that have a special meaning to dialogs, including Tab, Return, Enter, Escape, and Command-. (Command-Period), to `itsSupervisor`. It passes all other keys through to the base class.

BecomeGopher

```
Boolean BecomeGopher (Boolean fBecoming);
```

If `fBecoming` is `TRUE`, this dialog text field is becoming the gopher. This function enables the **Edit** menu if the field is in a modal dialog. If `fBecoming` is `FALSE`, this field is no longer the gopher. This function disables the **Edit** menu, if this field is in a modal dialog. It also validates the field if `validateOnResign` is `TRUE`.

Drawing

Draw

```
void Draw(Rect*);
```

Draws the text unless disabled.

Validation

Validate

```
Boolean Validate ();
```

Returns `TRUE` if the field is valid, according to the constraints you set. Returns `FALSE` otherwise. You set the constraints with the function `SetConstraints`. To be valid the field must contain no more than the maximum number characters, and, if the field is required, it must contain at least one character. Override this function if your derived class adds validations.

Enabling and disabling

SetEnabled

```
void SetEnabled (Boolean isEnabled);
```

By default, a CDialogText object is enabled and ready for users to edit. Calling `SetEnabled(FALSE)` makes the field uneditable, visually disables the field by drawing its border in gray and suppresses the drawing of its text value. The text value itself is not affected. `SetEnabled(TRUE)` restores the field to its default state.

SetEditable

```
void SetEditable (Boolean isEditable);
```

Calling `SetEditable(FALSE)` makes the field uneditable and removes the border. In other words, it temporarily turns it into a static text field. Calling `SetEditable(TRUE)` restores the field to its default state.

Object I/O

PutTo
```
void PutTo (CStream& aStream);
```
Writes to the stream.

GetFrom
```
void GetFrom (CStream& aStream);
```
Reads from the stream.

Member Functions: Protected

IDialogTextX
```
void IDialogTextX ();
```
Protected function. Performs common initialization.

DoSetEnabled
```
void DoSetEnabled (Boolean isEnabled);
```
Protected function called by `SetEnabled` and `SetEditable`. Does the work for `SetEnabled`.

DoSetEditable
```
void DoSetEditable (Boolean isEditable);
```
Protected function called by `SetEnabled` and `SetEditable`. Does the work for `SetEditable`.

MakeBorder
```
void MakeBorder ();
```
Protected function creates a pane border for the dialog text field. Override this function to change the border's appearance.

ReportInvalidText
```
void ReportInvalidText (short strIndex);
```
Protected function reports that the field is invalid by displaying an alert with string `strIndex` in `'STR#'` resource `STRdlgValidation`. If your derived class needs to report more

validation errors, add the error string to the `'STR#'` resource `STRdlgValidation`.

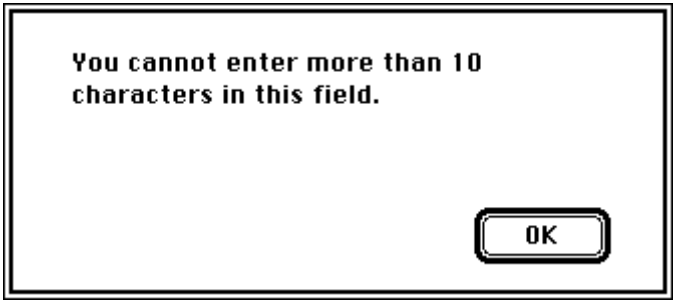


Figure 43-1 Invalid text alert

SetStatic

```
void SetStatic (Boolean isStatic);
```

Protected function called by `SetEnabled` and `SetEditable` which does common processing for both functions.

CDirector 44

Introduction

CDirector is a base class for supervising a window that can handle commands. Directors can act as an intermediary between the application data represented in a window and the panes in the window.

Heritage

Base Class	CDirectorOwner
Derived Classes	CDialogDirector
	CDocument
	CClipboard
	CTearOffMenu
	CFloatDirector

Using CDirector

A director is a base class that manages the communication between the application and a window. Anytime you want to display a window, it must be supervised by a director. The supervisor of a director must be the application or another director.

In most cases, you'll derive from the CDocument class to display and manipulate data stored in a file in a window. You'll derive the class from CDialogDirector to manage a dialog, or from CTearOffMenu to implement a tear-off menu. The only time you'll create a direct derived class of CDirector is when you need a special kind of window such as a status window or a subwindow.

When a window belonging to a director becomes the active window, the gopher points to the bureaucrat specified by the director's `itsGopher` data member. When the window becomes inactive, the gopher points to the application. When the user chooses a command

from the menu, the switchboard calls the gopher's DoCommand member function.

When the switchboard processes a window-related event (such as an activate event), it calls the appropriate window member function which in turn notifies the director.

Global Variables and Data Members

Global variables

The global variable gGopher points to the current bureaucrat. When a window becomes active, gGopher points to the bureaucrat specified in the director's itsGopher data member. When there are no active directors, gGopher points to the application.

Data member	Type	Description
gGopher	CBureaucrat*	The current bureaucrat

The data member itsGopher usually points to the main pane of a window. When the director becomes active, the bureaucrat specified in itsGopher becomes the gopher.

Data member	Type	Description
itsWindow	CWindow*	Window that the director controls
itsGopher	CBureaucrat*	Bureaucrat to become the gopher when the director is activated
activateWindOnResume	Boolean	TRUE, if the director activates its window when the program is resumed
dirty	Boolean	TRUE if document has been altered
wasDirty	Boolean	TRUE if it was dirty prior to command

Data member	Type	Description
cTaskWindow	CWindow*	Pointer to window from which the last task originated
alreadyClosing	Boolean	TRUE while attempting to close the director

Member Functions

Creation and destruction

CDirector

```
CDirector (CDirectorOwner *aSupervisor);
```

Constructor. Adds the director to aSupervisor's list of directors. By default, the director has no window and is not active. The itsGopher data member is initialized to point to the director. Your director derived class should create the window for the director.

CDirector

```
CDirector ();
```

Default constructor. Implicitly called when object is created by new_by_name. Can also use in combination with IDirector compatibly with previous release.

~CDirector

```
~CDirector ();
```

Destructor. ~CDirector deletes the director's window (if it has one) and removes the director from the supervisor's list of directors.

IDirector

```
void IDirector (CDirectorOwner *aSupervisor);
```

Initialization function compatible with the previous release. Must not be called if constructor argument is used.

Accessing

GetWindow

```
CWindow* GetWindow ();
```

Returns the window that the director controls.

FindViewByID

```
CView* FindViewByID (long aViewID);
```

Within the director's window, locate the view with ID `aViewID`.

OwnsWindow

```
Boolean OwnsWindow (CWindow *aWindow);
```

Returns `TRUE` if this director owns `aWindow`. The director owns the window if the window's supervisor is this object.

Commands

DoCommand

```
void DoCommand (long theCommand);
```

Handles a command. The default function handles this command:

Command	Description
`cmdClose`	Call `Close(FALSE)`

Your director class will usually override this function. You should handle your own commands first, then call the inherited `DoCommand` function to get the generic effects.

UpdateMenus

```
void UpdateMenus ();
```

If a window is associated with the director, this function enables the **Close** command (`cmdClose`).

Notify

```
void Notify (CTask *theTask);
```

A subordinate has completed a task. `Notify` stores `itsWindow` in `cTaskWindow`, calls `SetChanged(TRUE)` and calls the application's `NotifyClean` function.

NotifyClean

```
void NotifyClean(CTask *theTask);
```

This function operates the same as `Notify`, except that it doesn't call `SetChanged`.

SetChanged

```
void SetChanged (Boolean isChanged);
```

This function saves the current value of `dirty` in `wasDirty` and sets `dirty` to `TRUE`. It then calls its inherited `SetChanged` function to pass the change on to its director supervisors, if any.

GetChanged

```
Boolean GetChanged ();
```

If `dirty` is `TRUE`, `GetChanged` returns `TRUE`; otherwise, it returns the value returned by its inherited `GetChanged` function. `SetChanged` and `GetChanged` work as they do to allow windows to be individually dirty. When a director is supervised by a document, if the director is dirty, the document is dirty, but the document may be dirty and the director not. By default, the only class that calls `GetChanged` is CDocument. If you want a non-document director to behave differently based on its `dirty` flag, you will have to add this behavior to your own director derived class.

ToggleChanged

```
void ToggleChanged ();
```

This function calls `SetChanged(wasDirty)`. The application calls `ToggleChanged` when it accepts an undo or redo command. The point of `ToggleChanged` is to ensure that if a user undoes the first change to a window, the director's `dirty` flag will be correctly set.

IsTaskWindowInFront

```
Boolean IsTaskWindowInFront ();
```

This function returns `TRUE` if the director's window is the front window and the last task originated from this window and the window is visible. The undo command is enabled only if the first director in the chain of command returns `TRUE`.

Appearance

Activate

```
void Activate ();
```

A director is becoming active. If the director's window is not a floating window, the function calls the `BecomeGopher(TRUE)` function of the bureaucrat pointed to by the director's `itsGopher` data member. This is how the gopher gets set when a new director is

first activated, and when an active director is resumed. `Activate` then sets `gSleepTime` to `0` to force an idle event and calls its supervisor's `ActivateDirector` function.

Deactivate

```
void Deactivate ();
```

A director is becoming inactive. If the director is the gopher, this function calls the supervisor's `BecomeGopher(TRUE)` function to move the gopher outside the inactive director; it then calls the supervisor's `DeactivateDirector` function.

ActivateDirector

```
void ActivateDirector (CDirector *aDirector);
```

A director owned by this director is becoming active. This function calls `CDirectorOwner::ActivateDirector` to make `aDirector` the frontmost director and to set `active` to `TRUE`. Then it calls the supervisor's `ActivateDirector` function to ensure that the entire chain of command up to the application has `active` set `TRUE`.

DeactivateDirector

```
void DeactivateDirector (CDirector *aDirector);
```

A director owned by this director is becoming inactive. This function calls `CDirectorOwner::DeactivateDirector` to set `active` to `FALSE`, then calls the supervisor's ActivateDirector function to ensure that the entire chain of command up to the application has `active` set `FALSE`.

RemoveDirector

```
void RemoveDirector (CDirector *aDirector);
```

Remove a director from the directors list.

Suspend

```
void Suspend ();
```

Suspends the application. This function calls `CDirectorOwner::Suspend` to suspend all of the directors supervised by this director. If this director is active, and if it owns an active window, it calls the window's `Deactivate` function. The director remains active even though the application is suspended.

Resume

```
void Resume ();
```

The application is being resumed. This function calls `CDirectorOwner::Resume` to resume all of the directors supervised by this director. If the director is active and the director owns a window, the function calls the window's `Activate` function. The window, in turn, calls the director's `ActivateWind` function, which calls `Activate` for the director.

Close

```
Boolean Close (Boolean quitting);
```

Closes a director. This function calls `CDialogDirector::Close` to close all of the directors supervised directly or indirectly by this director. If all of the directors are closed, `Close` returns `TRUE`, otherwise it returns `FALSE`.

Windows

CloseWind

```
void CloseWind (CWindow *theWindow);
```

`CloseWind` calls `Close(FALSE)` if `theWindow` is `itsWindow` or if the director has no windows and supervises no other directors. If `theWindow` is not the window owned by this director, `CloseWind` calls that window's virtual destructor.

`CloseWind` is called by a window's `Close` function, which in turn is called when the user clicks in the window's close box. If a click in the window's close box is to mean something other than closing the director, override this function. For instance, you might want to hide the window instead of closing it. See the `CloseWind` function in Chapter 32, "CClipboard."

ActivateWind

```
void ActivateWind (CWindow *theWindow);
```

`ActivateWind` is called by the director's window when the window is activated. For example, if the user clicks in the window, it calls `Activate`. To perform specific actions at activate time, override `Activate`.

DeactivateWind

```
void DeactivateWind (CWindow *theWindow);
```

`DeactivateWind` is called by the director's window when the window is becoming inactive, for example, if the user clicks in another window it calls `Deactivate`. To perform some specific actions at deactivate time, override `Deactivate`.

IsActive

```
Boolean IsActive ();
```

Returns `TRUE` if the director is active; `FALSE` otherwise. You should not override this function.

ChangeName

```
void ChangeName(str 255 newname);
```

Changes the window name to `newName`

Change notification

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

One of this director's providers or subordinates has just changed. This function handles the reason `bureaucratIsGopher` by updating the data member `itsGopher`. If your derived class handles other reasons, it should override this function and call `CDirector::ProviderChanged` to deal with reasons your derived class doesn't handle.

`aProvider` is the provider or subordinate that changed. In this case, it is the bureaucrat that is becoming the gopher. `reason` is a `long` that describes the type of change. `bureaucratIsGopher` is the only reason handled by this function. `info` is a pointer to any additional information that a derived class might need.

For more information, see Chapter 33, "CCollaborator."

Apple events

DoCloseEvent

```
void DoCloseEvent (CAppleEvent *theEvent,
   AEDesc *theResult);
```

Acts on the close Apple event. Calls `CloseWind` on `itsWindow`.

Clone

CloneWindow

```
void CloneWindow (cWindow *aWindow,
   CAppleEvent *theEvent, AEDesc *result);
```

Responds to clone window Apple event. Derived class must override for anything to be done.

44 CDirector

CDirectorOwner 45

Heritage

Base Class	CBureaucrat
Derived Classes	CApplication
	CDirector

Using CDirectorOwner

CDirectorOwner is an abstract class for objects that own directors. A director is an object that manages the communication between an application and a window.

CDirectorOwner has two derived classes: CApplication and CDirector. An application needs to inform its directors when the application is suspending, resuming, or quitting. If your application implements multi-window documents, use an object of class CDocument as the main document and objects of class CDirector as the subwindows. The subwindows should be owned by the document object.

Data Members

CDirectorOwner defines the following data members.

Data member	Type	Description
`itsDirectors`	`CDirectorList*`	List of the directors this object owns
`active`	`Boolean`	`TRUE` if any director is active

Member Functions

Creation and destruction

CDirectorOwner

```
CDirectorOwner (CDirectorOwner
   *aSupervisor = NULL);
```

Constructor. This function sets `itsDirectors` to `NULL`.

~CDirectorOwner

```
~CDirectorOwner ();
```

Destructor. Dispose of all the directors this object owns.

IDirectorOwner

```
void IDirectorOwner (CDirectorOwner
   *aSupervisor);
```

Initializes the owner. This member function sets `itsDirectors` to `NULL`. Provided for backward compatibility. Must not be called if constructor was called with arguments.

Insertion and deletion

AddDirector

```
void AddDirector (CDirector *aDirector);
```

Adds a new director to the director list.

RemoveDirector

```
void RemoveDirector (CDirector *aDirector);
```

Removes a director from the director list.

Appearance

ActivateDirector

```
void ActivateDirector (CDirector *aDirector);
```

Called when a director that this object owns has been activated. Sets `active` to `TRUE`.

DeactivateDirector

```
void DeactivateDirector (CDirector *aDirector);
```

Called when a director that this object owns has been deactivated. Sets `active` to `FALSE`.

Suspend

```
void Suspend (void);
```

Calls the `Suspend` function of all the object's directors. Notifies this object's directors that the application is suspending.

Resume

```
void Resume ( );
```

Calls the `Resume` function of all the object's directors. Notifies this object's directors that the application is resuming.

Quit

```
Boolean Quit ( );
```

The application is about to quit. Calling this member function is the same as calling `Close(TRUE)`.

Close

```
Boolean Close (Boolean fQuitting);
```

Attempts to close all the directors that this object owns. If they all close, this member function returns `TRUE`. If any director doesn't close, this member function stops closing directors and returns `FALSE`. Set `fQuitting` to `TRUE` if you are quitting the application.

45 CDirectorOwner

CDLOGDialog 46

Introduction

CDLOGDialog creates a dialog window from `'DLOG'` and `'DITL'` resources. It represents standard dialog items by their closest analogs in the THINK Class Library.

Heritage

Base Class	CDialog
Derived Class	None

Using CDLOGDialog

CDLOGDialog is a derived class of CDialog that creates a dialog window from `DLOG` and `DITL` resources. It reads the items from the `DITL` resource and converts them into the analogous pane types.

After it creates the window, CDLOGDialog creates a CPanorama to act as a background for all the dialog items. If you specify horizontal or vertical scroll bars, it also creates a CScrollPane. The dialog must have both scrollbars for auto-scrolling to be enabled. CDLOGDialog examines the window `procID` specified in the `'DLOG'` resource to determine whether the dialog should have scroll bars. If the `procID` is `documentProc` or `zoomDocProc`, the dialog has vertical and horizontal scroll bars and a size box; otherwise, it has none.

CDLOGDialog can create pane types that have no analog in the dialog manager. If a `'DITL'` item is static text and begins with `@`, CDLOGDialog passes the text to `AddOverloadedItem`, which interprets it as a pane type.

Data Members

CDLOGDialog defines the following protected data members:

Data member	Type	Description
staticTextFont	short	Default font for static text items
staticTextSize	short	Default size for static text items
editTextFont	short	Default font for edit text items
editTextSize	short	Default size for edit text items
defaultBorderPen	short	Default border thickness for user items and CRadioGroupPanes

Member Functions

Creation and destruction

CDLOGDialog

```
CDLOGDialog (short DLOGid,
   CDirector *aSupervisor);
```

Constructor. Initialize a dialog window from the `'DLOG'` resource `DLOGid`, and create the dialog items from its associated `'DITL'` resource. `aSupervisor` is the dialog's supervisor. The enclosure is always `gDesktop`.

This member function creates a panorama object to act as a background for all the dialog items by calling `MakePanorama`. It gives the dialog scroll bars if the dialog's `procID` is `documentProc` or `zoomDocProc`. It adds the dialog items by calling `AddDITLItems`.

CDLOGDialog

```
CDLOGDialog ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with IDLOGDialog for backwards compatibility.

CDLOGDialogX

```
void CDLOGDialogX();
```

A private member function called by CDLOGDialog.

IDLOGDialog

```
void IDLOGDialog (short DLOGid,
   CDesktop *anEnclosure,
   CDirector *aSupervisor);
```

Initialization member function for backward compatibility. May not be called if constructor has arguments. `anEnclosure` should always be `gDesktop`. The other arguments are the same as for the constructor.

IDLOGDialogX

```
void IDLOGDialogX (short DLOGid, CDirector
   *aSupervisor);
```

Private initialization function called by IDLOGDialog and provided for backward compatability.

AddDITLItems

```
void AddDITLItems (short DITLid, long baseID);
```

Creates panes for the dialog items in the `'DITL'` resource `DITLid`. This function calls a different member function for each type of item. If the item is enabled, it calls `SetsWantsClicks` so it can respond to mouse clicks. To change how your derived class creates an item, override the appropriate member function of that item.

This table shows the panes that this function creates. For more information on how it creates an item, see the documentation on the member function it calls.

For this item…	It creates…	By calling…
User item	CPane	`AddDITLUserItem`
Push button	CButton	`AddDITLPushBtn`
Check box	CCheckBox	`AddDITLCheckBox`
Radio button	CRadioControl	`AddDITLRadioBtn`
Static text	CEditText, CRadioGroupPane, CIntegerText, CDialogText, or some other pane	`AddDITLStatText`
Edit text	CDialogText	`AddDITLEditText`
Icon	CIconPane	`AddDITLIcon`
Picture	CPicture	`AddDITLPicture`
Resource control	Nothing	`AddDITLResControl`

Table 46-1 How AddDITLItems creates dialog items

FindEnclosingView

```
CView *FindEnclosingView (Rect *boundsRect);
```

Returns the view that completely encloses the rectangle `Rect`. `Rect` must be in window coordinates. CDLOGDialogs lets you nest the pane for one dialog item within another. If no dialog item completely encloses `Rect`, this function returns the dialog.

Dialog item creation member functions

AddDITLPushBtn

```
CPane *AddDITLPushBtn (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem);
```

Adds a push button to the dialog by creating a CButton.

AddDITLRadioBtn

```
CPane *AddDITLRadioBtn (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem,
   long anID);
```

Adds a radio button to the dialog by creating a CRadioControl.

AddDITLCheckBox

```
CPane *AddDITLCheckBox (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem);
```

Adds a check box to the dialog by creating a CCheckBox.

AddDITLResControl

```
CPane *AddDITLResControl (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem);
```

Adds a resource control to the dialog. This function is a stub that does nothing. If your derived class handles resource controls, it must override this function.

AddDITLStatText

```
CPane *AddDITLStatText (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem);
```

Adds a static text item or an overloaded item to the dialog. If the item text begins with the character `@`, it is an overloaded item and

the rest of the text is passed to the `AddOverloadedItem` member function. If the item starts with any other character, it is a static text item and this function creates a CEditText object. It sets the font and font size for the item to `editTextFont` and `editTextSize`. It also sets the texts' attributes to `kNotEditable`, `kNotSelectable`, and `kNotStylable`.

AddDITLEditText

```
CPane *AddDITLEditText (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem);
```

Adds an editable text item to the dialog by creating a CDialogText. It sets the font and font size for the item to `editTextFont` and `editTextSize`.

AddDITLIcon

```
CPane *AddDITLIcon (short aWidth, short aHeight,
   short hEncl, short vEncl, CView *enclosure,
   tDITLItem *ditlItem);
```

Adds an icon to the dialog by creating a CIconPane.

AddDITLPicture

```
CPane *AddDITLPicture (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem);
```

Adds a picture to the dialog by creating a CPicture.

AddDITLUserItem

```
CPane *AddDITLUserItem (short aWidth,
   short aHeight, short hEncl, short vEncl,
   CView *enclosure, tDITLItem *ditlItem);
```

Adds a user item to the dialog. It creates a CPane object with a border of `defaultBorderPen` thickness.

AddOverloadedItem

```
CPane *AddOverloadedItem (StringPtr itemText,
   short aWidth, short aHeight, short hEncl,
   short vEncl, CView *enclosure,
   tDITLItem *ditlItem);
```

Add a pane to the dialog that has no analogue in the Dialog Manager. An overloaded item is a static text item that begins with @. This function supports these types:

To create…	Set the text to…
Any `Pane` resource	@*$ClassName*$*ResType*$*ResID* For example, `@$CPopupMenuPane$Popm$1000` creates a CPopupMenuPane from the template in the resource `'Pop' #1000`. The size and location of the pane are taken from the dialog item `rect`, not from the resource.
CRadioGroupPane	`@RadioGroupPane`
CIntegerText with *no* limits	`@#`
CIntegerText with limits	`@#`*min*`#`*max* For example, `@#100#200` creates a field that must be between 100 and 200.
Required CDialogText	`@!!`
CDialogText with maximum length	`@!`*num* For example, `@!10` creates a field with a maximum length of 10, but that's not required.
Required CDialogText with maximum length	`@!!`*num* For example, `@!!10` creates a required field with a maximum length of 10.

Table 46-2 Overloaded items

To create other types of overloaded items, override this function in your derived class.

Member Functions: Private

CDLOGDialogX

```
void CDLOGDialogX ();
```

Performs common initialization.

IDLOGDialogX

```
void IDLOGDialogX (
   short DLOGid,
   CDirector *aSupervisor);
```

Initialization function called by IDLOGDialog and provided for backward compatibility.

CDLOGDirector 47

Introduction

CDLOGDirector is a dialog director that creates its dialog from `'DLOG'` and `'DITL'` resources.

Heritage

Base Class	CDialogDirector
Derived Class	none

Using CDLOGDirector

CDLOGDirector is a derived class of CDialogDirector. It creates and manages a dialog object of type CDLOGDirector, which creates a dialog from `'DLOG'` and `'DITL'` resources.

You can implement a simple dialog from your application class with CDLOGDirector. For example:

```
CDLOGDirector dialog(kMyDLOGid, this);

long result = dialog.DoModalDialog(cmdOK);
if (result == cmdOK)
{
   // do something if user OK'd the dialog
}
else if (result == cmdCancel)
{
   // do something if user canceled the dialog
```

If your dialog needs to handle any commands other than `cmdOK` and `cmdCancel`, you need to derive a class from CDLOGDirector and override `DoCommand`.

Data Members

CDLOGDirector defines no data members.

Member Functions

Creation and Destruction

CDLOGDirector

```
CDLOGDirector (short DLOGid,
   CDirectorOwner *aSupervisor);
```

Constructor. Creates a CDLOGDialog from a `'DLOG'` resource `DLOGid`.

CDLOGDirector

```
CDLOGDirector ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with IDLOGDirector for backward compatibility.

IDLOGDirector

```
void IDLOGDirector (short DLOGid,
   CDirectorOwner *aSupervisor);
```

Initialization function may not be called if constructor has arguments.

Member Functions: Private

IDLOGDirectorX

```
void IDLOGDirectorX (short DLOGid);
```

Performs common initialization.

CDocument 48

Introduction

CDocument is the main class for presenting and manipulating information. You can think of a document as the association of a window, a file, and a set of panes.

The CSaver document class uses object I/O to automatically read and write object-based documents. If you do not use CSaver, your application must create a derived class of CDocument.

Heritage

Base Class	CDirector
Derived Classes	CAppleEventObject
	CSaver <template>

Using CDocument

You can think of a document as a file that you view through a window. A document can contain anything that you can display and manipulate inside a window.

The document is where your application draws and displays its data. Since documents are descendants of CDirector, all documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file is created automatically. You must create them yourself by overriding the appropriate member functions.

In addition to defining its own constructor and destructor, your document class should override these member functions:

```
DoCommand          DoSave
NewFile            DoSaveAs
OpenFile           Revert
```

Your document class must have a constructor. If your derived class defines new data members, the constructor sets them up.

If your application allocates memory, you should also define a destructor to deallocate it.

Note

You do not need to delete the window or file associated with a document. The virtual destructor does that for you.

Your document class's `DoCommand` function does most of the work in your application. When a window is active, the switchboard sends all commands to the gopher; if the gopher can't "handle" it, a command is passed up the chain of command to the document. If the document can't handle it, the application class tries to handle it. Your document class should handle all the commands it knows about, and call the inherited `DoCommand` function when it can't.

Usually, a command that changes a document is performed by creating an appropriate task object (derived from CTask) and calling `Notify` with the task as an argument. When `Notify` reaches a director, the director calls `SetChanged(TRUE)` (which sets its `dirty` data member to `TRUE`). The document later decides whether it needs to be saved by calling `GetChanged`. Because the `dirty` flag is managed automatically as a side effect of `Notify`, you seldom need to think about it. However, if some document command changes the state of a document without creating and notifying a task, and only in this circumstance, your command function needs to call `SetChanged(TRUE)` to ensure that the document is saved.

If a command that changes a document is undone and if the `dirty` flag was set for the first time as a result of the command, the director's `ToggleChanged` function resets the `dirty` flag. Do not call `SetChanged(TRUE)` if you do not need to, as this interferes with the operation of `ToggleChanged`.

When the user chooses **New** from the **File** menu, your application's `CreateDocument` function is called. `CreateDocument` should allocate a new document object and call the document's `NewFile` function. `NewFile` must create a window by calling

MakeNewWindow. NewFile does not need to create a file; this is done when the user first tries to save the document.

When the user chooses **Open** from the **File** menu, your application's OpenDocument function is called. OpenDocument should allocate a new document object and call the document's OpenFile function. OpenFile has one argument: a Macintosh SFReply record. When OpenFile is called, you can be sure that the SFReply record is properly filled in. Your OpenFile function needs to create an instance of a file object (usually of class CDataFile). To facilitate revert processing, it helps to do the actual file reading in a separate ReadDocument member function. ReadDocument can call any of several read functions on your file to get its contents. ReadDocument also needs to create a window to display the contents of the file, just as your NewFile function does, by calling MakeNewWindow. Finally, ReadDocument should create any panes in the window required to represent the contents of the file.

Your MakeNewWindow function should create a new, empty window and store a pointer to the window in the document's itsWindow data member.

When the user chooses **Save** from the **File** menu, your document's DoSave function is called. Your DoSave function should write the contents of its file to disk. The file object is stored in the data member itsFile. If the document does not currently have a file, your DoSave function should call SaveAs to ask the user for a new file name.

When the user chooses **Save As** from the **File** menu, your document's DoSaveAs function is called. This function takes an SFReply record, which is already filled in. Your DoSaveAs function should create a new file object (usually of class CDataFile) and store a pointer to the file in the document's itsFile data member. Once this is done, DoSaveAs can call DoSave to write the document to disk. If the user chose **Save** from the **File** menu to create a new file, the call to DoSave is recursive. This is not a problem, because the document now has a file and DoSave proceeds to write to it.

To support the **Revert** command, your application should override the `DoRevert` member function. Your implementation might close the file without saving, and then call `ReadDocument` to open the file again and recreate the window.

Global Variables and Data Members

Global variables

The global variable `gGopher` points to the current bureaucrat. When a window becomes active, `gGopher` points to the pane pointed to by the document's `itsGopher` data member or to the document itself. When there are no active documents, `gGopher` points to the application.

The supervisor of every document is the application.

Global variable	Type	Description
`gGopher`	`CBureaucrat*`	The current bureaucrat
`gApplication`	`CApplication*`	The application
`gWatchCursor`	`CursHandle`	Watch cursor for waiting
`gBartender`	`CBartender`	Manages all menus

Data members

CDocument defines the following data members:

Data member	Type	Description
`itsMainPane`	`CPane*`	The document's main pane. `NIL` if the document has no main pane. The main pane's enclosure should be `itsWindow`, a data member inherited from CDirector.
`itsFile`	`CFile*`	The file associated with this document. `NIL` if document has no file.
`itsPrinter`	`CPrinter*`	The printer object associated with this document. `NIL` if document is not printable.
`pageWidth`	`short`	The width of a page.
`pageHeight`	`short`	The height of a page.
`savePrintPane`	`CPane*`	Saved print pane while a subdirector is printing.

Data member	Type	Description
savePrinter	CPrinter*	Saved printer while a subdirector is printing.
askToSave	Boolean	TRUE if the user is always asked whether to save dirty documents.
SaveOption	static DescType	One of the following: kAEAsk (ask before saving dirty documents), kAEYes (save without asking), kAENo (don't save).
SaveType	OSType	File type to use when saving a document.

Member Functions

Creation and destruction

CDocument

```
CDocument(Boolean printable);
```

Constructor. If the value of printable is TRUE, CDocument calls MakePrinter to create an instance of CPrinter and stores it in the itsPrinter data member. The supervisor of a document is always gApplication.

CDocument

```
CDocument ();
```

Default constructor. Implicitly called when object is created by new_by_name. Can also be used in combination with IDocument for backward compatibility.

~CDocument

```
~CDocument ();
```

Destructor. Deletes itsFile and itsPrinter if they were allocated.

IDocument

```
void IDocument(CApplication *aSupervisor,
   Boolean printable);
```

Initialization function provided for backward compatibility. Must not be called if constructor has arguments.

Commands

DoCommand

```
void DoCommand (long theCommand);
```

Handles a document-related command. The default function handles these commands:

Command	Action
cmdClose	Calls Close.
cmdSave	Sets the cursor to the watch cursor and calls DoSave.
cmdSaveAs	Calls DoSaveFileAs. The default DoSaveFileAs function sets the cursor to a watch, lets the user enter a file name by using a standard file dialog, and calls DoSaveAs.
cmdRevert	Displays a "Do you really want to revert?" alert. If the user responds OK, sets the cursor to the watch cursor and calls DoRevert.
cmdPageSetup	If the document is printable, calls DoPageSetup.
cmdPrint	If the document is printable, calls DoPrint.

Table 48-1 How CDocument handles a command

Your document class usually overrides DoCommand. You should handle your own commands first, and then call CDocument::DoCommand to handle the default commands.

UpdateMenus

```
void UpdateMenus ();
```

Updates the menu items just before they appear on the screen. The default function enables the following commands:

Command	Enabled if…
cmdSaveAs	Always
cmdSave	The document has changed
cmdRevert	A file is associated with the document and it has changed
cmdPageSetup	The document is printable
cmdPrint	The document is printable

Table 48-2 How CDocument updates menus

Your document class should override this function to enable the appropriate commands for your document. Be sure you call `CDocument::UpdateMenus` *before* you enable your document's commands.

Appearance

Close

```
Boolean Close (Boolean quitting);
```

Calls the document's `ConfirmClose` function. If the result is `TRUE`, it calls the `Close` function of `itsFile` (if it has one). The `quitting` parameter tells the `ConfirmClose` function whether to ask to save before "closing" or "quitting." If the document was actually closed, this function returns `TRUE`. Otherwise it returns `FALSE`.

ConfirmClose

```
Boolean ConfirmClose (Boolean quitting);
```

Displays a "Save before closing?" or "Save before quitting?" dialog. If the user answers yes, this function sends a `DoSave` to the document and returns `TRUE`. If the user answers no, it returns `TRUE`. If the user answers Cancel, it returns `FALSE`.

File creation

NewFile

```
void NewFile ();
```

Opens a new file. Your application's `CreateDocument` function should create a new document and call `NewFile`. Your document class should override `NewFile` to call `MakeNewWindow` in order to create a new, empty window.

OpenFile

```
void OpenFile (SFReply *macSFReply);
```

Opens an existing file. Your application's `OpenDocument` function should create a new document object and call `OpenFile`. Your document class should override `OpenFile` and do the following:

- Call `MakeNewWindow` to create a new, empty window.
- Open the file specified in `macSFReply` and assign it to `itsFile`.
- Call `ReadDocument` to read the file.

Note

To learn how to specify a file, see the class CFile. To learn how to read and write from a data file, see the class CDataFile.

MakeNewWindow

```
void MakeNewWindow ();
```

Creates a new, empty window. You must override this function to do the following:

- Create a new window and assign it to `itsWindow`.
- Create the panorama and assign it to `itsMainPane`.

ReadDocument

```
void ReadDocument ();
```

Reads the file and displays its contents in the window. You must override this function.

Printing

MakePrinter

```
void MakePrinter ();
```

Makes the printer object for this document. For more information on printer objects, see CPrinter. If you create a derived class of CPrinter, you should override this function to create an object of your class.

SetupPrinter

```
void SetupPrinter ();
```

Sets up the printer parameters and calls `MakePrinter`.

Paginate

```
void Paginate ();
```

Calls the `Paginate` function of `itsMainPane` if it is not `NULL`. Otherwise, calls the `ResetPagination` function of `itsPrinter`.

PageCount

```
short PageCount ();
```

Returns the number of pages in the document. This function calls the `GetStripCount` function of `itsPrinter` to find out how many pages the document occupies. If the document has more than 999 pages, `PageCount` raises an exception.

If the document doesn't have a CPrinter object associated with it, this function returns 0 (zero).

AboutToPrintSubDirector

```
void AboutToPrintSubDirector (CPane *aPrintPane,
    CPrinter *aPrinter);
```

Sets which pane to print and which printer to use for the next print command, instead of the default `itsMainPane` and `itsPrinter`. A subdirector of this document should call this function before passing a `cmdPrint` or `cmdPageSetup` command. The subdirector must call `DonePrintingSubDirector` when it's finished printing.

A subdirector is a director that another director owns. For example, if your application supports multiple views of a document, the document owns a director for each view.

DonePrintingSubDirector

```
void DonePrintingSubDirector();
```

Restores the values of the document's `itsMainPane` and `itsPrinter` data members to the values they had before you called `AboutToPrintSubDirector`.

AboutToPrint

```
void AboutToPrint (short *firstPage,
   short *lastPage);
```

Checks the range of pages to be printed. The default function changes `lastPage` to be equal to `PageCount`. Your document class should override this function to do whatever is appropriate for your application. This is the place where the document can request information about the page size from `itsPrinter`.

PrintPageOfDoc

```
void PrintPageOfDoc (short pageNum);
```

Prints the specified page. The default function calls the `PrintPage` function of `itsMainPane` if it's not `NULL`.

DonePrinting

```
void DonePrinting ();
```

Calls the `DonePrinting` function of `itsMainPane`. The `PrintPageRange` function in CPrinter calls this function when printing is done.

Filing

DoSave

```
Boolean DoSave ();
```

Saves the document under its current name and returns `TRUE` if successful. The current file is available through the data member `itsFile`. Your document class must override this function.

DoSaveAs

```
Boolean DoSaveAs (SFReply *macSFReply);
```

Saves the document under a new name and returns `TRUE` if successful. The `macSFReply` record specifies where to write the file. Your document class must override this function.

DoRevert

```
void DoRevert ();
```

Reverts to the last saved version of this document. Your document class must override this function.

DoSaveFileAs

```
Boolean DoSaveFileAs ();
```

Responds to a **Save As** command and returns `TRUE` if successful. This function calls the document's `PickFileName` function. If you provide a good file name, it calls the document's `DoSaveAs` function. Since this function implements the standard **Save As** command through two other functions, you don't need to override it.

PickFileName

```
void PickFileName (SFReply *macSFReply);
```

Displays a standard file dialog to get a new file name. `PickFileName` uses the `GetName` function to specify the default name.

GetName

```
void GetName (Str255 theName);
```

Returns the name of the document in `theName`. If there is a file associated with the document, this function returns the name of the file. If there is no file associated with the document, but there is a window associated with it, this function returns the title of the window. If there is neither a file nor a window associated with the document, this function returns a null string.

SetAskToSave

```
void SetAskToSave (Boolean ask);
```

Sets the data member `askToSave` equal to `ask`. If `askToSave` is `TRUE`, the user is asked before saving `dirty` documents. Otherwise, they are saved automatically.

SetSaveOption

```
void SetSaveOption (DescType option);
```

Sets the data member `saveOption` equal to `option`, which is one of `kAEAsk`, `kAEYes`, `kAENo`.

Apple Event Support

For information regarding Apple event member functions, please refer to the supplemental information provided in the file `Apple event info` in the `Apple Event Classes` folder.

Class Resources

Resource	Description
`STRprompt 150`	`STR` resource ID for `PickFileName` prompt string
`ALRTrevert 150`	Reverts to saved alert
`ALRTsaveChanges 151`	Saves changes before close/quit alert
`STRtaskNames 130`	`'STR#'` resource ID for task names
`STRcommon 128`	`'STR#'` resource for commonly used strings

CEditText 49

Introduction

CEditText implements a pane that displays text. This class uses the Macintosh TextEdit routines.

Heritage

Base Class	CAbstractText
Derived Classes	CDialogText
	CStaticText
	CStyleText

Using CEditText

Use CEditText whenever you need to display unformatted text. An edit text pane is usually the panorama in a scroll pane, so you can scroll through the text. The `DoCommand` function of an edit text pane handles all the common text-editing commands such as cutting and pasting, font selection, line spacing, and so on. The `Specify` function, inherited from CAbstractText (see Chapter 14), lets you choose whether the user can edit and copy your text pane's text.

CEditText does not use the Styled Text Edit routines described in *Inside Macintosh Volume V*. To create a pane of styled text, use CStyleText.

To make sure that the edit text pane responds to commands, you must place it in the chain of command. The best way to do this is to set the value of your document's `itsGopher` data member to the edit pane.

Because CEditText is based on the Macintosh TextEdit (TE) routines, it has some limitations. It's designed to edit small amounts of text. The maximum number of characters you can store in a CEditText

record is around 32,000, but you notice performance degradation long before it gets that big. To create a pane to display more text, create a derived class of CAbstractText, described in Chapter 14.

Data Members

CEditText defines the following data members:

Data member	Type	Description
macTE	TEHandle	TextEdit record handle
spacingCmd	long	Line spacing command number
alignCmd	long	Alignment command number

Member Functions

Creation and destruction

CEditText

```
CEditText (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth, Boolean aScrollHoriz);
```

Constructor. By default aWidth, aHeight, aHEncl, and aVEncl have a value of 0 (zero). By default, aHSizing and aVSizing have a value of sizELASTIC. Most of the arguments are identical to the CPanorama constructor. aLineWidth specifies how wide the lines should be. If aLineWidth is less than 0 (zero), the width is the same as the Macintosh TextEdit record's viewRect. The default value is -1. aScrollHoriz is TRUE if the text should scroll horizontally when the line width is greater than the frame width and the cursor moves outside the frame. The default value is FALSE.

Note

The other arguments are described in Chapter 74, "CPanorama."

CEditText

```
CEditText();
```

Default constructor. Implicitly called when object is created by new_by_name. Can also be used in combination with IEditText for backward compatibility.

~CEditText

```
~CEditText();
```

Destructor. Disposes of the TextEdit record `macTE`.

IEditText

```
void IEditText (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initialization function provided for backward compatibility. Must not be called if constructor has arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr ViewData);
```

Initializes a VCEditText object from a resource template. The data members are set using the resource template pointed to by `ViewData`. Each derived class of CView overrides this function to use its own resource template.

IEditTextX

```
void IEditText ();
```

Performs common initialization.

MakeMacTE

```
void MakeMacTE ();
```

Creates a TextEdit record and sets `macTE` to it.

Mouse and keystrokes

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handles a mouse down in an edit text pane. This function calls `SelectionChanged` after processing the click.

Commands

PerformEditCommand

```
void PerformEditCommand (long theCommand);
```

Performs the standard cut, copy, paste, and clear commands on the text. The task classes call this function to undo an edit command.

TypeChar

```
void TypeChar (char theChar, short theModifiers);
```

Processes a keystroke. This function does not need to set up for an **Undo** command and should handle the key directly. The task classes call this function to undo key strokes.

Display

Draw

```
void Draw (Rect *area);
```

Draws the text pane.

Scroll

```
void Scroll (long hDelta, long vDelta,
   Boolean redraw);
```

Scrolls the text within the pane by `hDelta` characters and `vDelta` lines.

GetSteps

```
void GetSteps(short *hStep, short *vStep);
```

Returns the scrolling step values appropriate to the text font.

Activate

```
void Activate ();
```

Activates the text pane. This function enables the editing commands and either highlights the selection or shows the text insertion caret.

Deactivate

```
void Deactivate ();
```

Deactivates the text pane. This function disables the editing commands and either unhighlights the selection or hides the text insertion caret.

SetSelection

```
void SetSelection (long selStart, long selEnd,
   Boolean fRedraw);
```

Sets the selection to the range corresponding to character positions `selStart` through `selEnd`.

GetSelection

```
void GetSelection (long *selStart, long *selEnd);
```

Returns the start and end of the current selection.

HideSelection

```
void HideSelection(Boolean hide, Boolean redraw);
```

If `hide` is `TRUE`, hides the current selection or insertion point. Otherwise, shows it.

Text specification

SetTextPtr

```
void SetTextPtr (Ptr textPtr, long numChars);
```

Uses the first `numChars` characters that `textPtr` points to as the text for this abstract text object. This function makes a copy of the text.

GetTextHandle

```
Handle GetTextHandle ();
```

Returns a handle to the text of the CEditText object. This function returns a handle to the actual text, not to a copy of the text.

CopyTextRange

```
Handle CopyTextRange (long start, long end);
```

Returns a copy of the range of text specified by `start` and `end`.

InsertTextPtr

```
void InsertTextPtr (Ptr text, long length,
   Boolean fRedraw);
```

Inserts a copy of the given text and length at the start of the current selection. If `fRedraw` is `TRUE`, redraws the pane at the next update event.

CheckInsertion

```
void CheckInsertion (long insertLen,
   Boolean useSelection);
```

Checks whether an insertion of `insertLen` characters would exceed TextEdit's capacity. If `useSelection` is `TRUE`, this function deducts the size of the selection from the total length. If the insertion would fail, this function calls `Failure`.

Text characteristics

SetFontNumber

```
void SetFontNumber (short aFontNumber);
```

Sets the font for this text pane by font number.

SetFontStyle

```
void SetFontStyle (short aStyle);
```

Changes the font style for this text pane. `aStyle` may be one or any additive combination of: `normal`, `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, or `extend`.

SetFontSize

```
void SetFontSize (short aSize);
```

Sets the font size for this text pane to the specified size.

SetTextMode

```
void SetTextMode (short aMode);
```

Sets the text mode for this text pane to the specified mode. `aMode` can be set to: `srcCopy`, `srcOr`, `srcXor`, `srcBic`, `notSueCopy`, `notSrcOr`, `notSrcXou`, or `notSrcBic`.

SetAlignCmd

```
void SetAlignCmd (long anAlignCmd);
```

Sets the text alignment for this text pane. This function uses the THINK Class Library's names for the alignment choices: `cmdAlignLeft`, `cmdAlignRight`, or `cmdAlignCenter`. To use TextEdit's alignment names, call `SetAlignment`.

GetAlignCmd

```
long GetAlignCmd ();
```

Returns the current alignment for this text pane. It can be set to `cmdAlignLeft`, `cmdAlignRight`, `cmdAlignCenter`, or `cmdNull`.

SetAlignment

```
void SetAlignment (short anAlignment);
```

Sets the text alignment for this text pane. This function uses `TextEdit`'s names for these alignment choices: `teFlushDefault`, `teFlushLeft`, `teCenter`, or `teFlushRight`. To use the THINK Class Library's alignment commands, call `SetAlignCmd`.

SetSpacingCmd

```
void SetSpacingCmd (long aSpacingCmd);
```

Sets the space between lines of text. `aSpacingCmd` can be set to `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`.

GetSpacingCmd

```
long GetSpacingCmd ();
```

Returns the space between lines of text. This function can be set to `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`.

GetTEFontInfo

```
void GetTEFontInfo (FontInfo *macFontInfo);
```

Returns a QuickDraw `FontInfo` record for the font that this text pane uses. As a side effect, this member function changes the font, size, and style of the pane's port to that used by the `TextEdit` record.

GetHeight

```
long GetHeight (long startLine, long endLine);
```

Returns the total height in pixels of the indicated lines of text.

GetCharOffset

```
long GetCharOffset (LongPt *aPt);
```

Returns the character position nearest the coordinate `aPt`. `aPt` must be in frame coordinates.

GetCharPoint

```
void GetCharPoint( long offset, LongPt *aPt);
```

Returns the coordinate of the character position `offset`. `aPt` is in frame coordinates.

GetCharStyle

```
void GetCharStyle (long charOffset,
   TextStyle *theStyle);
```

Returns style information for the character at position `charOffset`.

GetTextStyle

```
void GetTextStyle (short *whichAttributes,
   TextStyle *aStyle);
```

Returns current style information for this text pane. `whichAttributes` is a flag that indicates which text attributes to report on. The attribute flags and `TextStyle` record are described in *Inside Macintosh VI*, Chapter 15, "TextEdit."

Calibrating

ResizeFrame

```
void ResizeFrame (Rect *delta);
```

Resizes the text's frame when the size of its pane changes. The `delta` rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left.

CalcTERects

```
void CalcTERects ();
```

Determines the destination and view rectangles for this text pane's `TextEdit` record.

AdjustBounds

```
void AdjustBounds ();
```

Adjusts the bounds of this text pane to match its TextEdit record. When you do something that could change the line width or the number of lines, call this function.

FindLine

```
short FindLine (long charPos);
```

Returns the line number containing the specified character position. Both line and character numbering start at 0 (zero). If the character position is before the start of the text (a negative number), this function returns 0 (zero). If the character position is beyond the end of the text, this function returns the number of the last line.

GetLength

```
long GetLength ();
```

Returns the length in bytes of this text pane's text buffer.

GetNumLines

```
long GetNumLines ();
```

Returns the total number of lines in this text pane's text buffer.

Printing

AboutToPrint

```
void AboutToPrint (short *firstPage,
   short *lastPage);
```

When the specified range of pages is about to be printed, this function deactivates the text pane to unhighlight the current selection.

PrintPage

```
void PrintPage (short pageNum, short pageWidth,
   short pageHeight, CPrinter *aPrinter);
```

Prints the specified page.

DonePrinting

```
void DonePrinting ();
```

When printing is over, this function rehighlights the current selection if the edit pane is active.

Cursor

Dawdle

```
void Dawdle (long *maxSleep);
```

This function flashes the insertion point when the edit text pane is active. This function sets `maxSleep` to the value of `GetCaretTime`, which is the rate at which the insertion point blinks. Setting this value ensures that `WaitNextEvent` generates a null event at least that often.

Note

To learn more about "sleep time," see the description of the `Dawdle` function in Chapter 27, "CBureaucrat."

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Private

CEditTextX

```
void CEditTextX ();
```

A private member function called by the constructor to initialize CEditText.

CEnvironment 50

Introduction

CEnvironment maintains a drawing environment for any pane.

Heritage

Base Class	None
Derived Classes	CTextEnvirons

Using CEnvironment

Every pane has a data member `itsEnvironment` that points to its drawing environment. If `itsEnvironment` is non-`NULL`, the pane's `Prepare` function calls `itsEnvironment->Restore`() to set up its QuickDraw drawing environment.

You can use this class to make sure that the drawing environment is set up correctly. Or you might want to change the drawing environment according to some saved settings.

Data Members

This class has no data members.

Member Functions

Creation and Destruction

Dispose

```
void Dispose ();
```

Provided for backward compatibility. Calls `operator delete`. This function can be used only if TCL_USE_DISPOSE is defined for the project.

Environment

Restore

```
void Restore ();
```

Restores the drawing environment in the current port. The default function just calls the QuickDraw routine `PenNormal`.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Friend Functions

operator <<

```
friend CStream operator << (CStream &s,
   CEnvironment *p);
```

Writes the environment to the stream `s`.

operator >>

```
friend CStream operator >> (CStream &s,
   CEnvironment *p);
```

Reads the environment from the stream `s`.

CError 51

Introduction

CError is an error-handling class. You can use the global error handler object `gError` to report errors.

Most error handling in the THINK Class Library uses the exception handling mechanism described in Chapter 9, “Exception Handling and RTTI.”

Heritage

Base Class	None
Derived Classes	None

Using CError

The application constructor `CApplication` creates an instance of CError and stores a pointer to it in the global variable `gError`.

The CError member function `PostAlert` provides a convenient way to show an alert displaying a string from a `STR#` resource.

Global Variables and Data Members

Global variables

There is one global CError object:

Variable	Type	Description
`gError`	`CError`	Global error handler

Data members

This class has no data members.

Member Functions

Error reporting

PostAlert

```
void PostAlert (short STRid, short index);
```

Posts a general alert and return. `STRid` is the resource ID of a `'STR#'` resource. `index` is the index into the `'STR#'` resource. The string is displayed in a generic alert box.

MissingResources

```
void MissingResources ();
```

Posts an alert announcing that the application can't find its resources, and exits the application. The alert suggests that your project doesn't have an associated resource file. The alert can't be in a resource, since this function is called only if the resource file is unavailable.

Global Function

Note that a global function is not a member function.

GrowZoneFunc

```
pascal long GrowZoneFunc (Size bytesNeeded);
```

The application member function `InitMemory` installs this function as the application's `GrowZone` function, which is called in low-memory conditions. This function calls the application's `GrowMemory` member function. See Chapter 18, "CApplication."

Class Resource

Resource	Description
`'ALRT'/'DITL' 128`	Generic alert box used by `PostAlert`. Contains `^0` for use with `ParamText`.

CException 52

Introduction

When the THINK Class Library throws an exception, the thrown exception object is of class CException. The simplified exception-handling macros also throw exceptions of this class. See Chapter 9, “Exception Handling and RTTI,” for a thorough description of exception handling.

Heritage

Base Class	None
Derived Classes	None

Note

CException is declared a `TCL_EXCEPTION_CLASS`, which, in the current implementation of exception handling, causes it to be derived from _BR_Exception. When exception handling is supported by the compiler, this derivation no longer exists, because the deliberately undocumented class _BR_Exception is removed from the THINK Class Library.

Using CException

The THINK Class Library throws exceptions by calling the `Failure` function, described in Chapter 9, “Exception Handling and RTTI.” The exception objects thrown by `Failure` are of class CException. The simplified exception-handling macros also throw exceptions of this class.

Data Members

CException defines these private data members:

Data member	Type	Description
`err`	`short`	An error code, usually an `OSErr` value returned by a Toolbox function.
`msg`	`long`	A resource ID of a `'STR#'` resource, and an index into that resource, packed into a single long value. See the description of `ErrorAlert` in Chapter 127, "TCL Utilities," to learn how this value is interpreted.

Member Functions

Creation and destruction

CException

```
CException (short err, long msg);
```

Constructor. Sets this object's `err` and `msg` data members equal to the corresponding arguments.

Error reporting

GetErr

```
short GetErr ();
```

Returns `err`.

GetMsg

```
long GetMsg ();
```

Returns `msg`.

SetErr

```
void SetErr (short err);
```

Sets this object's `err` data member equal to the parameter `err`.

SetMsg

```
void SetMsg (long msg);
```

Sets this object's `msg` data member equal to the parameter `msg`.

52 CException

CFile 53

Introduction

CFile is an abstract class for implementing classes that deal with disk files.

Heritage

Base Class	None
Derived Classes	CDataFile
	CResFile

Using CFile

CFile is an abstract class for dealing with Macintosh disk files. Most often you use a CDataFile derived class to work with regular data files.

Before you open a file, you must specify it. Specifying a file means identifying it to the Macintosh File Manager. This class gives you three ways to specify a file depending on the type of information you provide. The most common specification function, `SFSpecify`, lets you use an `SFReply` record from the standard file dialogs to identify a file.

Some CFile member functions use the THINK Class Library's exception-handling library ifa problem occurs during in a file operation. For more information on exception handling, see Chapter 9, "Exception Handling and RTTI."

Data Members

CFile defines these data members:

Data member	Type	Description
`name`	`Str63`	File name
`volNum`	`short`	Volume containing the file
`dirID`	`long`	Directory within the volume

Member Functions

Creation and destruction

CFile

```
void CFile ();
```

Constructor.

~CFile

```
~CFile ();
```

Destructor.

IFile

```
void IFile ();
```

Initialization function for backward compatibility. Does nothing.

Dispose

```
void Dispose ();
```

Disposal function for backward compatibility. Calls operator `delete this`. `TCL_USE_DISPOSE` must be defined to use this function.

Specification

Specify

```
void Specify (Str63 aName, short aVolNum);
```

Specify a file by its name and volume reference number. Use this function to specify files on MFS volumes.

SpecifyHFS

```
void SpecifyHFS (Str63 aName, short aVolNum,
   long aDirID);
```

Specify a file by its name, volume number, and directory ID. If the specification is for an alias, this function resolves it.

SpecifyFSSpec

```
void SpecifyFSSpec (const FSSpec *aFileSpec);
```

Specifies the file using a File Manager `FSSpec` record. If the specification is for an alias, this function resolves it.

SFSpecify

```
void SFSpecify (SFReply *macSFReply);
```

Specifies a file from the information in a `macSFReply` record. If the specification is for an alias, this function resolves it. Use this function to specify a file that the user selects through a Standard File dialog.

ResolveFileAlias

```
void ResolveFileAlias ();
```

If the current file specification is an alias, this function resolves it.

Open and close

Open

```
void Open (SignedByte permission);
```

Opens the file with the specified permission. This function does nothing. Derived classes must override this function. For an example of how to write an `Open` function, see Chapter 38, “CDataFile,” and other classes derived from this class.

Close

```
void Close ();
```

Closes this file. This function does nothing. Derived classes must override this function.

IsOpen

```
Boolean IsOpen ();
```

Returns `TRUE` if the file is open, `returns FALSE` otherwise.

Accessing

GetName

```
void GetName (Str63 theName);
```

Gets the name of the file.

GetFSSpec

```
void GetFSSpec (FSSpec *aFileSpec);
```

Gets an `FSSpec` record for this file.

ExistsOnDisk

```
Boolean ExistsOnDisk ();
```

Returns `TRUE` if there's an existing file that matches the current file specification.

GetMacFileInfo

```
void GetMacFileInfo (FInfo *fileInfo);
```

Returns the Finder information for this file. The information includes the file's type, creator, and icon position.

Filing

CreateNew

```
void CreateNew (OSType creator, OSType fType);
```

Creates a new file with the specified creator and file type. This function uses the name and volume information you set up with one of the specification functions. You can use the application signature in `gSignature` for the creator.

ThrowOut

```
void ThrowOut ();
```

Closes the file and deletes it.

ChangeName

```
void ChangeName (Str63 newName);
```

Renames this file to `newName`.

CFileStream 54

Introduction

CFileStream provides streaming, buffered object I/O to and from CDataFile objects.

Heritage

Base Class	CBufferedStream
Derived Classes	None

Using CFileStream

See Chapter 8, “Using Object I/O,” to learn how to use streams and files.

A CFileStream object is associated with a file; it is not the file itself. Moreover, the association is not permanent. You can create a CFileStream object, attach it to an existing file object, read or write the file stream, and destroy the CFileStream, leaving the file intact. This is useful for reading and writing documents; the stream needs to exist only when you actually do I/O.

The easiest way to set up a file stream is through its two utility functions, `NewInputFileStream` and `NewOutputFileStream`. These functions create, initialize, configure and open the stream (and, if necessary, the file) in the appropriate mode. Often, all that is left for your program to do is to call `GetObject` or `PutObject` for the object data structure you wish to read or write, and to destroy the stream.

Data Members

CFileStream defines the following protected data members:

Data member	Type	Description
itsFile	CDataFile*	The data file used by the stream.
ownsFile	Boolean	TRUE if the file was created and disposed of by the stream.
wasOpen	Boolean	TRUE if the file was open when attached to the stream.

Member Functions

Creation and destruction

CFileStream

```
CFileStream (CDataFile *theFile = NULL,
   long bufferSize = kTCLDefaultStreamBufferSize,
   Boolean check = TRUE);
```

Constructor. Initializes a file stream to read or write the file argument. If `theFile` is `NULL`, a new CDataFile is created for use by the stream and must be further initialized by `SFSpecify` before the stream is opened. The `bufferSize` argument is the same as for CBufferedStream. The value of `kTCLDefaultStreamBufferSize` is 2048. The `check` argument is the same as for CStream.

~CFileStream

```
~CFileStream ();
```

Destructor. Closes the stream. If the file is owned by the stream, it deletes the file.

Open and close

Open

```
void Open (TCLStreamMode mode);
```

Opens the stream for `ReadStream`, `WriteStream` or `ReadWriteStream` access. If the file was created by the stream or was attached to the stream but is not already open, the file is opened at the same time as the stream. If the file was open when attached to the stream, it must already be open in the appropriate mode. CFileStream will not close a non-owned file, even temporarily.

Close
```
void Close ();
```
Closes the stream, returning it to `ClosedStream` mode. If the file was opened by the stream, the file is also closed. If the file was not opened by the stream, it is left open, but all outstanding writes are forced to disk.

File Operations

SFSpecify
```
void SFSpecify (SFReply *macSFReply);
```
Points the attached file object to a file on disk. This is the only file operation provided by CFileStream; for all others, including other forms of `Specify`, use `GetFile` to call the file object directly.

GetFile
```
CDataFile *GetFile ();
```
Returns a pointer to the file object attached to the stream.

Member Functions: Protected

Fill
```
void Fill ();
```
Fills the buffer from disk when the buffer is empty.

Drain
```
void Drain ();
```
Writes the buffer to disk when the buffer is full.

Bracket
```
void Bracket (long aPosition);
```
Repositions the buffer by draining it, if necessary, and resetting the buffer variables so that the buffer is an empty buffer starting at the desired position. The next `Get` or `Put` operation will read the buffer from disk, or write to an empty buffer, as appropriate.

GetPhysicalSize
```
long GetPhysicalSize ();
```
Returns the size of the file.

SetPhysicalSize

```
void SetPhysicalSize (long newSize);
```

Sets the size of the file to `newSize` bytes by calling the Macintosh Toolbox function `SetEOF`.

Utility Functions

NewOutputFileStream

```
CFileStream *NewOutputFilestream (
   CDataFile *theFile);
```

Creates a new file stream, turns duplicate checking on, opens the stream for `WriteStream` and returns a pointer to the stream. If `theFile` is `NULL`, a new CDataFile object is created for use by the stream.

NewInputFileStream

```
CFileStream *NewInputFilestream (
   CDataFile *theFile);
```

Creates a new file stream, turns duplicate checking on, opens the stream for `ReadStream` and returns a pointer to the stream. `theFile` may not be `NULL`.

CFloatDirector 55

Introduction

CFloatDirector is an abstract class that supervises a Macintosh floating window.

Heritage

Base Class	CDirector
Derived Class	CTearOffMenu

Using CFloatDirector

CFloatDirector controls a floating window. Your CFloatDirector derived class must set `itsWindow` to a window you create.

CFloatDirector provides a number of services common to all standard Macintosh floating windows. The THINK Class Library assumes that all floating windows are created when the application is initialized and that they exist throughout the life of the program. Usually, a global variable is set to point to the floating window's director. Floating windows are hidden offscreen before they are first opened and while they are closed. Floating windows are temporarily hidden when the application is suspended, and displayed again when the application is resumed. A floating window should be hidden when it does not pertain to the topmost non-floating, non-modal window. Floating windows are allowed to go behind modal windows and are not hidden while modal windows are frontmost.

CFloatDirector makes this easy to implement. For each floating window, call CFloatDirector's `ShowWindow` function in the `Activate` function of each non-floating, non-modal window to which the floating window applies. Call `HideWindow` in the `Deactivate` function of each non-floating, non-modal window to which the floating window does not apply. `ShowWindow` will not

show the window unless it has been opened and then was hidden by `HideWindow`. Suspend and resume behavior is handled independently; your program logic does not need to be concerned about it.

Data Members

CFloatDirector defines these protected data members:

Data member	Type	Description
`itsPane`	`CPane*`	The pane being displayed in the window. Used by CTearOffMenu.
`corner`	`Point`	Upper left of torn-off window
`margins`	`Rect`	Space between window bounds and the pane in the window
`opened`	`Boolean`	`TRUE` if window has been opened
`hidden`	`Boolean`	`TRUE` if window has been hidden by `HideWindow`

Member Functions

Creation and destruction

CFloatDirector

```
CFloatDirector ();
```

Constructor. A float director's supervisor is always the application. CFloatDirector sets `itsWindow` and `itsPane` to `NULL`. The window's margins are initialized to one pixel on each side. To conform to *Macintosh Human Interface Guidelines*, call `SetFixedMargins` to set the margins for a top or left drag bar.

HideInitially

```
void HideInitially ();
```

Completes initialization by hiding the window offscreen so that it will only appear onscreen when opened or torn off.

Accessing

GetMacWindow

```
WindowPtr GetMacWindow ();
```

Returns the window record for the floating window.

IsOpen
```
Boolean IsOpen ();
```
Returns `TRUE` if the floating window is currently open—that is, it was opened by `OpenWind` or `MoveToCorner`.

Open and close

OpenWind
```
void OpenWind (Point where);
```
Opens the floating window by moving its upper-left corner to the global coordinates specified by `where`.

CloseWind
```
void CloseWind (CWindow *theWindow);
```
Closes the floating window by moving `theWindow` off the screen. `theWindow` must be the same as `itsWindow`. The argument is provided for backward compatibility.

MoveToCorner
```
void MoveToCorner ();
```
Moves the window to the corner stored by the `TornOff` member function.

Hide and Show

ShowWindow
```
void ShowWindow ();
```
Shows the floating window. Does nothing if the window is not open or was not previously hidden by `HideWindow`.

HideWindow
```
void HideWindow ();
```
Hides the floating window if it is currently open and not hidden.

Suspend and resume

Suspend
```
void Suspend ();
```
Hides the floating window when the application is suspended. You should not have to call this function.

Resume

```
void Resume ();
```

Shows the floating window when the application is reactivated. You should not have to call this function.

Commands

DoCommand

```
void DoCommand (long theCommand);
```

Overrides `DoCommand` to forward all commands received by the director to the gopher, provided the gopher is not in the director's window. This makes it simple to implement floating palettes; every selection can become a command received by the front window.

Margins

SetFixedMargins

```
void SetFixedMargins (tDragBar dragBar);
```

Sets the margin for a top or left drag bar. The value of `dragBar` must be either `kTopDragBar` or `kLeftDragBar`. The drag bar is ten pixels high or wide.

SetMargins

```
void SetMargins(Rect *aMargins);
```

Sets the margin between the window outline and the pane in the window. The margin is not a rectangle; it is the amount by which the pane's frame should be enlarged to create the gray outline of the floating window.

GetMargins

```
void GetMargins(Rect *theMargins);
```

Returns the margins in `theMargins`.

CFWDesktop 56

Introduction

CFWDesktop is obsolete, and provided only for backward compatibility.

Heritage

Base Class	CDesktop
Derived Classes	None

Using CFWDesktop

All the functionality of CFWDesktop is now included in CDesktop. Unless you have existing code which depends on CFWDesktop, you will never need to use this class.

For information on CDesktop, see Chapter 40.

Data Members

CFWDesktop defines no data members.

Member Functions

Creation and destruction

IFWDesktop

```
void IFWDesktop (CBureaucrat *aSupervisor);
```

Initializes the desktop. Provided for backward compatibility. Does nothing.

CGridMDEF 57

Introduction

CGridMDEF overrides CSelectorMDEF to do a more standard job of highlighting the current menu selection. CGridMDEF highlights by inverting, rather than by constantly flashing.

Heritage

Base Class	CSelectorMDEF
Derived Classes	None

Using CGridMDEF

Use CGridMDEF exactly as you would use CSelectorMDEF. See Chapter 69, “CMenuDefProc,” for detailed coding hints.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CGridMDEF

```
CGridMDEF (short MDEFid, CPane *aPane,
   CTearOffMenu *aTearOffMenu);
```

Constructor. Initializes the object with the resource ID of the dummy `'MDEF'` and the associated `'MENU'` resource. `aPane` must be a CSelector-derived class object. `aTearOffMenu` is the director that manages a torn-off menu and must be a CTearOffMenu object.

Drawing

ChooseItem

```
void ChooseItem (MenuHandle macMenu,
   Rect *menuRect, Point hitPt,
   short *whichItem);
```

Handles menu selection.

DrawMenu

```
void DrawMenu (MenuHandle macMenu,
   Rect *menuRect);
```

Draws the pane specified as `aPane` in `IGridMDEF` as a menu.

CGridSelector 58

Introduction

CGridSelector is an abstract class for drawing a pane with several items arranged in a table.

Heritage

Base Class	CSelector
Derived Classes	CCharGrid
	CPatternGrid
	CPICTSelector
	CPICTGrid

Using CGridSelector

CGridSelector is a derived class of CSelector that lets you display items in a table. To use CGridSelector, all you have to do in any class you derived from it is override the `DrawItem` member function. CGridSelector takes care of everything else.

The THINK Class Library includes two derived classes of CGridSelector. CPatternGrid displays a pattern palette. CCharGrid displays a set of characters. You can use CCharGrid with a special font to display tool palettes.

Items in a grid are numbered in row-column order starting with 1.

Data Members

CGridSelector defines the following data members:

Data member	Type	Description
rows	short	The number of rows in the grid
columns	short	The number of columns in the grid
boxWidth	short	The width in pixels of each box
boxHeight	short	The height in pixels of each box
gridOn	Boolean	TRUE if the grid lines should be drawn

Member Functions

Creation and Destruction

CGridSelector

```
CGridSelector ();
```

Default constructor. Implicitly called when object is created by new_by_name. Can also be used in combination with IGridSelector for backward compatibility.

CGridSelector

```
CGridSelector (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aSelection, short aCommandBase
   short aRows, short aColumns,
   short aBoxWidth, short aBoxHeight);
```

See the description of CSelector's DoClick function in Chapter 97 to learn how selectors generate command numbers.

Constructor. aSelection is the initial item to select. aCommandBase is the base value for converting the selected item into a command number. aRows and aColumns determine how many boxes are in the grid. aBoxWidth and aBoxHeight determine the size of each box in pixels. CGridSelector sets the vertical and horizontal scales to the size of the boxes.

Note

The other arguments are described in Chapter 71, "CPane."

IGridSelector

```
void IGridSelector (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aSelection, short aCommandBase
   short aRows, short aColumns,
   short aBoxWidth, short aBoxHeight);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the grid selector. If `gridOn` is true, this function first draws the grid. Then it uses `DrawItem` to draw each box. Finally, it highlights the current selection.

Since most of the work is done in the `DrawItem` function, you probably won't need to override this function.

DrawGrid

```
void DrawGrid ();
```

Draws the grid lines between items. The `Draw` function calls `DrawGrid` if `gridOn` is `TRUE`.

DrawItem

```
void DrawItem (short theItem, Rect *theBox);
```

Draws the item. `theItem` is the number of the item to draw, and `theBox` is the rectangle to draw it in. Your derived class must override this function.

Accessing

HiliteItem

```
void HiliteItem (short theItem,
   HiliteState state);
```

Highlights the specified item. `HiliteItem` can take one of three values for the `state` parameter:

Hilite state value	Behavior
`hiliteOFF`	Invert the item's box
`hiliteON`	Invert the item's box
`hiliteDYNAMIC`	Flash the edges of the item's box

Table 58-1 How CGridSelector highlights items

If you want to highlight an item differently, you can override this function.

FindItem

```
short FindItem (Point hitPt);
```

Determines which item corresponds to a mouse down at a specified point. If the `hitPt` isn't in the grid, `FindItem` returns zero. Grid selector items are numbered in row-column order starting with 1, as in Figure 58-1.

1	2	3
4	5	6
7	8	9

Figure 58-1 Item numbering in grid selectors

FindItemBox

```
void FindItemBox (short theItem, Rect *theBox);
```

Returns the box that corresponds to `theItem`. `HiliteItem` uses this function to figure out what box to highlight.

SetGridOn

```
void SetGridOn (Boolean aGridOn);
```

Sets the data member `gridOn` to `aGridOn`. If `gridOn` is `TRUE`, the `Draw` function calls `DrawGrid` to draw grid lines between items.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CGroupButton 59

Introduction

CGroupButton is an abstract class that models the behavior of group buttons. Each group button has a group ID. All buttons with the same non-zero group ID are members of the same button group. Button groups are confined to a single window.

Heritage

Base Class	None
Derived Classes	CSwissArmyButton
	CIconButton
	CCheckBox
	CRadioControl

Using CGroupButton

Buttons in a group exhibit two kinds of behavior. They can act like radio buttons or like check boxes. When a radio button is turned on, all other radio buttons and check boxes in the same group are turned off. When a check box is turned on, other checkboxes are unaffected, but all radio buttons in the same group are turned off. A group button acts like a radio button if its `IsRadioButton` member function returns `TRUE`; otherwise, it acts like a check box. A CSwissArmyButton or CIconButton that belongs to a group can be made to exhibit either kind of behavior by calling its `SetButtonKind` member function.

Data Members

CGroupButton defines the following protected data members:

Data member	Type	Description
`groupID`	`unsigned short`	The button group ID; has the value `0` if the button is not a member of a group.
`itsGroupEnclosure`	`CGroupButtonEnclosure*`	The enclosure of the button group. (The Type must be entered with no break.)

Member Functions

Creation and destruction

CGroupButton

```
CGroupButton (CView *anEnclosure = NULL,
   short aGroupID = 0);
```

Constructor. If `anEnclosure` is not `NULL`, the constructor sets the data member `itsGroupEnclosure` to the first CGroupButtonEnclosure in the visual hierarchy found by searching from `anEnclosure` upward toward the desktop. Sets the data member `groupID` to `aGroupID`.

~CGroupButton

```
~CGroupButton ();
```

Destructor. Removes this button from `itsGroupEnclosure` by calling the enclosure's `RemoveGroupButton` member function.

IGroupButton

```
void IGroupButton (CView *anEnclosure);
```

Initialization function provided for backward compatibility. If no constructor arguments are specified, the object must be further initialized by calling `IGroupButton`. If any constructor arguments are specified, `IGroupButton` must not be called.

If `anEnclosure` is not `NULL`, `IGroupButton` sets the data member `itsGroupEnclosure` to the first CGroupButtonEnclosure

in the visual hierarchy found by searching from `anEnclosure` upward toward the desktop.

Dispose

```
void Dispose ();
```

For backward compatibility. Calls `delete this`. May be used only if `TCL_USE_DISPOSE` is defined in your project.

Group ID

SetGroupID

```
void SetGroupID (short id);
```

Sets the data member `groupID` to `id`.

GetGroupID

```
short GetGroupID ();
```

Returns the data member `groupID` .

Changing state

TurningOn

```
void TurningOn (CGroupButton *button);
```

Called when `button`, a button whose group enclosure is also `itsGroupEnclosure`, is turning on. If `button` and `this` belong to the same group, if `button` is not `this` button, and if either `button` or `this` button is a radio button, then this function calls `TurnOff`. If those three conditions are not met, the function does nothing.

This function is usually called by the member function `TurningOn` of `itsGroupEnclosure` when that function is called with the argument `button`. `itsGroupEnclosure` has a data member `itsGroupButtons` that is a CGroupButtonList. When called with the argument `button`, the `TurningOn` function of `itsGroupEnclosure` loops through the array and calls each CGroupButton's `TurningOn` function with the same `button` argument.

Object I/O

PutTo
```
void PutTo (CStream& aStream);
```
Writes to the stream.

GetFrom
```
void GetFrom (CStream& aStream);
```
Reads from the stream.

Member Functions: Protected

TellTurningOn
```
short TellTurningOn ();
```
Protected function, called when this button changes state from off to on. If this button is not a member of a group, `TellTurningOn` does nothing. Otherwise, this function notifies all other buttons in the group of the state change by calling `itsGroupEnclosure->TurningOn(this)`.

TurnOff
```
void TurnOff ();
```
Pure virtual function, which derived classes must override.

Querying

IsRadioButton
```
Boolean IsRadioButton ();
```
The default function returns FALSE. Derived classes should override this function as appropriate.

CGroupButtonEnclosure 60

Introduction

A CGroupButtonEnclosure object maintains a list of group buttons (objects of class CGroupButton). When a group button is turned on, this list is searched to find every other button in the same group and inform it of the state change.

Heritage

Base Class	None
Derived Classes	CWindow

Using CGroupButtonEnclosure

A CGroupButtonEnclosure object maintains a list of CGroupButtons in its data member `itsGroupButtons`. Every CGroupButton on that list has its data member `itsGroupEnclosure` pointing back to the CGroupButtonEnclosure that owns the list.
When a group button is turned on, its `TellTurningOn` member function is called. `TellTurningOn` calls `itsGroupEnclosure->TurningOn(this)`. The CGroupEnclosure object `itsGroupEnclosure` then searches its list to find every button in the same group and inform it of the state change.

Data Members

CGroupButtonEnclosure defines the following protected data member:

Data member	Type	Description
`itsGroupButtons`	`CGroupButtonList*`	Array of group buttons belonging to this group button enclosure

Member Functions

Creation and destruction

CGroupButtonEnclosure

```
CGroupButtonEnclosure ();
```

Default constructor. Initializes `itsGroupButtons` to `NULL`.

~CGroupButtonEnclosure

```
~CGroupButtonEnclosure ();
```

Destructor. Deletes `itsGroupButtons`.

Adding and removing

AddGroupButton

```
void AddGroupButton(CGroupButton *button);
```

Adds `button` to the list `itsGroupButtons` by calling the list's `Add` member function with the argument `button`. If `itsGroupButtons` is `NULL`, this function first creates a new CPtrArray of CGroupButtons.

RemoveGroupButton

```
void RemoveGroupButton(CGroupButton *button);
```

Removes `button` from the list `itsGroupButtons` by calling the list's `Remove` member function with the argument `button`.

Changing state

TurningOn

```
void TurningOn (CGroupButton *button);
```

Called when `button`, a member of this group enclosure, is turning on. This function communicates the state change to every button in the list `itsGroupButtons` by calling the button's `TurningOn` function with the argument `button`.

CHandleStream 61

Introduction

CHandleStream provides streaming object I/O to and from relocatable blocks on the heap.

Heritage

Base Class	CBufferedStream
Derived Classes	None

Using CHandleStream

See Chapter 8, “Using Object I/O,” to learn how to use streams and files.

The primary use of CHandleStream is to read and write resources. A resource is read from disk, attached to a stream, and converted to objects. In the other direction, a stream is created, objects are written to the attached handle, then the handle is written to disk as a resource.

A secondary use of CHandleStream is to implement cutting and pasting objects to and from the scrap. To cut or copy, use an output stream to convert the objects to handle form, then pass the handle to the scrap. To paste, get the handle from the scrap, use an input stream to convert the handle to objects, then link the objects into your other data structures. See Chapter 8, “Using Object I/O,” for sample code.

The easiest way to set up a handle stream is through the two utility functions of this class, `NewInputHandleStream` and `NewOutputHandleStream`. As their names suggest, these functions create, initialize, and open the stream in the appropriate mode. Often, all that remains for your program to do is call

GetObject or PutObject for the data structure you want to read or write, and delete the stream.

Data Members

CHandleStream defines this protected data member:

Data members	Type	Description
blockSize	long	The number of bytes added to the handle each time it must grow

There is no handle data member. The handle owned by the CBufferedStream base class is used instead. CHandleStream uses that part of the CBufferedStream's implementation that transfers data to and from a handle but does not perform buffering. This is one of the cases in which a derived class is not logically a subtype of its base class. The derivation relationship expresses not "is a," but rather "is implemented in terms of."

Member Functions

Creation and destruction

CHandleStream

```
CHandleStream (Handle theHandle,
   long bufferSize = kTCLDefaultStreamBufferSize,
   Boolean check = TRUE);
```

Constructor. Initializes a handle stream to read or write the handle argument. If theHandle is NULL, a new handle of bytes is created for use by the stream. If theHandle is not NULL, it is used by the stream. The bufferSize and check arguments are the same as for CBufferedStream, except that bufferSize is also the initial value of the blockSize data member.

CHandleStream

```
CHandleStream ();
```

Default constructor. Implicitly called when object is created by new_by_name.

~CHandleStream

```
~CHandleStream ();
```

Destructor. Does not delete the associated handle.

DisposeAll

```
void DisposeAll ();
```

Deletes the stream and the associated handle.

SetBlockSize

```
void SetBlockSize (long aBlockSize);
```

Sets the number of bytes to be allocated each time more space is needed for the associated handle.

Open and Close

Open

```
void Open (TCLStreamMode mode);
```

Opens the stream for `ReadStream`, `WriteStream`, or `ReadWriteStream` access.

Close

```
void Close ();
```

Closes the stream, returning it to `ClosedStream` mode. If the stream was opened for output, the associated handle is truncated to the number of bytes actually written.

GetStreamHandle

```
Handle GetStreamHandle ();
```

Returns the handle used by the stream. If the stream is open for writing, it is closed before the handle is returned, ensuring that the handle is truncated to the exact size of the stream.
`GetSreamHandle` is most often used to get the handle created by an output stream so it can be written as a resource, for example:

```
CHandleStream *stream;
Handle h;

stream = GetNewOutputHandleStream(NULL);
stream->PutObject(myData);
h = stream->GetStreamHandle();
delete stream;
// write the handle h as a resource
```

NewOutputHandleStream

```
CHandleStream *NewOutputHandleStream (
   Handle theHandle);
```

Creates a new handle stream, turns on duplicate checking, opens the stream for `WriteStream`, and returns a pointer to the stream. If `theHandle` is `NULL`, a new handle is allocated as well.

NewInputHandleStream

```
CHandleStream *NewInputHandleStream (
   Handle theHandle);
```

Creates a new handle stream, turns on duplicate checking, opens the stream for `ReadStream`, and returns a pointer to the stream. `theHandle` may not be `NULL`.

Member Functions: Protected

The following protected functions are used to communicate with the CBufferedStream base class.

Drain

```
void Drain ();
```

Increases the size of the handle by `blocksize` when it is full.

GetPhysicalSize

```
long GetPhysicalSize ();
```

Returns the size of the handle.

SetPhysicalSize

```
void SetPhysicalSize (long newSize);
```

Truncates the handle to length `newSize`.

Utility Functions

Bracket

```
void Bracket (long aPosition);
```

Do not call this function. Bracket is not implemented for CHandleStream.

CIconButton 62

Introduction

CIconButton extends CIconPane, turning it into a full-fledged, multi-state button.

You can specify the highlighting style as either multi-state or inverted and the drawing style as either multi-state or bordered. Multi-state buttons can be constructed by specifying up to four `'ICON'`/ `'cicn'` resource IDs, one for each of the button states: off, off-highlighted, on, and on-highlighted. For example:

Figure 62-1 off off-highlighted on on-highlighted

Icon buttons can participate in button groups. All buttons with the same non-zero group ID participate in the same button group. An icon button can behave like a push button, a check box or a radio button. Radio buttons turn off all other buttons in the group when they are selected; check boxes turn off only radio buttons in the group, but not other check boxes.

An icon button that is not multi-state represents its highlight states by inverting itself. It represents its on state by drawing a border around the icon.

Heritage

Base Classes	CIconPane CGroupButton
Derived Classes	None

Using CIconButton

Set up a CIconButton object in the constructor. You can also specify that the button receive the command `clickCmd` when it is clicked.

To change the size or shape of the border shown when `onStyle` equals `kSABorderOn`, call `GetBorder` to obtain a pointer to the border and use, for example, `SetPenSize`.

Data Members

CIconButton defines the following data members:

Data members	Type	Description
`buttonKind`	`short`	One of `kSAPushButton`, `kSACheckBox` or `kSARadioButton`.
`colorHilite`	`Boolean`	`TRUE` if highlighted by inverting; `FALSE` if multi-state highlight.
`outlineOn`	`Boolean`	`TRUE` if on state indicated by border; `FALSE` if multi-state.
`value`	`short`	Value: 0 or 1 for standard buttons.
`itsIconID[4]`	`short`	Array of `'ICON'`/`'cicn'` resource IDs for multi-state buttons.
`itsIcon[4][kIconBytes]`	`char`	Array of `'ICON'` bitmaps.
`itsCicnH[4]`	`CIconHandle`	Array of cicn handles.
`borderPen`	`Point`	Border pen width and height saved while border is not showing.

Member Functions

Creation and destruction

CIconButton

```
CIconButton ();
```

Default constructor. Implicitly called when object is created by `new_by_name`.

CIconButton

```
CIconButton (CView *anEnclosure,
   CBureaucrat *aSupervisor, short aHEncl,
   short aVEncl, SizingOption aHSizing,
   SizingOption aVSizing, short buttonKind,
   short hilightStyle, short onStyle,
   short offID,short offHiliteID,
   short onID, short onHiliteID);
```

Constructor. The `anEnclosure`, `aSupervisor`, `aHEncl`, `aVEncl`, `aHSizing`, and `aVSizing` arguments are the same as for any pane. Specify `buttonKind` as one of the following: `kSAPushButton`, `kSACheckBox`, or `kSARadioButton`; `highlightStyle` as one of the following: `kSAStateHilight` or `kSADimHilight`; and `onStyle` as either `kSAStateOn` or `kSABorderOn`. The four resource IDs: `offID`, `offHiliteID`, `onID`, and `onHiliteID`, are used to obtain one to four `'ICON'` and, if available, `'cicn'` resources, which correspond to the off, off-selected, on and on-selected states of the button. The on-selected and off-selected states are displayed when the button is on or off, respectively, and the user has the mouse down in the button.

~CIconButton

```
~CIconButton ();
```

Deletes the button's color icon storage. Does not interfere with any other CIconPane or CIconButton objects that might be using the same icons at the same time.

Tracking

DrawIcon

```
void DrawIcon (Boolean fHilite);
```

Override to set up the correct icon to draw and to adjust the `fHilite` setting based on the value of `colorHilite`. Do this before calling `CIconPane::DrawIcon`.

Track
```
Boolean Track ();
```
Overrides to call `SetValue` for radio buttons or check boxes that have changed state.

Access

SetValue
```
void SetValue (short aValue);
```
Sets the button value. The standard buttons accept only `0` or `1`. Setting the value of an icon button that acts as a radio button or check box affects other buttons in the same button group, as described in the Introduction at the beginning of this chapter.

GetValue
```
short GetValue ();
```
Returns the button value.

SetButtonKind
```
void SetButtonKind (short aKind);
```
Sets the value of `buttonKind`. `aKind` must be one of these: `kSAPushButton`, `kSACheckBox`, or `kSARadioButton`.

GetButtonKind
```
short GetButtonKind ();
```
Returns the button kind.

Button Groups

TurnOff
```
void TurnOff ();
```
For radio button and check box style buttons, turns the button off by calling `SetValue(0)`.

IsRadioButton
```
Boolean IsRadioButton ();
```
Returns `TRUE` if the `buttonKind` is `kSARadioButtonKind`.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the button to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the button from the stream.

Member Functions: Protected

Internal

CalcDrawState

```
short CalcDrawState (Boolean fHilite);
```

Based on the following three data members: `value`, `outlineOn`, and `colorHilite`, and on the argument `fHilite`, this function computes the index into the icon array required to display the current button state. The `fHilite` argument is `TRUE` if the mouse is down in the button.

DoDrawIcon

```
void DoDrawIcon (Boolean fHilite);
```

Verifies that the icon needs to be erased before drawing and erases it if necessary before calling `DrawIcon`.

FixupBorder

```
void FixupBorder ();
```

Depending on the button `value` and the value of the data member `outlineOn`, it adds or removes a border around the icon. If a border is present, it is hidden while `value` is 0 by setting its pen width and height to 0.

SetStateIcons

```
void SetStateIcons (short offHiliteID,
   short onID, short onHiliteID);
```

Loads the `'ICON'` and `'cicn'` resources required to display the states of the button.

Member Functions: Private

CIconButtonX

```
void CIconButtonX ();
```

Performs common constructor initialization.

IIconButtonX

```
void IIconButtonX (short offHiliteID,
   short onID, short onHiliteID);
```

Performs common post-initialization by loading the first icon array element and calling `SetStateIcons` to load the rest.

CIconPane 63

Introduction

CIconPane implements panes that draw `'ICON'` and `'cicn'` (color icon) resources and that behave like buttons.

Heritage

Base Class	CPane
Derived Classes	CIconButton

Using CIconPane

CIconPane implements panes that draw `'ICON'` and `'cicn'` resources and that behave like buttons. This class takes advantage of the System 7 Icon Utilities if they are present. Additional button-like functionality is implemented in the derived class CIconButton.

CIconPane uses the following constants defined in `CIconPane.h`:

Constant	Value	Description
`kIconPixels`	`32`	The number of pixels on each side of an `'ICON'`
`kIconBytes`	`128`	The number of bytes in an `'ICON'`
`kIconRowBytes`	`4`	32 black-and-white pixels occupy 4 bytes

Data Members

All data members of CIconPane are protected.

CIconPane defines these static data members:

Data member	Type	Description
`cMaskBitMap`	`BitMap`	Holds an icon's bitmap while drawing icons
`cIconMaskBits [kIconBytes]`	`char`	Holds the bits of the icon bitmap while drawing icons
`cHaveIconDispatch`	`Boolean`	`TRUE` if System 7 Icon Utilities are present

CIconPane defines these data members for each object:

Data member	Type	Description
`iconID`	`short`	The resource ID of the icon
`icon [kIconBytes]`	`char`	The icon's data
`cicnH`	`CIconHandle`	Handle to the `'cicn'` if the icon is a color icon
`clickCmd`	`long`	Command executed by `DoClick`

Member Functions

Creation and destruction

CIconPane

```
CIconPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short iconID, Boolean fPreferColor = TRUE);
```

Constructor. The first six arguments are identical to those for the CPane constructor. `iconID` is the resource ID of the icon, which the constructor saves in the data member of the same name. This constructor loads the icon from an `'ICON'` resource with the specified ID. If `fPreferColor` is `TRUE`, if Color QuickDraw is available, and if a `'cicn'` resource with the same ID exists, then the color icon is used instead. In that case, the data member `cicnH` is set to a CIconHandle that refers to the loaded color icon; otherwise, `cicnH` is set to `NULL`. `clickCmd` is initialized to `NULL`, and `'icon'` is filled with the icon's data.

CIconPane

```
CIconPane ();
```

Default constructor. Implicitly called when this object is created by `new_by_name`. Can also be used in combination with `IIconPane` for backward compatibility.

~CIconPane

```
~CIconPane ();
```

Destructor. Frees the color icon `cicnH` if it exists.

IIconPane

```
void IIconPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short iconID, Boolean fPreferColor);
```

Initialization function provided for backward compatibility. If no constructor arguments are specified, new object must be further initialized by calling `IIconPane`. If any constructor arguments are specified, `IIconPane` must not be called. See constructor CIconPane for description of the arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   Ptr viewData);
```

Initializes the CIconPane from a view resource.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the icon in response to an update by calling `DrawIcon(FALSE)`.

Command

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Responds to a click. If the mouse is clicked and released over the icon, this function calls its supervisor's `DoCommand` function with the argument `clickCmd`.

SetClickCmd

```
void SetClickCmd (long aCmd);
```

Sets the data member `clickCmd` equal to `aCmd`.

GetClickCmd

```
long GetClickCmd ();
```

Returns `clickCmd`.

Object I/O

PutTo

```
void PutTo (Stream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (Stream& aStream);
```

Reads from the stream.

Member Functions: Protected

Track

```
Boolean Track ();
```

Used by `DoClick`. Tracks the mouse; returns `TRUE` if the mouse is over the icon when released, `FALSE` otherwise.

DrawIcon

```
void DrawIcon (Boolean fHighlite);
```

Used by `Draw`. Draws the icon highlighted or non-highlighted.

IIconPaneX

```
void IIconPaneX (Boolean fPreferColor);
```

Performs common initialization.

63 CIconPane

CIntegerText 64

Introduction

CIntegerText is a derived class of CDialogText that handles numeric fields in dialogs.

Heritage

Base Class	CDialogText
Derived Classes	None

Using CIntegerText

CIntegerText handles numeric fields in dialogs. It lets you enter only integer values in the field. You can also specify these constraints on the value:

- Range. You can specify a minimum and maximum value with `SpecifyRange`.
- Default value. You can specify a value to return if the user leaves the field empty with `SpecifyDefaultValue`.

For more information on how a dialog enforces these constraints or on how to create derived classes with more constraints, see Chapter 43, "CDialogText."

Data Members

CIntegerText defines these protected data members:

Data members	Type	Description
`minValue`	`long`	Minimum valid value.
`maxValue`	`long`	Maximum valid value.

Data members	Type	Description
defaultValue	long	Default value if text empty. If isRequired is FALSE, empty text is considered valid and GetIntValue returns this value.
showRangeOnErr	Boolean	If TRUE, validation errors display the allowed range of values.

Member Functions

Creation

CIntegerText

```
CIntegerText ();
```

Default constructor. Implicitly called when object is created by new_by_name. Can also be used in combination with IIntegerText for backward compatibility.

CIntegerText

```
CIntegerText (CView *anEnclosure,
   CView *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizELASTIC,
   SizingOption aVSizing = sizELASTIC,
   short aLineWidth = -1,
   Boolean aScrollHoriz = 0,
   Boolean aIsRequired = FALSE,
   long aMaxValidLength = MAXLONG,
   long aMinValue = MINLONG,
   long aMaxValue = MAXLONG,
   long aDefaultValue = 0,
   Boolean aShowRangeOnErr = FALSE);
```

Constructor.

IIntegerText

```
void IIntegerText (CView *anEnclosure,
   CView *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

IViewTemp

```
void IViewTemp(CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally for initializing from a resource template. Each derived class of CView overrides this function to use its own resource template.

Accessing

SpecifyRange

```
void SpecifyRange(long aMinimum, long aMaximum);
```

Sets the range of valid integer values to be `aMinimum` to `aMaximum`. This function also sets `showRangeOnErr` to indicate whether the range will be displayed in error messages. If the range is `MINLONG` to `MAXLONG`, it does not show the range (that is, it sets `showRangeOnErr` to `FALSE`). Otherwise, it does show the range (it sets `showRangeOnErr` to `TRUE`).

SpecifyDefaultValue

```
void SpecifyDefaultValue(long aDefaultValue);
```

Specifies that this field isn't required and the value to return if it's empty. This function sets `defaultValue` to `aDefaultValue` and `isRequired` to `FALSE`.

SetIntValue

```
void SetIntValue(long aValue);
```

Sets this field's text to a string containing the number `aValue`.

GetIntValue

```
long GetIntValue();
```

Returns this field's value. If the field is empty, this returns the default value.

Validation

Validate

```
Boolean Validate();
```

Returns `TRUE` if the field is valid according to your constraints.

Returns FALSE if:

- The field is required, but is empty
- The field's value is outside the allowed range
- The field's text is longer than the maximum length

If the field is invalid, this function uses `ReportInvalidText` to display an error message in an alert.

Override this function if your derived class adds more validations.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom(CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

ConvertToInteger

```
void ConvertToInteger(long *intValue,
   Boolean *valid);
```

Converts the field's text to a `long`, and sets `*intValue` to it. If the field contains a valid long integer, it sets `*valid` to `TRUE`; otherwise, it sets `*valid` to `FALSE`.

CLine 65

Introduction

CLine draws a line from one point to another.

CLine is also a CSwissArmyButton, which means that you can make a line click-sensitive and have it send a command. More often, though, lines are just decoration.

Heritage

Base Class	CSwissArmyButton
Derived Classes	None

Using CLine

Use CLine when you need to draw a few lines in your user interface or when you need a click-sensitive line. For drawing many lines, you may want something more lightweight.

Unlike most CPane-derived classes, the size and location of a CLine is specified by a pair of points in enclosure coordinates. CLine figures out the frame and other location values.

Lines are actually more difficult to program than the usual rectangle-based objects. CLine contains two functions you may find handy. `CalcBoundingRect` computes the smallest rectangle that completely encloses a line, taking into account the pen size. The more interesting `CalcBoundingRgn` computes the smallest region that completely encloses a line. You can use this to detect a hit, draw a border around a line, compute intersections, and so forth. You can set the argument `fuzz` of arbitrary size around a line so that the user can select it with a mouse. This function is fast enough for user interface work. These two functions are static, so you can call them from any object.

Data Members

Data member	Type	Description
fromPt	Point	Starting point of the line
toPt	Point	Ending point of the line
straight	Boolean	TRUE if line should be constrained to a slope that is a multiple of 45°

Member Functions

Creation and destruction

CLine

```
CLine ();
```

Default constructor. Implicitly called when object is created by new_by_name.

CLine

```
CLine (CView *anEnclosure, CBureaucrat
   *aSupervisor, Point fromPt, Point toPt,
   short penWidth, short penHeight,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY,
   Boolean aNeedsUpdate = FALSE,
   Boolean aStraight = TRUE);
```

Constructor. Initializes the line object by specifying its starting fromPt and ending toPt and pen height and width. The aHSizing and aVSizing options have their usual CPane meaning. A line is a CSwissArmyButton; push button style, only. The aNeedsUpdate argument specifies whether the line must be updated to draw its highlight state, that is, whether it is overlapped by any other graphics.

CLine creates a CColorTextEnvirons object to hold the pen size. You can access the environment directly to set pen color, pattern, and others. If you change the pen size, be sure to call SetEndpoints to recalculate the frame and enclosure coordinates.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the line.

Tracking

InButton

```
Boolean InButton (Point aPoint);
```

Returns TRUE if the point is on the line. This function uses a one-pixel fuzz; override if you want more.

Access

SetEndpoints

```
void SetEndpoints (Point fromPt, Point toPt);
```

Resets the line's endpoints and recalculates the frame and enclosure coordinates.

GetEndpoints

```
void GetEndpoints (Point *fromPt, Point *toPt);
```

Returns the endpoints of the line.

GetBoundingRgn

```
void GetBoundingRgn (short fuzz, RgnHandle rgn);
```

Returns the smallest region that completely encloses the line, after thickening and extending the line by the amount `fuzz`.

SetStraight

```
short SetStraight (Boolean straightLine);
```

Sets the `straight` data member. `straight` doesn't actually do anything; it just remembers the value for other objects which might want to change the slope of the line.

GetStraight

```
Boolean GetStraight ();
```

Returns the value of `straight`.

Utility

CalcBoundingRect

```
static void CalcBoundingRect (Point fromPt,
   Point toPt, short penWidth, short penHeight,
   Rect *theRect);
```

Calculates the smallest rectangle that completely encloses the line specified by the arguments.

CalcBoundingRgn

```
static void CalcBoundingRgn (Point fromPt,
   Point toPt, short penWidth,
   short penHeight, short fuzz,
   RgnHandle theRgn);
```

Calculates the smallest region that completely encloses the line specified by the arguments, after thickening and extending the line by the amount of `fuzz`. Fuzz is useful for hit-testing. A mathematical hit-test would be faster, but this function is fast enough for most user interface work and more generally useful.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the line to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the line from the stream.

CList 66

Introduction

CList implements an ordered list of objects.

Heritage

Base Class	CPtrArray
Derived Classes	None

Using CList

Use an object of class CList when you need to maintain an ordered list of objects that can do Object I/O. If the list does not need to be read from or written to a stream, use CPtrArray instead.

Several of the objects in the THINK Class Library use CList to maintain lists. You can use the iteration functions inherited from CPtrArray to apply functions to each item in a list. Every object in a list has an index. Index values begin at 1.

CList is a template class. To use a CList object, you must specify the class of objects the list is to contain. For example:

```
CList<MyClass> *myClassList;
```

Create and use a CList class in the usual way, for example:

```
myClassList = new CList<MyClass>;
myClassList->Append(myClassObject);
```

All THINK Class Library headers are compiled with `#pragma template_access extern`. For each CList-derived class declared as above, you will need to compile a file or files containing at least the following lines:

```
#include <CList.h>
#include <MyClass.h>
#pragma template_access public
#pragma template CPtrArray<MyClass>
#pragma template CList<MyClass>
TCL_DEFINE_CLASS_M1(CPtrArray<MyClass>,
   CPtrArrayBase);
TCL_DEFINE_TEMPLATE_CLASS_D1(CList, MyClass,
   CPtrArray<MyClass>)

#include <CPtrArray.tem>
#include <CList.tem>
```

These lines create global instances of the template classes `CPtrArray<MyClass>` and `CList<MyClass>` and construct the tables required for Object I/O. You must define each template class exactly once in this manner. For simple cases like the above, just copy the `CList_MyClass` file in the `THINK Class Library:Templates` folder and make the appropriate substitutions. See the files `CPtrArray_CView.cpp` and `CList_CView.cpp` for examples of more complex cases, for example, where member functions are overridden.

Data Members

This class has no data members.

Member Functions

For functions that insert new objects before or after existing objects, be sure that the existing object is actually in the list. If you don't, strange things may happen.

Creation and destruction

CList

```
CList (short blockSize);
```

Constructor. `blockSize` specifies the number of slots to add to the array each time more memory is needed; the default value is `3`.

Copy

```
void *Copy();
```

Creates a copy of this CList object and returns a pointer to it.

Object I/O

PutItems

```
void PutItems (CStream& aStream);
```

Writes each object in the list to the stream.

GetItems

```
void GetItems (CStream& aStream);
```

Reads each object in the list from the stream.

CListIterator 67

Introduction

CListIterator lets you iterate through the elements of a CList array.

Heritage

Base Class	CArrayIterator
Derived Classes	None

Using CListIterator

CListIterator provides a straightforward and error-resistant way to loop through the items in a CList.

Note

CListIterator is implemented as a CPtrArrayIterator. For information on how to use CListIterator, refer to Chapter 87, “CPtrArrayIterator.”

---

67 CListIterator

CMBarChore 68

Introduction

CMBarChore is a chore that works with CBartender to redraw the menu bar.

Heritage

Base Class	CChore
Derived Classes	None

Using CMBarChore

The CBartender class uses a CMBarChore to redraw the menu bar after deleting menus from the menu bar. Your application should not need to use this class.

The CBartender's `DeleteFromBar` function creates a CMBarChore and assigns it as an urgent chore, so that the menu bar gets redrawn the next time through the event loop.

Data Members

This class has no data members.

Member Functions

Perform

```
void Perform (long *maxSleep);
```

Calls the Toolbox routine `DrawMenuBar` to redraw the menu bar.

CMenuDefProc 69

Introduction

CMenuDefProc is an abstract class that gives you an object-oriented interface for Macintosh menu definition procedures (`'MDEF'`s). The description of this class assumes that you're familiar with the Macintosh Menu Manager in general and with `'MDEF'`s in particular. If you need to learn more, see *Inside Macintosh Volume I*, Chapter 11, "The Menu Manager" and *Inside Macintosh Volume V*, Chapter 13, "The Menu Manager."

Heritage

Base Class	None
Derived Classes	CPaneMDEF

Using CMenuDefProc

CMenuDefProc lets you write object-oriented menu definition functions. To write an `'MDEF'` object, you need to create a derived class of CMenuDefProc or use one provided by the THINK Class Library.

CMenuDefProc uses a stub `'MDEF'` that contains a jump to a generic menu definition procedure and a reference to an `'MDEF'` object. The Menu Manager calls the generic menu definition procedure, which calls member functions to your `'MDEF'` object.

Creating the stub MDEF

The stub `'MDEF'` is a 10-byte data structure that looks like this:

```
typedef struct GenericMDEFRec
{
   short          JMPinstruction;
   VoidFunc       defProc;
   CMenuDefProc*  itsMenuDefProc;
};
```

To create the 'MDEF', use your favorite resource editor or resource compiler to create a 10-byte resource that contains these hex values:

```
4EF9 0000 0000 0000 0000
```

When you create the menu that uses your 'MDEF', be sure to specify the 'MDEF' ID. Otherwise, the Menu Manager uses the default 'MDEF'.

Apple reserves IDs 0 to 127 for definition procedures and recommends that you number your 'MDEF' in the range 128 to 4095. In the THINK Class Library, the convention is to give your 'MDEF' the same ID as the menu that you'll be using it with.

Writing the CMenuDefProc-derived class

Your object-oriented 'MDEF' must handle all the messages that a regular 'MDEF' handles. This means that your derived class needs to override all the functions of CMenuDefProc:

```
Constructor     DrawMenu        ChooseItem
SizeMenu        PlacePopup
```

Your derived class can define data members to hold whatever information your menu needs to do its work. Your constructor should allocate memory for the data members if necessary. Your derived class must call CMenuDefProc to set up the stub 'MDEF' resource.

If you want to use your 'MDEF' for more than one menu, be careful how you use data members. If the data members of the 'MDEF' control the state of the menu being displayed, you can use it for only one menu.

For example, the CPaneMDEF class derived from CMenuDefProc keeps two data members: one refers to the pane being displayed as a menu and the other refers to the tear-off menu. If you were to use the same 'MDEF' based on CPaneMDEF for two menus, both menus would display the same pane.

Using your CMenuDefProc-derived class

The most important thing to keep in mind about using a descendant of CMenuDefProc is that you must initialize it before the Menu Manager loads the menu associated with the object. If you don't, and you nevertheless load the menu (perhaps by calling either CApplication's SetUpMenus or CBartender's AddMenu), your application will crash.

You can find the description of GenericMDEF near the end of this chapter.

The sequence of events is as follows. `CMenuDefProc` loads the `'MDEF'` and sets up the fields so `defProc` points to a function called `GenericMDEF` and the `itsMenuDefProc` points to the current object. `GenericMDEF` is the "real" menu definition procedure that converts Menu Manager messages to calls to member functions of your `'MDEF'` object.

As it's loading the menu, the Toolbox routine `GetMenu` loads the `'MDEF'` and sends it an `mSizeMsg` message to find out the size of the menu. If you haven't initialized the `MDEF`, this call to the menu definition procedure will jump to random memory, and your program will crash.

Examples of MDEF objects

If you're including the menu in your application's `'MBAR'` resource, override CApplication's `SetUpMenus` function to create the `'MDEF'` object and initialize it. Then call CApplication's `SetUpMenus` function. You are probably overriding this function anyway to load resource-based menus like the Font menu and to set up the dimming and checking options for your menus.

Your `SetUpMenus` function should look like this:

```
void CMyApp::SetUpMenus ()
{
   CMyCustomMDEF *theMDEF;

   theMDEF = new CMyCustomMDEF(CustomID);
   inherited::SetUpMenus();
   // Load font menu,
   // set check & dim options
}
```

Your constructor (`CMyCustomMDEF` in this example) must call CMenuDefProc's constructor.

If you're using CBartender's `AddMenu` function to add the menu to the menu bar, you need to create and initialize the `'MDEF'` object before you call `AddMenu`. The next example shows a function that creates a tool menu which uses a custom `'MDEF'` and adds it to the end of the menu bar. Note that the example uses the THINK Class Library convention of giving the menu and `'MDEF'` the same ID.

This is what the function looks like:

```
void CPaintPane::AddToolMenu(void)
{
   CToolMDEF *theToolMDEF;

   theToolMDEF = new CToolMDEF(ToolMENUID);
   gBartender->AddMenu(ToolMENUID,
      TRUE, 0);
}
```

Data Members

This class has no data members.

Member Functions

Except for its constructor, all of CMenuDefProc's functions handle a Menu Manager message sent to a menu definition procedure. The descriptions given here of your derived class's functions are fairly complete. For details, however, be sure to consult *Inside Macintosh Volume I*, Chapter 11, "The Menu Manager," and *Inside Macintosh Volume V*, Chapter 13, "The Menu Manager."

CMenuDefProc
`CMenuDefProc ();`

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IMenuDefProc` for backward compatibility.

CMenuDefProc
`CMenuDefProc (short MDEFid);`

Constructor. This function loads the stub `'MDEF'` and stores the address of the generic `'MDEF'` function `GenericMDEF` in the `defProc` field. It stores the reference to this object in the `itsMenuDefProc` field.

IMenuDefProc
`void IMenuDefProc (short MDEFid);`

Initialization function compatible with previous release. May not be called if constructor has an argument.

Dispose

```
void Dispose ();
```

`Dispose` function compatible with previous release. Can only be used if `TCL_USE_DISPOSE` is defined. Calls `delete this`.

DrawMenu

```
void DrawMenu (MenuHandle macMenu,
   Rect *menuRect);
```

Draws the menu in response to a Menu Manager `mDrawMsg`. `menuRect` is given in global coordinates, the current port is the Window Manager port, and the clipping region is set to `menuRect`. Your `DrawMenu` function should check whether the menu is enabled and draw it in gray if it's not.

ChooseItem

```
void ChooseItem(MenuHandle macMenu,
   Rect *menuRect, Point hitPt,
   short *whichItem);
```

Chooses an item in response to a Menu Manager `mChooseMsg`. Both `hitPt` and `menuRect` are in global coordinates, and `whichItem` is the last item chosen. `whichItem` is initially 0. Your `ChooseItem` function should check that `hitPt` is within `menuRect`. If it is, it should unhighlight `whichItem`, highlight the new item (if it's not the same as `whichItem`), and set `whichItem` to the new item. If `hitPt` isn't in `menuRect`, you should unhighlight `whichItem` and set `whichItem` to 0.

SizeMenu

```
void SizeMenu (MenuHandle macMenu);
```

Sets the size of the menu in response to a Menu Manager `mSizeMsg` message. Your function should store the width of the menu in `macMenu`'s `menuWidth` field and the height in the `menuHeight` field. Note that the Menu Manager sends `mSizeMsg` to your menu soon after it has been loaded. See "Using your CMenuDefProc-derived class" earlier in this chapter for more details.

PlacePopUp

```
void PlacePopUp (MenuHandle macMenu,
   Rect *menuRect,
   Point hitPt, short *whichItem);
```

Determines the rectangle for a pop-up menu in response to a Menu Manager `mPopUpMsg`. `whichItem` contains the previously selected item, and `hitPt` contains the top-left corner of the pop-up menu (`hitPt.h` contains the top and `hitPt.v` contains the left). Your function should set `menuRect` to the rectangle the menu is displayed in. If the menu would scroll, set `whichItem` to the actual top item. *Inside Macintosh Volume V*, Chapter 13, "The Menu Manager" contains additional useful information.

Auxiliary Functions

CMenuDefProc uses an auxiliary, non-member function, `GenericMDEF`, to convert messages from the Menu Manager to member function calls. The source code to this function is in `GenericMDEF.c` in the `FW/Tearoffs` folder.

GenericMDEF

```
pascal void GenericMDEF (short theMessage,
   MenuHandle macMenu, Rect *menuRect,
   Point hitPt, short *whichItem);
```

As far as the Menu Manager is concerned, `GenericMDEF` is the "real" menu definition procedure. This routine calls the member functions of your object-oriented `MDEF`.

Be aware that the `GenericMDEF` only handles the four standard Menu Manager messages. If you write an `MDEF` that responds to user messages (as described in *Inside Macintosh Volume V*, Chapter 13, "The Menu Manager"), you will need to write your own version of `GenericMDEF` and write your constructor accordingly.

CMouseTask 70

Introduction

CMouseTask is an abstract class that implements mouse tracking.

Heritage

Base Classes	CTask CTableDragger
Derived Classes	You must define a derived class of this class.

Using CMouseTask

CMouseTask is an abstract class that lets you implement undoable mouse-related actions. For example, if you're writing a drawing application, you want to make sure that you can undo anything that the user moves or draws.

You don't have to use this class to implement mouse-related actions. You can always track the mouse yourself in your pane's `DoClick` function. If you do use CMouseTask to implement mouse actions, you don't have to make them undoable.

Defining a mouse task

To implement a mouse-tracking task, define a derived class of CMouseTask and override the `KeepTracking` and `EndTracking` functions. `KeepTracking` does whatever you want to happen while the mouse button is held down. `EndTracking` does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, the `KeepTracking` function might draw a gray outline that moves as you move the mouse. The `EndTracking` function would erase the rectangle from its old location and redraw it in the new location.

If you want to make your mouse task undoable, you need to store enough information in the object to undo the effects of mouse tracking. You must also override the `Undo` function (inherited from CTask) to use this information to undo the effects of the mouse task.

In the moving rectangle example, your `EndTracking` function might keep the old location of the rectangle in a data member. The `Undo` function would erase the rectangle from its current location and redraw it again in the old location.

Using a mouse task

You use mouse tasks in the `DoClick` function of a pane. Create an instance of your mouse task, initialize it, and pass it as an argument to the `TrackMouse` function of your pane. A part of your `DoClick` function might look like this:

```
CShapeMover *myMover;

myMover = new CShapeMover(UNDOMoverStr);
this->TrackMouse(myMover, hitPt,
   &pinRect);
itsSupervisor->Notify(myMover);
```

The `TrackMouse` function calls your mouse task's `BeginTracking` function to give you an opportunity to adjust the starting point. As long as the mouse button is held down, `TrackMouse` repeatedly calls the `KeepTracking` function of your mouse task. When the mouse button is released, `TrackMouse` calls your mouse task's `EndTracking` function.

The value you pass to your constructor is the index of a string in the `'STR#' 130` resource that describes your task. The document function `UpdateUndo` uses this string for the wording of the **Undo** command in the **Edit** menu. If your task is not undoable, you can use any value and ignore it.

After you've tracked the mouse, you can pass the task as an argument to the document's `Notify` function. The document stores the task in its `lastTask` data member. When you choose **Undo** from the **Edit** menu, the document calls the task's `Undo` function to undo the effects of mouse tracking.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CMouseTask

```
CMouseTask (short aNameIndex = 0);
```

Constructor. `aNameIndex` is the index for the undo/redo string in the `'STR#' 130` resource.

IMouseTask

```
void IMouseTask (short aNameIndex);
```

Initialization function compatible with previous release. May not be called if constructor has an argument.

Mouse tracking

BeginTracking

```
void BeginTracking (LongPt *startPt);
```

Mouse tracking is starting. The document's `TrackMouse` function calls this function at the beginning of mouse tracking. `startPt` is the starting point (in local coordinates). If your mouse task derived class doesn't alter the starting point, you don't need to override this function.

KeepTracking

```
void KeepTracking (LongPt *currPt,
   LongPt *prevPt, LongPt *startPt);
```

Mouse tracking is under way. The document's `TrackMouse` function calls this function repeatedly as long as the mouse button is held down. `currPt` is the current mouse location. `prevPt` is the previous mouse location. `startPt` is a the original mouse location. Your derived class must override this function.

EndTracking

```
void EndTracking (LongPt *currPt, LongPt *prevPt,
   LongPt *startPt);
```

Mouse tracking is over. The document's `TrackMouse` function calls this function when the mouse button is released. `currPt` is the current mouse location. `prevPt` is the previous mouse location. `startPt` is the original mouse location. Your derived class must override this function. If your mouse task is undoable, this is the function that will actually undo what was done. You should use data

members to store all the information you will require to undo what was done.

Note

If you're implementing an undoable mouse task, you'll need to override the Undo function as well.

Class Resources

Resource	Description
'STR#' 130	List of strings that describe undoable tasks. For example, if you're implementing a mouse task that moves graphic images, the string for that task might be "move." The item in the **Edit** menu would then read "Undo move" or "Redo move."

CPane 71

Introduction

CPane is an abstract class that defines a drawing area within a window or within another pane. In the THINK Class Library, all drawing is done within a pane. Each pane has its own drawing environment and coordinate system.

Heritage

Base Class	CView
Derived Classes	CControl
	CPanorama
	CRadioGroupPane
	CScrollPane
	CSizeBox
	CPopupPane
	CSwissArmyButton
	CSubViewDisplayer
	CIconPane

Using CPane

Use a pane when you need a non-scrolling area to draw in within a window. If you need a scrollable area, use CPanorama and CScrollPane. If you need a pane for displaying text, see Chapter 14, “CAbstractText,” or Chapter 49, “CEditText.”

Your application must define a derived class of CPane (or one of its derived classes) to draw in a window. In your derived class, you need to override the `Draw` member function. If your pane class handles mouse clicks (most do), you should also override the `DoClick` function.

Your pane class should have a constructor. If your class defines new data members, this is the function that initializes them. The

supervisor of a pane should be either the pane that encloses it, the window it appears in, or the director its window belongs to.

The constructor is where you set the pane's characteristics and its location within its enclosure. If you want your pane, or any of the panes it encloses, to receive clicks, be sure to call `SetWantsClicks(TRUE)`; otherwise, mouse clicks in your pane will be ignored.

If your pane allocates memory, you should also define a destructor to deallocate it.

The `Draw` function tells your pane to draw its contents. The port and frame coordinate system are set up correctly before `Draw` is called. The clip region is set so that drawing will be clipped to the pane's aperture (the visible portion of the frame). If your pane uses long coordinates, you need to map from frame coordinates to QuickDraw coordinates before you do any drawing.

When the user clicks in the pane, its `DoClick` function is called. Your `DoClick` function can either handle the mouse click itself, or it can create a task and call its `TrackMouse` function. To learn more about mouse tracking in a pane, see Chapter 70, "CMouseTask."

Coordinate systems in panes

Panes use two of the coordinate systems in the THINK Class Library: frame coordinates and QuickDraw coordinates.

Frame coordinates provide a local coordinate system for a pane. Units in frame coordinates are in pixels, and the point (0, 0) is usually the upper-left corner of the pane. If the pane moves within its enclosure, the coordinate system does not change; the upper-left corner is still (0, 0). The only time this origin point changes is when you scroll the pane. Each pane can choose to use long or short coordinates.

All drawing and mouse tracking is done in QuickDraw coordinates. This is the coordinate system that the Macintosh Toolbox uses for its drawing operations. QuickDraw coordinates are only valid after a call to `Prepare`. The relationship between QuickDraw coordinates and the other coordinate systems depends on whether a pane is using long-frame or short-frame coordinates.

Short coordinates map directly to QuickDraw coordinates. Each element in a short coordinate uses 16-bit values, so a pane that uses short coordinates is limited to the rectangle (–32768, –32768, 32767, 32767). Long coordinates layer a 32-bit coordinate system on top of the QuickDraw 16-bit coordinates. The long coordinate system lets you use a much larger coordinate area for your pane. Since all drawing takes place in QuickDraw coordinates, you have to map the long coordinates to QuickDraw coordinates when you draw in a pane.

If a pane uses long coordinates, the data member `fUsingLongCoord`, which CPane inherits from CView, is `TRUE`; otherwise, `fUsingLongCoord` is `FALSE`. The default is to not use long coordinates.

The THINK Class Library uses the types `LongRect` and `LongPt` for both long and short coordinates. You'll notice that most of the descendants of CView that work with points and rectangles use these types. These two types are defined as follows in `LongCoordinates.h`:

```
typedef struct LongPt
{
   long v, h;
} LongPt;

typedef struct LongRect
{
   long top, left, bottom, right;

} LongRect;
```

If a pane uses short coordinates, frame coordinates and QuickDraw coordinates are identical. Therefore, the values stored in a `LongRect` or in a `LongPt` are in QuickDraw coordinates. To use them with QuickDraw routines, however, you need to convert them to the QuickDraw types `Rect` and `Point`. The THINK Class Library provides several utility routines to do these conversions. For a complete listing, see the section "Long Coordinate Utilities" in Chapter 127, "TCL Utilities."

Drawing in panes

To draw in a pane, use the standard QuickDraw routines. The pane's `Prepare` function sets up the QuickDraw port before `Draw` is called. If your pane uses short coordinates, the coordinate system is set up correctly. If your pane uses long coordinates, you need to

transform frame coordinates to QuickDraw coordinates before you draw. The CPane functions `FrameToQD` and `FrameToQDR` convert frame points and rectangles to QuickDraw points and rectangles.

Here's an example. The following `Draw` function draws a line from the upper-left corner of the pane to the lower-right corner, and then draws a rectangle inset 10 pixels from the edges of the frame.

In THINK C, the function looks like this:

```
void CCoolPane::Draw (Rect *area)
{
   Rect theRect;

   MoveTo(frame.left, frame.top);
   LineTo(frame.right, frame.bottom);
   LongToQDRect(&frame, &theRect);
   InsetRect(&theRect, 10, 10);
   FrameRect(&theRect);
}
```

Note that to draw the line from one corner to the other, you can use the values directly since they're QuickDraw values. To perform an operation on a rectangle, however, you have to convert the `LongRect` into a QuickDraw `Rect`.

If a pane uses long coordinates, all coordinates are in frame coordinates. Before you can do any drawing, you need to map those values to the QuickDraw space on the screen using the CPane functions `FrameToQD` and `FrameToQDR`.

For example, imagine you have a pane that uses long coordinates, and you want its `Draw` function to display the text "Hello World." and then draw a rectangle inset 10 pixels from the frame. The location of the string is stored in a data member named `stringLoc`, which is declared as a `LongPt`. The location in `stringLoc` could be well outside QuickDraw coordinate space—it might be at (100000, 100000).

You might write the `Draw` function as follows:

```
void CMyLongPane::Draw (Rect *area)
{
   Point qdLoc;
   Rect qdRect;

   FrameToQD(&stringLoc, &qdLoc);
   MoveTo(qdLoc.h, qdLoc.v);
   DrawString("\pHello World.");
   FrameToQDR(&frame, &qdRect);
   InsetRect(&qdRect, 10, 10);
   FrameRect(&qdRect);
}
```

Keep in mind the difference between converting long-coordinate structures to QuickDraw structures and mapping long-coordinate structures to QuickDraw structures. When you're using short coordinates, frame coordinates are the same as QuickDraw coordinates, so all you have to do is translate one data structure to another. When you're using long coordinates, you need to map a portion of long coordinate space into QuickDraw space so you can draw in it.

Data Members

CPane defines these data members:

Data member	Type	Description
`width`	`short`	Horizontal size in pixels
`height`	`short`	Vertical size in pixels
`hEncl`	`long`	Horizontal location in enclosure
`vEncl`	`long`	Vertical location in enclosure
`hSizing`	`SizingOption`	Horizontal sizing option
`vSizing`	`SizingOption`	Vertical sizing option
`autoRefresh`	`Boolean`	Refresh after a resize
`frame`	`LongRect`	Area for displaying the pane that defines the frame coordinates
`aperture`	`LongRect`	Active drawing area of the pane
`hOrigin`	`long`	Window left in frame coordinates

Data member	Type	Description
vOrigin	long	Window top in frame coordinates
itsEnvironment	CEnvironment*	Drawing environment
printClip	ClipOption	The region to clip to when printing
printing	Boolean	Is printing in progress
itsBorder	CPaneBorder*	Border of this pane
enabled	Boolean	When FALSE, pane (and border, if any) is drawn "grayed out"
itsLastTask	Ctask*	Last CTask created by the pane
cPageArea	Rect	Area of page being printed

Member Functions

Creation and destruction

CPane

```
CPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth = 0, short aHeight = 0,
   short aHEncl = 0, short aVEncl = 0,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY);
```

Constructor. Almost all of the derived classes of CPane use the same arguments in their constructors.

`anEnclosure` is the enclosing view that contains this pane. Typically, the enclosure is either a window or another pane. If your pane is a panorama, its enclosure should be a scroll pane.

`aSupervisor` is the bureaucrat that handles all the commands that this pane won't. Typically, the supervisor is the document (or director) associated with this pane's window.

`aWidth` and `aHeight` are the width and height of the pane in pixels. `aHEncl` and `aVEncl` are the horizontal and vertical position of the pane within its enclosure.

The `aHSizing` and `aVSizing` parameters specify what happens to the pane when the size of its enclosure changes. The length and

height of a pane changes relative to its original position in the enclosing pane. `aHSizing` can have these values:

aHSizing value	Meaning
`sizFIXEDLEFT`	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as when it was originally placed.
`sizFIXEDRIGHT`	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
`sizELASTIC`	The width of the pane grows and shrinks by the same amount as the enclosing pane.

Table 71-1 Values for aHSizing

`aVSizing` can have these values:

aVSizing value	Meaning
`sizFIXEDTOP`	The upper edge of the pane is always the same number of pixels from the top edge of the enclosing pane as when it was originally placed.
`sizFIXEDBOTTOM`	The lower edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The upper and lower edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
`sizELASTIC`	The height of the pane grows and shrinks by the same amount as the enclosing pane.

Table 71-2 Values for aVSizing

A couple of examples will make this clear. A vertical scroll bar in a window would be `sizFIXEDRIGHT` horizontally and `sizELASTIC` vertically. It has a fixed horizontal length and remains anchored to the right edge of the window. Vertically, it stretches and contracts

You may want to look at Figure 7-2 in Chapter 7, "Programming with the THINK Class Library."

with the height of the window. A status box in the lower-left corner of a window would be `sizFIXEDLEFT` horizontally and `sizFIXEDBOTTOM` vertically. It has a constant size and remains anchored to the lower-left corner of the window.

CPane

```
CPane ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IPane` for backward compatibility.

~CPane

```
~CPane ();
```

Destructor. Deletes the environment and border, if any. Also calls `NotifyClean(NULL)` to ensure that no task is left with a dangling pointer to this pane.

IPane

```
void IPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialization function provided for backward compatibility. May not be called if the constructor was passed arguments.

IViewRes

```
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initializes a pane from a resource template. `rType` is the resource type for the CView-derived class you want to initialize. `resID` is the resource ID of the resource. `anEnclosure` and `aSupervisor` are the same as for `IBorder`. This function is inherited from CView.

To initialize a pane from a resource file, use a `'Pane'` resource.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally for initializing from a resource template. Each class derived from CView overrides this function to use its own resource template.

IPaneX

```
void IPaneX ();
```

Performs extra initialization for a pane.

Accessing

SetFrameOrigin

```
void SetFrameOrigin (long fLeft, long fTop);
```

Sets the coordinates of the upper-left corner of the pane's frame to determine the frame coordinates of the pane.

GetFrame

```
void GetFrame (LongRect *theFrame);
```

Returns the frame of the pane in `theFrame`. `theFrame` is the rectangle that encloses the pane in pane coordinates. Usually the upper-left edge of this rectangle is at (0, 0).

GetLengths

```
void GetLengths (short *theWidth,
   short *theHeight);
```

Returns the width of the pane in `theWidth` and the height of the pane in `theHeight`.

GetOrigin

```
void GetOrigin (long *theHOrigin,
   long *theVOrigin);
```

Returns the origin of the pane in `theHOrigin` and `theVOrigin`. The origin of a pane is the upper-left corner of the window in frame coordinates, or the distance from the upper-left corner of the window to the (0, 0) of the pane. You should not use or override this function.

GetAperture

```
void GetAperture (LongRect *theAperture);
```

Returns the aperture of the pane in `theAperture`. The aperture is the visible portion of a pane where drawing can occur. The aperture is given in frame coordinates.

Contains

```
Boolean Contains (Point windPt);
```

Returns `TRUE` if `windPt` is in the aperture of the pane. `windPt` is given in window coordinates.

ReallyVisible

```
Boolean ReallyVisible ();
```

Returns `TRUE` if the pane is visible and within QuickDraw space. This function actually returns `TRUE` if the pane and its enclosure is potentially visible. The pane may not actually be on the screen. A pane is on the screen if `ReallyVisible` returns `TRUE` and the aperture isn't empty.

GetPixelExtent

```
void GetPixelExtent (long *hExtent,
   long *vExtent);
```

Returns the dimensions of the pane in pixels.

SetPrintClip

```
void SetPrintClip (ClipOption aPrintClip);
```

Specifies the print option to use when printing. `aPrintClip` can be one of the following:

aPrintClip value	Meaning
`clipAPERTURE`	Print only what is visible
`clipFRAME`	Print the contents of the frame
`clipPAGE`	Print to fill the page

The default is `clipFRAME`.

GetWindow

```
CWindow *GetWindow ();
```

Returns the window that encloses the pane. Note that this function returns a window object, not a Macintosh window.

SetBorder

```
void SetBorder (CPaneBorder *aBorder);
```

Sets the border for this pane. For more information about borders, see Chapter 72, "CPaneBorder."

SetResBorder

```
void SetResBorder (short resID);
```

Creates a border for this pane from a `'PBrd'` resource whose ID is `resID`.

GetBorder

```
CPaneBorder *GetBorder ();
```

Returns the border for this pane.

GetHelpResID

```
short GetHelpResID ();
```

Returns the ID of the `'hrct'` resource that has the Balloon Help information for this pane. The resource ID is actually stored in the window object that encloses this pane. For more information about help resources for panes, see "Using Balloon Help with views" in Chapter 121, "CView."

Appearance

Show

```
void Show ();
```

Shows the pane if it was hidden. The default function calls the pane's `Refresh` function after making it visible.

Hide

```
void Hide ();
```

Hides the pane if it was visible. The default function calls the pane's `Refresh` function before hiding it.

Size and location

Place

```
void Place (long hEncl, long vEncl,
   Boolean redraw);
```

Places the pane at the point `hEncl`, `vEncl` of its enclosure. `hEncl` and `vEncl` are given in the coordinate system of the enclosure. If `redraw` is `TRUE`, the function redraws the pane after moving it.

Offset

```
void Offset (long hOffset, long vOffset,
   Boolean redraw);
```

Offsets the pane by `hOffset` pixels horizontally and `vOffset` pixels vertically. If `redraw` is `TRUE`, the function redraws the pane after moving it.

ChangeSize

```
void ChangeSize (Rect *delta, Boolean redraw);
```

Changes the size of the pane. The values of each field of the `delta` rectangle specify how each side should change. Positive values indicate down and to the right. Negative values indicate up and to the left.

AdjustToEnclosure

```
void AdjustToEnclosure (Rect *deltaEncl);
```

Adjusts the size or location of the pane when the enclosure has moved or changed size. Do not override or use this function.

AdjustHoriz

```
void AdjustHoriz (Rect *deltaEncl, Rect *delta,
   short *offset, Boolean *moved,
   Boolean *sized);
```

Adjusts the horizontal size or location of a pane. You should not override or use this function.

AdjustVert

```
void AdjustVert (Rect *deltaEncl, Rect *delta,
   short *offset, Boolean *moved,
   Boolean *sized);
```

Adjusts the vertical size or location of a pane. You should not override or use this function.

EnclosureScrolled

```
void EnclosureScrolled (long hOffset,
   long vOffset);
```

Adjusts the location of a pane after its location has scrolled. This function affects the pane only if it is sticky in the direction it is being moved. You should not use or override this function.

Adapting

FitToEnclosure

```
void FitToEnclosure (Boolean horizFit,
   Boolean vertFit);
```

Makes the frame of the pane fit the interior of its enclosure in either the vertical or horizontal direction. If `horizFit` is `TRUE`, the left edge and width of the pane's frame change to coincide with the enclosure's interior. If `vertFit` is `TRUE`, the upper edge and height of the pane's frame change to coincide with the enclosure's interior. `FitToEnclosure` calls the enclosure's `GetInterior` function to determine the interior of the pane. `FitToEnclosure` does not redraw the pane.

FitToEnclFrame

```
void FitToEnclFrame (Boolean horizFit,
   Boolean vertFit);
```

Fits the frame of the pane to the frame of its enclosure in either the vertical or horizontal direction. If `horizFit` is `TRUE`, the left edge and the width of the pane's frame change to coincide with the enclosure's frame. If `vertFit` is `TRUE`, the upper edge and the height of the pane's frame change to coincide with the enclosure's frame. `FitToEnclFrame` does not redraw the pane.

CenterWithinEnclosure

```
void CenterWithinEnclosure (Boolean horizCenter,
   Boolean vertCenter);
```

Centers the pane within its enclosure horizontally or vertically. Only the location of the pane changes. The size of the pane does not change. `CenterWithinEnclosure` does not redraw the pane.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the contents of the pane. The `area` parameter specifies the portion of the pane that needs to be redrawn. Your derived class must override this function.

If the pane is not using long coordinates, the `area` parameter is given in frame coordinates, which in this case are the same as QuickDraw coordinates.

If the pane uses long coordinates, `area` is given in QuickDraw coordinates. Your `Draw` function can use `QDToFrame` or `QDToFrameR` to convert from QuickDraw coordinates to frame coordinates and `FrameToQD` and `FrameToQDR` to convert from frame coordinates to QuickDraw coordinates.

DrawAll

```
void DrawAll (Rect *area);
```

Draws the pane and all its subviews, and any borders that the pane or subviews may have. Use this function when you want to force the entire pane to be redrawn without waiting for an update event. Scrolling is a good example: You want to redraw the pane as soon as it has scrolled instead of waiting for an update event. This function prepares the pane before calling the `Draw` function of the pane and its subpanes. You should not override this function. Look at the implementation of CPanorama's `Scroll` function for an example of using `DrawAll`.

Refresh

```
void Refresh ();
```

Forces the pane to redraw itself on the next update event. The default function calls the pane's `RefreshLongRect` function using the frame as the area.

RefreshRect

```
void RefreshRect (Rect *area);
```

Forces a portion of the pane to redraw itself on the next update event. `area` is a `Rect` in frame coordinates. Only the portion of the pane that's visible will actually be redrawn.

RefreshLongRect

```
void RefreshLongRect (LongRect *area);
```

Forces a portion of the pane to redraw itself on the next update event. `area` is a `LongRect` in frame coordinates. Only the portion of the pane that's visible will be redrawn.

RefreshBorder

```
void RefreshBorder ();
```

Forces the pane's border to redraw itself on the next update event.

Printing

Paginate

```
void Paginate (CPrinter *aPrinter,
   short pageWidth, short pageHeight);
```

Calculates pagination for the pane. By default, panes print only on one page. The document that a pane belongs to calls this member function. For examples of more complex pagination, see Chapter 14, "CAbstractText," and Chapter 74, "CPanorama."

AboutToPrint

```
void AboutToPrint (short *firstPage,
   short *lastPage);
```

Prepares the pane for printing. Your derived class can override this function to perform any actions necessary to prepare for printing.

PrintPage

```
void PrintPage (short pageNum, short pageWidth,
   short pageHeight, CPrinter *aPrinter);
```

Prints the specified page of the pane. By default, panes have only one page. `aPrinter` is usually supplied by the document that this pane belongs to. If your derived class supports multi-page documents, you must override this function to determine which part of the pane to draw.

DonePrinting

```
void DonePrinting ();
```

Cleans up after printing. If you want to take any action when printing is done, override this function.

PrepareToPrint

```
void PrepareToPrint ();
```

Initializes the coordinate system and the clipping region before printing. The default function sets the clipping region to the region described in the `printClip` data member. You can set `printClip` with the `SetPrintClip` function.

Calibration

Prepare

```
void Prepare ();
```

Prepares the pane for drawing. `Prepare` sets up the port and the QuickDraw coordinates for the pane, converting long coordinates to QuickDraw coordinates if necessary. It also sets the clipping region to the aperture (the visible portion of the pane) so that drawing is constrained to the visible portion of the pane. If the pane has an associated environment, `Prepare` calls the environment's `Restore` function. If the pane is being printed, the default function calls the pane's `PrepareToPrint` function instead.

The `Prepare` function inherited from CView sets `cPreparedView` to this view. If this view's `Prepare` function is called subsequently, `Prepare` avoids setting up the drawing environment all over again.

RestoreEnvironment

```
void RestoreEnvironment ();
```

Restores the pane's drawing environment. If the pane has an associated drawing environment, this function calls the environment's `Restore` function.

CalcFrame

```
void CalcFrame ();
```

Calculates the coordinates of the pane's frame based on the frame's width and height and its location within its enclosure. Generally, the base classes of your CPane-derived classes will call this function. You should not use or override this function unless your derived class needs something other than (0, 0) at its upper-left corner.

ResizeFrame

```
void ResizeFrame (Rect *delta);
```

Adjusts the frame when the size of the pane changes. The `delta` rectangle specifies how each side changes. Positive values indicate down and to the right. Negative values indicate up and to the left. The default function always sets the upper-left corner of the frame to (0, 0).

You should not need to use or override this function unless your derived class requires that the upper-left corner of the frame be something other than (0, 0). See the implementation of `ResizeFrame` in Chapter 74, "CPanorama," for an example.

CalcAperture

```
void CalcAperture ();
```

Calculates the visible (or drawable) portion of a pane. The aperture of a pane is the area that is not obscured by the bounds of the enclosing view. You should not use or override this function. To obtain the aperture, use `GetAperture`, described earlier in this chapter.

Cursor

TrackMouse

```
void TrackMouse (CMouseTask *theTask,
   LongPt *startPt, LongRect *pinRect);
```

To learn more about mouse tracking and mouse tasks, see Chapter 70, "CMouseTask."

Tracks the mouse in this view. `theTask` is a mouse task that you create. `startPt` is the starting point in frame coordinates. `pinRect` is a constraining rectangle in frame coordinates. This function is usually called from within a `DoClick` function.

The default function calls:

- The view's `Prepare` function.
- `theTask->BeginTracking`.
- `theTask->KeepTracking` as long as the mouse button is held down. The current point is constrained to `pinRect`. (Your task-derived class must override the `KeepTracking` function.)
- `theTask->EndTracking`. (Your task-derived class must override this function as well.)

Coordinate transformation

WindToFrame

```
void WindToFrame (Point windPt,
   LongPt *framePt);
```

Converts the point `windPt` from window coordinates to frame coordinates and places the converted point in `framePt`.

WindToFrameR

```
void WindToFrameR (Rect *windRect,
   LongRect *frameRect);
```

Converts the rectangle `windRect` from window coordinates to frame coordinates and places the converted rectangle in `frameRect`.

FrameToWind

```
void FrameToWind (LongPt *framePt,
   Point *windPt);
```

Converts the point `framePt` from frame coordinates to window coordinates and place the converted point in `windPt`.

FrameToWindR

```
void FrameToWindR (LongRect *frameRect,
   Rect *windRect);
```

Converts the rectangle `frameRect` from frame coordinates to window coordinates and places the converted rectangle in `windRect`.

EnclToFrame

```
void EnclToFrame (LongPt *thePoint);
```

Converts a point from the frame coordinates of its enclosure to the frame coordinates of the pane.

EnclToFrameR

```
void EnclToFrameR (LongRect *theRect);
```

Converts a rectangle from the frame coordinates of its enclosure to the frame coordinates of the pane.

FrameToEncl

```
void FrameToEncl (LongPt *thePoint);
```

Converts a point from frame coordinates to the frame coordinates of its enclosure.

FrameToEnclR

```
void FrameToEnclR (LongRect *theRect);
```

Converts a rectangle from frame coordinates to the frame coordinates of its enclosure.

FrameToGlobalR

```
void FrameToGlobalR (LongRect *frameRect,
   Rect *globalRect);
```

Converts the rectangle `frameRect` from frame coordinates to global coordinates and places the converted rectangle in `globalRect`.

QDToFrame

```
void QDToFrame (Point qdPoint, LongPt *framePt);
```

Converts `qdPoint` from QuickDraw coordinates to frame coordinates, and puts the result in `framePt`. `qdPoint` is assumed to be in the portion of QuickDraw space that the pane is mapping to. If the pane is not using long coordinates, the values in `qdPoint` and `framePt` are the same.

QDToFrameR

```
void QDToFrameR (Rect *qdRect,
   LongRect *frameRect);
```

Converts `qdRect` from QuickDraw coordinates to frame coordinates and puts the result in `frameRect`.

FrameToQD

```
void FrameToQD (LongPt *framePt, *Point qdPt);
```

Converts `framePt` from frame coordinates to QuickDraw coordinates and puts the results in `qdPt`.

FrameToQDR

```
void FrameToQDR (LongRect *frameRect,
   Rect *qdRect);
```

Converts `frameRect` from frame coordinates to QuickDraw coordinates and puts the result in `qdRect`.

SectAperture

```
Boolean SectAperture (LongRect *srcRect,
   Rect *destRect);
```

Clips the `srcRect` (in frame coordinates) to the pixels visible through the aperture and returns the result in a QuickDraw rectangle, `destRect`. If `destRect` is not empty, this function returns `TRUE`.

Enable/disable

GetEnabled

```
Boolean GetEnabled ();
```

Returns the value of the `enabled` data member.

SetEnabled

```
void SetEnabled (Boolean isEnabled);
```

Sets the value of the `enabled` data member. When `enabled` is `FALSE`, the pane is drawn "grayed out." `enabled` has no effect on the behavior of CPane objects. Special behavior when disabled is the responsibility of derived classes. For an example, see Chapter 43, "CDialogText."

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Private

CPaneX

```
void CPaneX ();
```

Performs common initialization.

71 CPane

CPaneBorder 72

Introduction

CPaneBorder is a class that draws a border around a pane.

Heritage

Base Class	None
Derived Classes	None

Using CPaneBorder

CPaneBorder draws a border around a pane. You can choose from several types, including a rectangle, a rectangle with a shadow, and a rounded rectangle. If you need a different type, create a derived class of CPaneBorder. This class doesn't use any coordinate system. Its parameters are described as offsets from the pane's frame.

To give a pane a border, create a CPaneBorder and call its `SetBorder` member function with your border as the argument; the pane takes care of drawing it. When the size of the pane changes, the size of the border changes automatically. If you want to access the border to change it, call the pane's `GetBorder` function.

When you create a `CPaneBorder`, you supply its shape as an argument to the constructor. These are the available shapes:

Name	Description
`kBorderFrame`	Rectangle
`kBorderRoundRect`	Rounded rectangle
`kBorderOval`	Oval
`kBorderLeft`	Line on left side
`kBorderTop`	Line on top side
`kBorderRight`	Line on right side
`kBorderBottom`	Line on bottom
`kBorderNone`	No border

You can combine `kBorderLeft`, `kBorderTop`, `kBorderRight`, and `kBroderBottom` to create new shapes. For example, `kBorderTop + kBorderRight` gives you a border with lines on the top and right of the pane. To create yet more shapes, you need to create a derived class of CPaneBorder.

After you create your border, you can change its attributes, such as the width of the border outline, the distance between the border outline and the frame, and whether there's a shadow. Figure 72-1 shows you the data members that control these attributes.

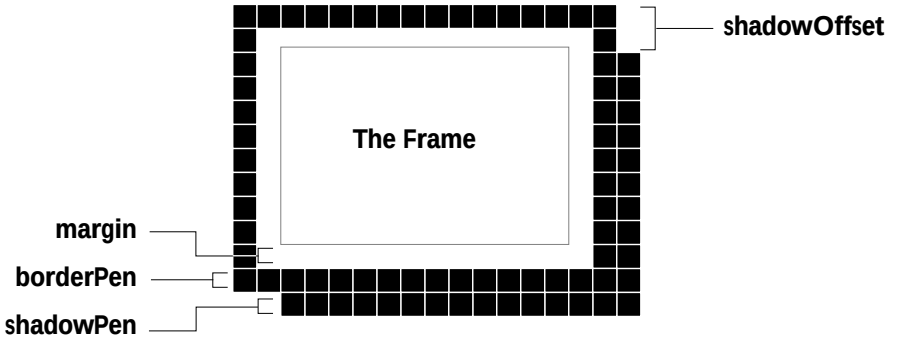


Figure 72-1 A pane border

The following table describes what these data members control and the member functions to set them:

Data member	Member function	Description
`margin`	`SetMargin`	Distance between the frame and the border
`borderPen`	`SetPenSize`	Width of the pen used to draw the border
`doShadow`	`SetShadow`	TRUE, if this border has a shadow
`shadowPen`	`SetShadow`	Width of the pen used to draw the shadow
`shadowOffset`	`SetShadow`	Distance between the border and its shadow

The following table describes other data members that control the border's appearance:

Data member	Member function	Description
borderFlags	SetBorderFlags	The shape of the border
penpat	SetPattern	The pattern used to draw the border and shadow
roundDiameter	SetRounding	The roundness of the border's corners, if the shape is a rounded rectangle

Data Members

The following are protected data members defined by CPaneBorder:

Data member	Type	Description
borderFlags	long	The shape of the border
borderPen	Point	The width of the pen used to draw the border
shadowOffset	Point	The distance between the border and its shadow
shadowPen	Point	The width of the pen used to draw the shadow
doShadow	Boolean	TRUE if shadow is drawn
roundDiameter	Point	The roundness of the border's corners, if the shape is a rounded rectangle
penPat	Pattern	The pattern used to draw the border and shadow
margin	Rect	The distance between the frame and the border

Member Functions

Creation and destruction

CPaneBorder

```
CPaneBorder (long borderFlags);
```

Constructor. The borderFlags describe the shape of the border. borderFlags may be one of: kBorderNone, kBorderOval,

kBorderRoundRect, kBorderFrame; or one or more of: kBorderLeft, kBorderTop, kBorderRight, kBorderBottom. All the defaults may be changed via member functions.

CPaneBorder

```
CPaneBorder ();
```

Default constructor. Implicitly called when an object is created by new_by_name. Can also be used in combination with IPaneBorder for backward compatibility.

~CPaneBorder

```
~CPaneBorder( );
```

Destructor.

IPaneBorder

```
void IPaneBorder (long borderFlags);
```

Initialization function compatible with previous release. Should not be called if the constructor has an argument.

IResPaneBorder

```
void IResPaneBorder (short resID);
```

Initializes a border from a 'PBrd' resource.

Accessing

SetPattern

```
void SetPattern (ConstPatternParam aPattern);
```

Sets the pattern used for drawing the border's outline and its shadow.

GetPattern

```
void GetPattern (Pattern *aPattern);
```

Returns the current pattern in aPattern.

SetBorderFlags

```
void SetBorderFlags (long aBorderFlags);
```

Sets the border flags. aBorderFlags can have one of these values: kBorderNone, kBorderOval, kBorderRoundRect,

kBorderFrame; or any combination of these: kBorderLeft, kBorderTop, kBorderRight, kBorderBottom.

GetBorderFlags

```
long GetBorderFlags ();
```

Returns the current border flags.

SetPenSize

```
void SetPenSize (short penWidth,
   short penHeight);
```

Sets the pen size used to draw the border.

GetPenSize

```
void GetPenSize (short *penWidth;
   short *penHeight);
```

Returns the pen size used to draw the border.

SetShadow

```
void SetShadow (short hOffset, short vOffset,
   short width, short height);
```

Sets the shadow attributes. hOffset and vOffset specify the distance of the shadow from the border outline. width and height specify the pen size used to draw the shadow. This function also sets doShadow to TRUE.

GetShadow

```
void GetShadow (short *hOffset, short *vOffset,
   short *width, short *height);
```

Returns the current shadow attributes.

SetRounding

```
void SetRounding (short hDiameter,
   short vDiameter);
```

Sets the roundness of the corners of rounded rectangles. hDiameter and vDiameter are used as parameters to FrameRoundRect. This function also sets the border style to kBorderRoundRect, if that has not already been done.

GetRounding

```
void GetRounding (short *hDiameter,
   short *vDiameter);
```

Returns the current rounding diameters in `hDiameter` and `vDiameter`.

SetMargin

```
void SetMargin (Rect *aMargin);
```

Sets the distance between the border outline and the frame. `aMargin` is a `Rect` with positive offsets for each side of the border. For example, to have a one-pixel margin on all sides, use (1, 1, 1, 1).

GetMargin

```
void GetMargin (Rect *aMargin);
```

Returns the current border margin in `aMargin`.

Drawing

DrawBorder

```
void DrawBorder (Rect *paneFrame);
```

Draws the border. `paneFrame` is the pane's frame in the pane's current coordinates.

Calibration

CalcBorderRect

```
void CalcBorderRect (Rect *paneFrame);
```

Returns the area for the frame and its border in `paneFrame`. `paneFrame` is the pane's frame in the pane's current coordinates. This function adds the size of the border, including the margin and shadow on all sides, to the area of the frame.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Private

CPaneBorderX

```
void CPaneBorderX ();
```

Private function; does common construction.

CPaneMDEF 73

Introduction

CPaneMDEF is an abstract class that lets you display any pane as a Macintosh menu. The class includes member functions for handling tear-off menus.

Heritage

Base Class	CMenuDefProc
Derived Classes	CSelectorMDEF

Using CPaneMDEF

For information about CTearOffMenu, see Chapter 117, "CTearOffMenu."

CPaneMDEF's base class, CMenuDefProc, lets you write object-oriented menu definition procedures. This class and its descendants take that idea a step further to let you use THINK Class Library panes as menus. The pane that you use in a CPaneMDEF-derived class should belong to a CTearOffMenu-derived object.

A tear-off menu is a window that appears to float above all the other windows in your application. In that environment, the pane is part of the visual hierarchy, and it will behave just like any other pane in a window.

When it appears as a custom menu, the pane isn't really in the visual hierarchy at all. As far as the pane is concerned, its enclosure is the floating window it appears in when it's torn off. When you draw a custom menu in your `Draw` function, you have to set up the QuickDraw environment explicitly by calling `SetupQuickDraw`. This function changes the QuickDraw coordinate system so that drawing takes place in pane coordinates. Be sure to call `RestoreQuickDraw` before returning from `Draw`.

Because this class doesn't have a `ChooseItem` function, you need to create a derived class of CPaneMDEF that handles item selection.

The class CSelectorMDEF is a derived class of CPaneMDEF that lets you use selectors as menus. It's also a good class to study if you want to learn how to write a `ChooseItem` function. Your `ChooseItem` function should also use `SetupQuickDraw` and `RestoreQuickDraw`.

Data Members

CPaneMDEF defines these data members:

Data member	Type	Description
`itsPane`	`CPane*`	The pane to display as a menu
`itsTearOffMenu`	`CTearOffMenu*`	The torn-off menu
`savePort`	`GrafPtr`	Used internally to save the grafport
`saveClip`	`RgnHandle`	Used internally to save the clipping region

Member Functions

Creation and destruction

CPaneMDEF

```
CPaneMDEF (short MDEFid, CPane *aPane,
   CTearOffMenu *aTearOffMenu);
```

Constructor. Constructs a CMenuDefProc object with the `MDEFid`. The data member `itsPane` is set to `aPane`, which is the pane that you want to display as a menu. `aTearOffMenu` is a tear-off menu object. You can pass `NULL` if this menu is not a tear-off menu. See Chapter 117, "CTearOffMenu."

CPaneMDEF

```
CPaneMDEF ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IPaneMDEF` for backward compatibility.

IPaneMDEF

```
void IPaneMDEF (short MDEFid, CPane *aPane,
   CTearOffMenu *aTearOffMenu);
```

Initialization function provided with backward compatibility. May not be called if constructor has arguments.

DrawMenu

```
void DrawMenu (MenuHandle macMenu,
   Rect *menuRect);
```

Draws the pane as a menu. `macMenu` and `menuRect` are provided by the Macintosh Menu Manager and are in global coordinates. `DrawMenu` uses `SetupQuickDraw` to set up the QuickDraw drawing environment so that the point (0, 0) is the upper-left corner of the pane. It calls `itsPane->RestoreEnvironment`, and then a `itsPane->Draw` to display the pane. If the menu is disabled, `DrawMenu` grays out the menu. Finally, the function restores the QuickDraw environment.

Unless your derived class has unusual requirements for drawing your menu, you should not override this function.

SizeMenu

```
void SizeMenu (MenuHandle macMenu);
```

Virtual function; overrides `SizeMenu` of base class CMenuDefProc. This function is called indirectly by the Macintosh Menu Manager to determine the size of the menu. More precisely, it is called by `GenericMDEF` whenever that global function receives an `mSizeMsg` from the Menu Manager. See Chapter 69, "CMenuDefProc," for a description of how CMenuDefProc objects interact with the Menu Manager. This function stores the height and width of the pane in the `menuWidth` and `menuHeight` fields of `macMenu`. Your derived class should not override this function.

TearOffMenu

```
void TearOffMenu (Rect *menuRect,
   Point mouseLoc);
```

This function doesn't change the QuickDraw coordinates because it expects to be called from `ChooseItem`*, which has already set things up.*

Draws a gray outline representing the menu when you tear it off the menu bar. The size of the gray outline is the `menuRect` plus the margins specified in `itsTearOffMenu`. You can drag the outline as long as the mouse is still down, and the mouse is in the tear-off region. If the mouse is in the tear-off region when the button is released, `TearOffMenu` calls the `TornOff` function of `itsTearOffMenu` to do the "tearing."

Unless your derived class needs to do something unusual while being torn off, you should not override this function.

Your derived class should call `TearOffMenu` in its `ChooseItem` function when the `hitPt` is not in the `menuRect`. You should also check to make sure that there is a tear-off menu (`itsTearOffMenu` is not `NULL`) and that the menu is not disabled.

For a good example of a `ChooseItem` *function, see Chapter 97, "CSelector."*

Here's what your `ChooseItem` might look like:

```
void CMyMDEF::ChooseItem(
   MenuHandle macMenu, Rect *menuRect,
   Point hitPt, short *whichItem)
{
   if (PtInRect(hitPt, menuRect) {
      SetupQuickDraw(menuRect);
      /* choose an item */
      RestoreQuickDraw();
   } else {
      *whichItem = NOTHING;
      if (itsTearOffMenu != NULL &&
            MenuEnabled(macMenu))
         TearOffMenu(menuRect, hitPt);
   }
}
```

PtInTearRgn

```
Boolean PtInTearRgn (Point hitPt,
   Rect *menuRect);
```

Returns `TRUE` if `hitPt` (given in global coordinates) is in the tear-off region. The tear-off region is anywhere except the menu bar and within `TEARMARGIN` pixels of the menu.

SetupQuickDraw

```
void SetupQuickDraw (Rect *menuRect);
```

Sets up the QuickDraw environment so that the `'MDEF'`s pane draws correctly. Because panes associated with `'MDEF'`s aren't really part of the visual hierarchy, you have to set up the QuickDraw environment explicitly. This function sets the port to the desktop's port, the origin to the upper-left of the pane, and the clipping region to `menuRect`.

RestoreQuickDraw

```
void RestoreQuickDraw ();
```

Restores the QuickDraw environment to its state before the preceding `SetupQuickDraw` call.

CPanorama 74

Introduction

CPanorama is a class for implementing displays that may be larger than the frame of a pane. The frame is a viewport through which a portion of the panorama is visible.

Heritage

Base Class	CPane
Derived Classes	CPicture
	CStarterPane
	CTable
	CBitMapPane
	CSelector
	CAbstractText

Using CPanorama

Use a derived class of CPanorama whenever you want to display something that is larger than the pane you want to view it through. For instance, some pictures are much bigger than a standard Macintosh screen. For an example of a panorama, see Chapter 49, “CEditText.” You can also use a CPanorama class object to hold subpanes that are to be scrolled together, as in a scrolling dialog. Every scroll pane contains a panorama. (See Chapter 96, “CScrollPane,” for a description.)

Drawing in a panorama is almost the same as drawing in a pane. The main difference is that a panorama can have its own bounds coordinate system. Each panorama bounds unit can map to one or more pixels. For example, in the CEditText class, the horizontal unit is set to the number of pixels in the widest character, and the vertical unit is set to the number of pixels of the height of a line. Scroll panes use this information to set the scroll bars accurately.

Note

Panoramas do not support fractional scales or non-linear scales for coordinate systems.

The size of the panorama image is kept in the data member `bounds`. In most cases, the upper-left corner of `bounds` is the point (0, 0). Think of the relationship between the panorama and the frame of the pane as a stationary image (the panorama) with a frame moving over it. The `position` data member contains the location of the upper-left corner of the frame in bounds coordinates. As a panorama is scrolled, the `position` changes and the frame and origin are offset accordingly.

For example, suppose each vertical scrolling unit is 16 pixels, the `bounds` are (0, 0, 9, 20), initially the `position` is (0, 0), the `frame` is (0, 0, 144, 200), and `vOrigin` is -300. After scrolling up one unit, the `position` is (1, 0), the `frame` is (16, 0, 160, 200), and `vOrigin` is -284. The `bounds` are unchanged. Drawing, which is done in frame coordinates, is offset 16 pixels upward due to the change in `vOrigin`.

Data Members

CPanorama defines these data members:

Data member	Type	Description
`bounds`	`LongRect`	Bounds defining panorama coordinates
`position`	`LongPoint`	Location of frame in panorama in bounds coordinates
`hScale`	`short`	Pixels per horizontal unit
`vScale`	`short`	Pixels per vertical unit
`savePosition`	`LongPoint`	Save for later restoration
`itsScrollPane`	`CScrollPane*`	Scroll pane that a panorama belongs to, if any

Member Functions

Creation and destruction

CPanorama

```
CPanorama (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth = 0, short aHeight =0,
   short aHEncl = 0, short aVEncl = 0,
   SizingOption aHSizing = sizELASTIC,
   SizingOption aVSizing = sizELASTIC);
```

Constructor. The arguments to this routine are identical to those for CPane. This function initializes the data members `hScale` and `vScale` to 1 pixel. Other arguments are described in Chapter 71, "CPane."

CPanorama

```
CPanorama ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IPanorama` for backward compatibility.

IPanorama

```
void IPanorama (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialization function for backward compatibility. May not be called if the constructor has arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally to initialize from a resource template. Each derived class of CView overrides this function to use its own resource template.

Accessing

GetExtent

```
void GetExtent (long *theHExtent,
   long *theVExtent);
```

Returns the size of each side of the panorama in panorama units in `theHExtent` and `theVExtent`.

GetFramePosition

```
void GetFramePosition (long *theHPos,
   long *theVPos);
```

Determines the offset of the upper-left of the frame from the upper-left of the bounds, in panorama coordinates. The CScrollPane class uses this function to figure out the maximum settings for the scroll bars. You should not use or override this function.

GetFrameSpan

```
void GetFrameSpan (short *theHSpan,
   short *theVSpan);
```

Returns the number of panorama units that the frame spans in `theHSpan` and `theVSpan`.

SetBounds

```
void SetBounds (LongRect *aBounds);
```

Sets the bounds of the panorama. The bounds define the size of the data displayed in the panorama and the panorama coordinates. If the panorama's enclosure is a scroll pane, this function calls its `AdjustScrollMax` function to adjust the scroll bars.

GetBounds

```
void GetBounds (LongRect *theBounds);
```

Returns the bounds of a panorama in `theBounds`.

SetPosition

```
void SetPosition (LongPt aPosition);
```

Sets the position of the frame in relation to the panorama. The point is in panorama coordinates. If the panorama's enclosure is a scroll pane, this function calls its `Calibrate` function to adjust the position of the scroll box (that is, the scroll bar's thumb).

GetPosition

```
void GetPosition (LongPt *thePosition);
```

Returns the position of the frame in relation to the panorama. In other words, this function sets `*thePosition` equal to the upper-left corner of the frame in panorama coordinates.

SetScales

```
void SetScales (short aHScale, short aVScale);
```

Sets the horizontal and vertical scales. Specify the scales in pixels per panorama unit. For instance, the CStaticText class (a derived class of CPanorama) uses the width of the widest character as its horizontal unit and the height of a line as a vertical unit. If the panorama's enclosure is a scroll pane, this function calls its `AdjustScrollMax` function to adjust the scroll bars.

GetScales

```
void GetScales (short *theHScale,
   short *theVScale);
```

Returns the scale factors of the panorama in `theHScale` and `theVScale`.

SetScrollPane

```
void SetScrollPane (CScrollPane *aScrollPane);
```

Specifies the scroll pane that controls this panorama. This function is for use only by the CPanorama class. To associate a panorama with a scroll pane, use `InstallPanorama`.

GetHomePosition

```
void GetHomePosition (LongPt *theHomePos);
```

Returns the location of the upper-left corner of the panorama in panorama coordinates.

GetPixelExtent

```
void GetPixelExtent (long *hExtent,
   long *vExtent);
```

Returns the size of each side of the panorama in pixels.

Calibrating

ResizeFrame

```
void ResizeFrame (Rect *delta);
```

Adjusts the frame of a panorama when its size changes. The `delta` rectangle specifies the amount of change for each side. Positive numbers mean down and to the right; negative numbers mean up and to the left.

Scrolling

Scroll

```
void Scroll (long hDelta, long vDelta,
   Boolean redraw);
```

Scrolls a panorama by `hDelta` units horizontally and `vDelta` units vertically. The units are given in panorama coordinates.

ScrollTo

```
void ScrollTo(LongPt *aPosition, Boolean redraw);
```

Scrolls the panorama to a specific position. The units given in `aPosition` are in panorama coordinates.

ScrollToSelection

```
void ScrollToSelection ();
```

Scrolls the panorama so the selection is visible. The default function does nothing. Your panorama-derived class must override this function.

AutoScroll

```
Boolean AutoScroll (LongPt *mouseLoc);
```

Scrolls automatically during a mouse-down. Returns `TRUE` if scrolling actually took place. Typically you would call this function in the same routine that tracks a selection.

GetSteps

```
void GetSteps (short *hStep, short *vStep);
```

Returns the number of units to scroll in a single step. If the panorama has a scroll pane, the default function calls that pane's `GetSteps` function and returns those values. If the panorama has no scroll pane, the default function returns one unit. You should override this function if you want to use different step sizes.

DoKeyDown

```
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

This function supports the Home, End, Page Up, and Page Down keys of the extended keyboard. The Home and End keys cause the `ScrollTo` function of the panorama's scroll pane to be called, in order to scroll to the beginning or end of the panorama. The Page Up and Page Down keys cause the scroll pane's `DoVertScroll` function to be called, in order to simulate hits in the page up and page down regions of the scroll bar.

Printing

Paginate

```
void Paginate (CPrinter *aPrinter,
   short pageWidth, short pageHeight);
```

Determines how many pages to print. The panorama is divided into horizontal and vertical strips of equal size. `aPrinter` is the printing object; typically, it belongs to the document that owns this panorama.

AboutToPrint

```
void AboutToPrint (short *firstPage,
   short *lastPage);
```

Prepares the panorama for printing from `firstPage` to `lastPage`. You can override this function if your derived class needs to take some action before printing.

PrintPage

```
void PrintPage(short pageNum, short pageWidth,
   short pageHeight, CPrinter *aPrinter);
```

Prints the specified page. `pageWidth` is the width of the page in pixels. `pageHeight` is the height of the page in pixels. `aPrinter` is the printer object; typically, it belongs to the document that owns this panorama.

DonePrinting

```
void DonePrinting ();
```

Printing has stopped. Override this function if you need to do some clean-up after printing.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Private

CPanoramaX

```
void CPanoramaX ();
```

Private function; performs common construction.

CPatternGrid 75

Introduction

CPatternGrid is a derived class of CGridSelector that displays patterns in a table and lets the user choose one. CPatternGrid is useful for implementing pattern palettes like those displayed by MacPaint™ and HyperCard™.

Heritage

Base Class	CGridSelector
Derived Classes	None

Using CPatternGrid

The CPatternGrid class lets you create panes that display patterns in a table. The most common use for this kind of table is a pattern palette for a painting or drawing program. You can use a CPatternGrid as a tool palette that's part of a window or as a custom tear-off menu. The Art Class demonstration program uses CPatternGrid to display its Patterns tear-off menu.

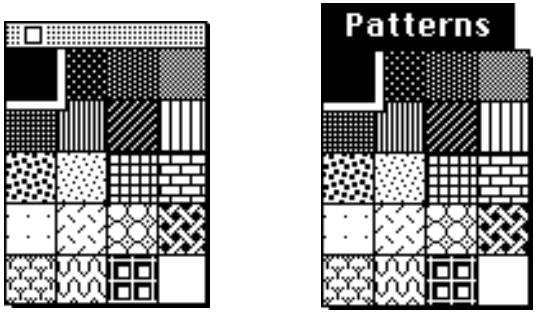


Figure 75-1 Art Class uses CPatternGrid as tear-off menu

Unlike other derived classes of CPane that let you optionally initialize an object from a resource, you must use a resource to construct a CPatternGrid. The first argument to the CPatternGrid constructor is the resource ID of a `'PtGd'` resource which contains the values that CPatternGrid passes up to CGridSelector's constructor.

Data Members

CPatternGrid defines the following data members:

Data member	Type	Description
`patListID`	`short`	The ID of the `'PAT#'` resource to use
`numPats`	`short`	Number of patterns to use
`thePatterns`	`short`	Handle to the list of `'PAT#'` resource indexes

Member Functions

Creation and destruction

CPatternGrid

```
CPatternGrid (short PtGdid,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Constructor. `PtGdid` is the resource ID of the `'PtGd'` resource that describes the location of the pane and pattern list to use for the grid. See "Using CPatternGrid" earlier in this chapter for more information on `'PtGd'` resources.

CPatternGrid

```
CPatternGrid ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IPatternGrid` for backward compatibility.

~CPatternGrid

```
~CPatternGrid ();
```

Destructor. Disposes of the index list.

IPatternGrid

```
void IPatternGrid (short PtGdid,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

Drawing

DrawItem

```
void DrawItem (short theItem, Rect *theBox);
```

Draws pattern numbered `theItem` so that it fills the box. Remember that `theItem` doesn't specify the pattern in the pattern list, but specifies the index in the index list. Pattern numbers begin with 1.

HiliteItem

```
void HiliteItem (short theItem,
   HiliteState state);
```

Highlights the specified item. If `state` is `hiliteON`, this function draws a white outline around the box. If `state` is `hiliteOFF`, it draws the box normally. When `state` is `hiliteDYNAMIC`, `HiliteItem` flashes the edges of the box. Figure 75-1 shows how CPatternGrid highlights items.

Accessing

GetPattern

```
void GetPattern (Pattern *thePattern);
```

Returns the currently selected pattern in `thePattern`. The current selection is stored in the `selection` data member of the CSelector base class.

CPictFile 76

Introduction

CPictFile is a class for reading and writing `PICT` files of the kind that MacDraw produces.

Heritage

Base Class	CDataFile
Derived Classes	None

Using CPictFile

To learn the details of the PICT file format, see Macintosh TechNote 27.

You can use this class to read and write files whose type is `PICT`. These files contain graphic images such as those made with applications like MacDraw. A `PICT` file begins with 512 bytes of header information followed by a Macintosh picture.

Since CPictFile inherits its behavior from CFile, you need to use one of CFile's specification functions to indicate which file the `Open` function will open.

Data Members

CPictFile defines a single data member:

Data member	Type	Description
`header`	`Handle`	A handle to store the `PICT` header. Used internally.

Member Functions

CPictFile

```
CPictFile ();
```

Constructor. Allocates memory for the header.

~CPictFile

```
~CPictFile ();
```

Deletes the header.

IPictFile

```
void IPictFile ();
```

Initialization function provided for backward compatibility. Does nothing.

ReadAll

```
Handle ReadAll ();
```

Reads the picture from the `PICT` file into a new handle. You can use the handle with Quickdraw routines that expect a `PicHandle`, or you can pass it to CPicture's `SetMacPicture` function. If an error occurs reading the file, this function returns `NULL`.

WriteAll

```
void WriteAll (Handle contents);
```

Writes a picture to a `PICT` file. The contents handle should be a `PicHandle`.

CPICTGrid 77

Introduction

CPICTGrid is a CGridSelector pane that draws a palette using a `'PICT'` resource.

Heritage

Base Class	CGridSelector
Derived Classes	None

Using CPICTGrid

Use CPICTGrid when you want to create a palette pane for a palette area of a standard window, a floating window, or a tear-off menu.

CPICTGrid is initialized from a `'PcGd'` resource. The first argument to the `CPatternGrid` constructor is the resource ID of a `'PtGd'` resource that contains the values `CPatternGrid` passes up to CGridSelector's constructor. The `'PtGd'` resource looks like this:

Field	Size	Description
Rows	`short`	The number of rows in the grid.
Columns	`short`	The number of columns in the grid.
Box Width	`short`	The width of each box in pixels.
Box Height	`short`	The height of each box in pixels.
Horizontal Sizing	`short`	Horizontal sizing option. Usually `sizFIXEDLEFT(0)`.
Vertical Sizing	`short`	Vertical sizing option. Usually `sizFIXEDTOP(2)`.

Table 77-1 Contents of the 'PtGd' resource

Field	Size	Description
Horizontal Location	`short`	Horizontal location of grid in its enclosure.
Vertical Location	`short`	Vertical location of grid in its enclosure.
Command Base	`short`	The command base for turning selections into command numbers.
Pattern List ID	`short`	The resource ID of the pattern list resource `'PAT#'` to use.

Table 77-1 Contents of the 'PtGd' resource *(Continued)*

The command sent when a grid square is selected is formatted like a menu command; the `commandBase` is in the high word, the item number is in the low word, and the entire long word is negated. For a tear-off menu, the command base should be set the same as the menu ID. `PICTid` is the resource ID of the `'PICT'` resource. The `'PICT'` is detached after loading, making a private copy.

Data Members

Data member	Type	Description
`macPict`	`PicHandle`	Handle to picture
`resID`	`short`	Resource ID of the `'PICT'`
`doubleClickCmd`	`long`	Command sent by double-click

Member Functions

Creation and destruction

CPICTGrid

```
CPICTGrid (short PcGdid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. Initializes the PICT grid from a `'PcGd'` resource, as described above.

CPICTGrid

```
CPICTGrid ();
```

Default constructor. Used only with object I/O.

~CPICTGrid

```
~CPICTGrid ();
```

Destructor. Removes the picture from memory.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the grid. The picture is drawn unscaled, with upper-left corner at (0, 0) in frame coordinates.

HiliteItem

```
void HiliteItem (short theItem,
   HiliteState state);
```

Highlights the selected item.

Accessing

ChangeSelection

```
void ChangeSelection (short aSelection);
```

Changes the selection from the current selection to `aSelection`. If `aSelection` is not the same as the current selection, this function turns off the highlighting of the current selection and turns on the highlighting of the new selection. Your derived class should not need to override this function.

DoDoubleClick

```
void DoDoubleClick ();
```

Responds to a double-click. The first click sets the selection, so the double-click pertains to that item. If your selector-derived class responds to double-clicks, you should override this function.

SetDoubleClickCmd

```
void SetDoubleClickCmd (long theCommand);
```

Sets the command to be sent on double-click.

SetPicture

```
void SetPicture (short PICTid);
```

Sets or resets the picture. If there is already a picture, it is destroyed. This function has no effect on the dimensions of the grid.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the grid to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the grid from the stream.

CPicture 78

Introduction

CPicture lets you display Macintosh pictures.

Heritage

Base Class	CPanorama
Derived Classes	None

Using CPicture

Use CPicture to display a Macintosh picture. You can display the picture in a scroll frame or scale it to fit its frame. The picture cannot be edited. The picture is a standard Macintosh picture and is stored in a data member. You can look at the source code for this class to learn how to create your own CPanorama-derived classes.

Data Members

CPicture defines these data members.

Data member	Type	Description
`macPicture`	`PicHandle`	Handle to the QuickDraw picture
`scaled`	`Boolean`	`TRUE` if picture is scaled to fit in frame
`isResPicture`	`Boolean`	`TRUE` if picture is read from a resource
`ownsPicture`	`Boolean`	`TRUE` if object should release memory for the picture when the object is destroyed

Member Functions

Creation and destruction

CPicture

```
CPicture ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IPicture` for backward compatibility.

CPicture

```
CPicture (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth = 0, short aHeight = 0,
   short aHEncl = 0, short aVEncl = 0,
   SizingOption aHSizing = sizElastic,
   SizingOption aVSizing = sizElastic);
```

Constructor. The arguments are identical to those of CPanorama. To completely initialize a CPicture object, you must call `UsePICT` or `SetMacPicture`.

Note

The arguments are described in Chapter 74, "CPanorama."

~CPane

```
~CPane ();
```

Destructor. If the `CPicture` object owns the picture (`ownsPicture` is `TRUE`), the memory for the picture is released. If the picture comes from a resource (`isResPicture` is `TRUE`), this function uses `HPurge` to purge the `'PICT'` resource. If the picture is not from a resource, this function uses `KillPicture` to dispose of the picture.

IPicture

```
void IPicture (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialization function compatible with previous release. Must not be called if constructor has arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally to initialize from a resource template. Each derived class of CView overrides this function to use its own resource template.

Dispose

```
void Dispose ();
```

Disposes of this object. If the object owns the picture (`ownsPicture` is `TRUE`) the memory for the picture is released. If the picture comes from a resource (`isResPicture` is `TRUE`), this function uses `HPurge` to purge the `'PICT'` resource. If the picture is not from a resource, this function uses `KillPicture` to dispose of the picture.

Appearance

Draw

```
void Draw (Rect *area);
```

Draws the picture. This function ignores the `area` parameter.

Accessing

SetMacPicture

```
void SetMacPicture (PicHandle aMacPicture);
```

Uses `aMacPicture` as the Macintosh picture for this object. If the picture comes from a resource and the resource can be purged, this function sets `ownsPicture` to `FALSE`. If the picture does not come from a resource, the picture is set to be unpurgeable, and `ownsPicture` is set to `TRUE`.

UsePICT

```
void UsePICT (short PICTid);
```

Uses the `PICT` resource with ID `PICTid` as the Macintosh picture for this object. This function gets the resource and then calls `SetMacPicture`.

GetMacPicture
```
PicHandle GetMacPicture ();
```
Returns a handle to the Macintosh picture.

Calibration

SetScaled
```
void SetScaled (Boolean aScaled);
```
If `aScaled` is `TRUE`, the picture will be scaled to fit its frame.

GetScaled
```
Boolean GetScaled ();
```
Returns `TRUE` if the picture is scaled.

ResizeFrame
```
void ResizeFrame (Rect *delta);
```
Resizes the picture's frame by the amount specified.

FrameToBounds
```
void FrameToBounds ();
```
Makes the frame of the picture the same size as the bounds.

Object I/O

PutTo
```
void PutTo (CStream& aStream);
```
Writes to the stream.

GetFrom
```
void GetFrom (CStream& aStream);
```
Reads from the stream.

Member Functions: Private

CPictureX
```
void CPictureX ();
```
Performs common initialization.

CPictureButton 79

Introduction

CPictureButton is a multi-state button drawn from pictures.

You can specify the highlighting style as either multi-state or inverted, and the drawing style of the on state as either multi-state or bordered. Multi-state buttons can be constructed by specifying up to four `'PICT'` resource IDs, one for each of the button states: off, off-highlighted, on, and on-highlighted. For example,

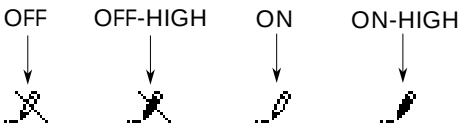


Figure 79-1 Button states

CPictureButtons can participate in button groups. All buttons with the same non-zero group ID participate in the same button group. A picture button can behave like a push button, check box, or radio button. Radio buttons turn off all other buttons in the group when they are selected; check boxes turn off only radio buttons in the group, but not each other.

Heritage

Base Class	`CSwissArmyButton`
Derived Classes	None

Using CPictureButton

`'PICT'` resources for a CPictureButton can be created with any graphics editor. The resources should always be marked as purgeable. A `'PICT'` resource in use by a picture button can be

used at the same time by other picture buttons, and by CPicture panes provided the latter do not dispose of the picture. CPictureButton leaves its 'PICT's purgeable and does a LoadResource before drawing.

Data members

Data member	Type	Description
itsPictID[4]	short	PICT resource IDs.
itsPictH[4]	PicHandle	Handle to 'PICT' resources.
scaled	Boolean	TRUE if pictures will be scaled to fit the button; FALSE otherwise

Member Functions

Creation and destruction

CPictureButton

```
CPictureButton (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aButtonKind, short aHiliteStyle,
   short anOnStyle, Boolean aNeedsUpdate,
   short offID, short offHiliteID,
   short onID, short onHiliteID);
```

Constructor. The anEnclosure, aSupervisor, aHEncl, aVEncl, aHSizing, and aVSizing arguments have their standard CPane meaning. Specify aButtonKind as kSAPushButton, kSACheckBox, or kSARadioButton, aHighliteStyle as kSAStateHilight or kSADimHilight, and anOnStyle as kSAStateOn or kSABorderOn. Set aNeedsUpdate to TRUE if the button is overlapped by another graphic or the pictures do not redraw its entire area. The four resource IDs—offID, offHiliteID, onID, and onHiliteID—are used to obtain one to four 'PICT' resources, which correspond to the off, off-highlighted, on, and on-highlighted states of the button. The on- and off-highlighted states are displayed when the button is on or off, respectively, and the user has the mouse down in the button. The frame dimensions are set to those of the off-state picture.

CPictureButton
```
CPictureButton ();
```
Default constructor. Implicitly called when an object is created by `new_by_name`.

~CPictureButton
```
~CPictureButton();
```
Destructor.

State

UsePICT
```
void UsePICT (short pictID, short state);
```
Use the `pictID` `'PICT'` resource for the specified state of the button.

GetPICTID
```
short GetPICTID (short state);
```
Returns the resource ID of the `'PICT'` resource associated with this state.

SetScaled
```
void SetScaled (Boolean scalePictures);
```
Sets the data member scaled equal to `scalePictures`.

Drawing

Draw
```
void Draw (Rect *area);
```
Draws the appropriate picture. The pictures are drawn unscaled, with the upper-left corner at (0, 0) in frame coordinates.

Object I/O

PutTo
```
void PutTo (CStream& aStream);
```
Writes the button to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the button from the stream.

Member Functions: Protected

DrawButton

```
DrawButton (Boolean fHilite);
```

Overrides `DrawButton` to erase the picture before drawing during mouse tracking.

IPictureButtonX

```
IPictureButtonX ();
```

Completes initialization by loading the `PICT` resources.

AdjustFrameSize

```
AdjustFrameSize ();
```

Adjusts the frame size to the size of the picture associated with the off state.

Member Function: Private

CPictureButtonX

```
void CPictureButtonX();
```

Performs common initialization.

CPNTGFile 80

Introduction

CPNTGFile is a class for reading and writing "paint" files like those MacPaint produces.

Heritage

Base Class	CDataFile
Derived Classes	None

Using CPNTGFile

To learn the details of the PNTG file format, see Macintosh TechNote 86.

You can use this class to read and write files whose type is `PNTG`. These files contain paint or bitmapped graphic images made with applications such as MacPaint. A `PNTG` file begins with 512 bytes of header information followed by a packed bitmap of the image.

Since CPNTGFile inherits its behavior from CFile, you need to use one of CFile's specification functions to indicate which file the `Open` function will open.

The reading and writing methods of this class use objects of class CBitMap. That class gives you an object-oriented way to work with Macintosh bitmaps.

Data Members

CPNTGFile defines this data member:

Data member	Type	Description
`header`	`Handle`	A handle to store the `PNTG` header. Used internally.

Member Functions

CPNTGFile

```
CPNTGFile ();
```

Constructor. Allocates memory for the header.

~CPNTGFile

```
~CPNTGFile ();
```

Destructor. Disposes of the header.

IPNTGFile

```
void IPNTGFile ();
```

Initialization function included for backward compatibility.

ReadNewBitMap

```
CBitMap *ReadNewBitMap (Boolean makePort);
```

Creates a new object of class CBitMap and reads the contents of this `PNTG` file into it. The `makePort` parameter is passed to CBitMap's constructor. If it is true, CBitMap creates a Quickdraw grafport. If an error occurs while reading the image, this function returns `NULL`.

Note that this function creates a new bitmap every time you call it. The image is not stored in a data member.

WriteBitMap

```
void WriteBitMap (CBitMap *theBitMap);
```

Writes an object of class CBitMap to a `PNTG` file.

CPolyButton 81

Introduction

CPolyButton is a polygon object. It can be drawn filled or unfilled, with various pen sizes and styles, as specified in the associated CColorTextEnvirons object.

A CPolyButton object can also function as a push button. See Chapter 111, "CSwissArmyButton," for more information about button styles and behavior.

Heritage

Base Class	CShapeButton
Derived Classes	None

Using CPolyButton

Initialize CPolyButton with a `PolyHandle` created by QuickDraw. The frame size is adjusted to fit the polygon. The polygon is always drawn relative to (0, 0) in frame coordinates.

Data Members

Data member	Type	Description
`poly`	`PolyHandle`	Handle to QuickDraw polygon.

Member Functions

Creation and destruction

CPolyButton

```
CPolyButton ();
```

Default constructor. Implicitly called when object is created by `new_by_name`.

CPolyButton

```
CPolyButton (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   SizingOption aHSizing,
   SizingOption aVSizing, short aButtonKind,
   short aHilightStyle, short anOnStyle,
   Boolean aNeedsUpdate, PolyHandle aPolygon);
```

Constructor. The `anEnclosure`, `aSupervisor`, `aHEncl`, `aVEncl`, `aHSizing` and `aVSizing` arguments have their standard CPane meaning (but no default values). Set `aNeedsUpdate` to `TRUE` if the polygon is overlapped by another graphic. The frame dimensions are set to those of the polygon.

~CPolyButton

```
~CPolyButton ();
```

Destructor. Deletes the `PolyHandle`.

Change Size

ChangeSize

```
void ChangeSize(Rect *delta, Boolean redraw);
```

Adjusts the frame size to that of the polygon. Call this after doing anything—for example, changing the pen size—that might affect the frame size.

SetPen

```
void SetPen(Point aPen);
```

Changes the pen size and implicitly the frame size.

Drawing

FrameShape

```
void FrameShape (Rect *theFrame);
```

Frames the shape (draws its outline).

EraseShape

```
void EraseShape (Rect *theFrame);
```

Erases the shape.

InvertShape

```
void InvertShape (Rect *theFrame);
```

Inverts the shape.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the button to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the button from the stream.

81 CPolyButton

CPopupMenu 82

Introduction

CPopupMenu manages a menu and pops it up on request. It registers its menu with the `Bartender`.

Heritage

Base Class	CBureaucrat
Derived Classes	None

Using CPopupMenu

CPopupMenu, a derived class of CBureaucrat, is the basic pop-up menu class. It manages a Macintosh `MenuHandle` and pops up the menu when its `PopupSelect` function is called. Usually you won't use CPopupMenu. Instead, you'll create a CPopupPane, CStdPopupPane, or CArrowPopupPane, all of which create a CPopupMenu and communicate with it.

To use a CPopupMenu, you must create a `'MENU'` resource in your resource file. In your program, call the `CPopupMenu` constructor with a menu ID, a supervisor, and the `autoSelect`, `multiSelect` flags.

CPopupMenu registers its menu with the global `CBartender` object, `gBartender`. Only one copy of the menu exists, even if several pop-ups are using the same menu.

The most important function in CPopupMenu is `PopupSelect`, which actually pops up the menu and issues the command corresponding to the chosen menu item. See the description of PopupSelect later in this chapter for more information.

The easiest way to use a pop-up menu is to use a radio-style menu, which is like using a group of radio buttons. In a radio-style pop-up

menu, you can check only one item at a time, which you can use as the menu's value. To get the checked item, call `GetHighlightSelection`. The pop-up menu automatically checks the item when the user clicks the menu. To make a radio-style pop-up menu, you must set `radioStyle` to `TRUE` and set `autoSelect` and `multiSelect` to `FALSE`.

If you don't use a radio-style pop-up menu, CPopupMenu stores whatever is selected but in the menu rather than in a CPopupMenu data member. If you use the same menu in more than one place (for example, a font menu that occurs in several dialogs), you need to store the state for each use of the menu. Reset the menu in an `UpdateMenus` member function in one of the menu's supervisors in the chain of command.

If you don't use a radio-style pop-up menu, you can use `autoSelect` and `multiSelect` to set how the menu works. `autoSelect` determines whether or not menu items are checked and unchecked automatically when selected. If `autoSelect` is `FALSE`, it is your responsibility to maintain the check status of menu items. CPopupMenu calls the bartender's `UpdateAllMenus` before it pops up the menu, so any bureaucrat in the chain of command may check and enable menu items.

`multiSelect` determines whether or not more than one menu item may be checked at a time. If `multiSelect` is `FALSE`, then one and only one menu item must be checked.

The other functions in CPopupMenu provide services for accessing and manipulating menus and menu items. You can also modify the pop-up menu by calling CBartender functions that manipulate menus and commands.

Data Members

CPopupMenu defines these protected data members:

Data member	Type	Description
`macMenu`	`MenuHandle`	Handle to the menu
`menuID`	`short`	ID of the menu
`lastCmd`	`long`	Last command selected from this menu

Data member	Type	Description
autoSelect	Boolean	TRUE if the pop-up menu automatically checks and unchecks items when you select them
multiSelect	Boolean	TRUE if the pop-up menu allows you to check more than one menu item
radioStyle	Boolean	TRUE if the menu is used like a radio button cluster
firstSelection	short	The first checked menu item number
itsMark	short	The mark character used to check items in the menu

Member Functions

Creation and destruction

CPopupMenu

```
CPopupMenu (short aMenuID,
   CBureaucrat *aSupervisor,
   Boolean fAutoSelect = FALSE,
   Boolean fMultiSelect = TRUE,
   Boolean fRadioStyle);
```

Constructor. `MenuID` specifies the resource ID of the pop-up menu. If `fAutoSelect` is `TRUE`, menu items are checked and unchecked automatically as you select them. Otherwise, your program must check and uncheck items. If `fMultiSelect` is `TRUE`, the menu lets you check more than one menu item. If `fRadioStyle` is `TRUE`, the menu keeps track of a single checked item internally, without interacting with other uses of the same `'MENU'` resource. See "Using CPopupMenu" earlier in this chapter for a more complete discussion.

CPopupMenu

```
CPopupMenu ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IPopupMenu` for backward compatibility.

IPopupMenu

```
void IPopupMenu (short aMenuID,
   CBureaucrat *aSupervisor,
   Boolean fAutoSelect, Boolean fMultiSelect);
```

Initialization function for backward compatibility. May not be called if constructor call has arguments.

~CPopupMenu

```
~CPopupMenu ();
```

Destructor. Removes the menu from CBartender's list by calling `RemoveMenu`.

Accessing

GetMacMenu

```
MenuHandle GetMacMenu ();
```

Returns the menu handle for the pop-up menu.

GetTitle

```
void GetTitle (StringPtr aTitle);
```

Returns the menu's title in `aTitle`.

GetWidth

```
short GetWidth ();
```

Returns the width of the menu when it's popped up.

GetNumItems

```
short GetNumItems ();
```

Returns the number of items in the menu.

GetItemString

```
void GetItemString (short item,
   StringPtr itemStr);
```

Returns the text of the item in `itemStr`.

GetItemCommand

```
long GetItemCommand (short item);
```

Returns the command number of the `item`.

GetCheckedItem

```
short GetCheckedItem ();
```

If there are any checked items, returns the item number of the first checked item. Otherwise, returns `0`.

GetCurrItemString

```
void GetCurrItemString (StringPtr string);
```

If there are any checked items, returns the name of the first checked item. Otherwise, returns the empty string (`""`).

GetHighlightSelection

```
short GetHighlightSelection ();
```

Returns the menu item that is selected when the user clicks the menu.

SetAutoSelect

```
Boolean SetAutoSelect (Boolean fAutoSelect);
```

Sets the `autoSelect` data member and returns its previous value. If `fAutoSelect` is `TRUE`, the pop-up menu automatically checks and unchecks items as you select them.

SetMultiSelect

```
Boolean SetMultiSelect (Boolean fMultiSelect);
```

Sets the `multiSelect` data member and return its previous value. If `fMultiSelect` is `TRUE`, the user can check more than one menu item.

SetRadioStyle

```
Boolean SetRadioStyle (Boolean fRadioStyle);
```

Sets the `radioStyle` data member and returns its previous value. If `radioStyle` is `TRUE`, only one item at a time can be checked. The pop-up menu itself checks the item, so you don't have to update it in any `UpdateMenus` function.

SetMarkChar

```
void SetMarkChar (short markChar);
```

Sets the `itsMark` data member. The mark character is used to indicate checked items in the menu. Only the low-order 8 bits are used. Setting the mark character is important when the menu is

displayed in any font other than Chicago, as most other fonts do not contain the checkmark character.

GetMarkChar

```
short GetMarkChar ();
```

Returns the mark character used to check items in the menu. Only the low-order 8 bits are significant.

Item insertion

AppendResNames

```
void AppendResNames (ResType aResType);
```

Appends the names of all the available resources of the type `aResType`. This is useful for creating a menu of all available fonts, for example.

Item manipulation

CheckCmdKey

```
void CheckCmdKey (char theChar, Byte keyCode,
   EventRecord *macEvent, long *theCmd,
   short *theItem);
```

If the key down event is a Command key for this pop-up menu, this function sets `theCmd` to the command number and `theItem` to the item number. Otherwise, it sets `theCmd` to `cmdNull` and `theItem` to `0`.

ItemIsChecked

```
Boolean ItemIsChecked (short item);
```

Returns `TRUE` if the item has a checkmark.

SelectItem

```
void SelectItem (short itemNum,
   tPMSelectAction actionType);
```

Places or removes a check mark from item `itemNum` in the pop-up menu. If the menu allows you to select more than one item

(multiSelect is TRUE), you can check or uncheck the item itemNum by setting actionType to one of these values:

If actionType is…	The item is…
pmForceOn	Always checked
pmForceOff	Always unchecked
pmToggle	Checked if it's unchecked, unchecked if it's checked

If the menu allows you to select one and only one item (multiSelect is FALSE), this function unchecks the currently checked item and checks the itemNum. item. actionType must be either pmForceOn or pmToggle. This function does nothing if actionType is pmForceOff or if itemNum is the currently checked item.

SelectItemName

```
void SelectItemName (StringPtr name,
   tPMSelectAction actionType);
```

Places or removes a checkmark from the item in the pop-up menu titled name. If the menu allows you to select more than one item (multiSelect is TRUE), you can check or uncheck the item name by setting actionType to one of these values:

If actionType is…	The item is…
pmForceOn	Always checked
pmForceOff	Always unchecked
pmToggle	Checked if it's unchecked, unchecked if it's checked

If the menu allows you to select one and only one item (multiSelect is FALSE), this function unchecks the currently checked item and checks the name item. actionType must be either pmForceOn or pmToggle. This function does nothing if actionType is pmForceOff or if the item named name is the currently checked item.

Mouse

PopupSelect

```
long PopupSelect (Point where, long aCmd);
```

Called when the user clicks the menu. It pops up the menu and lets the user select an item. `where` specifies in global coordinates where the menu will pop up. `aCmd` specifies the command that should be selected when the menu pops up. To have this function choose a suitable default selection, set `aCmd` to `NULL`.

If the user selects nothing, this function returns `cmdNull`. If the user selects an item, it calls the `DoCommand` function of the pop-up menu pane (which passes it to its supervisor), then calls the `BroadcastChange` function of its supervisor and dependents with `popupMenuNewSelection` as the reason, and returns the item's command number.

For more information on dependent objects, see Chapter 33, "CCollaborator."

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

CheckMenuItem

```
void CheckMenuItem (short item,
   Boolean fCheckIt);
```

Checks the item if `fCheckIt` is `TRUE` and unchecks it if `fCheckIt` is `FALSE`. Use `SelectItem` to check and uncheck items. You shouldn't need to use this function.

PreparePopup

```
void PreparePopup ();
```

Sets up a menu before popping it up. This function turns off dimming for all menu items and, if `autoSelect` is `TRUE`, it checks the selected items. If `autoSelect` is `FALSE`, your program must check items with calls to `UpdateMenus`. Override this function if you need to perform any additional preparation.

IPopupMenuX

```
void IPopupMenuX ();
```

Performs common initialization.

CPopupMenuX

```
void CPopupMenuX ();
```

Performs common construction.

CPopupPane 83

Introduction

CPopupPane is an abstract class for pop-up menu panes. A pop-up menu pane is a pane that owns a CPopupMenu and pops up the menu when clicked.

Heritage

Base Class	CPane
Derived Classes	CStdPopupPane
	CArrowPopupPane

Using CPopupPane

CPopupPane is an abstract class that pops up a menu when the user clicks on the pane. It leaves the pane blank, drawing the menu only when CPopupPane is popped up. The THINK Class Library includes two derived classes that draw the menu when CPopupPane is *not* popped up: CStdPopupPane or CArrowPopupPane. Usually you'll use one of these derived classes.

CStdPopupPane displays the menu's title and the currently selected menu item, as shown in Figure 83-1.

Figure 83-1 A CStdPopupPane before and after clicking on it

CArrowPopupPane displays only an arrow (see Figure 83-1).

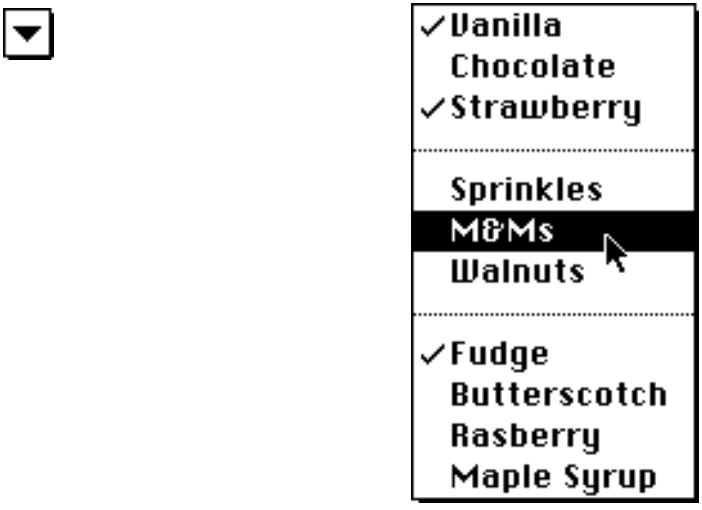


Figure 83-2 A CArrowPopupPane before and after clicking on it

The THINK Class Library handles pop-up menus such as those in the menu bar. CPopupMenu registers itself with the bartender. It provides functions for associating command numbers with menu items and for defining the pop-up menu from a `'MENU'` resource.

When you click in a pop-up menu pane, the `DoClick` function calls the pop-up menu's `PopupSelect` function. This function, in turn, displays the pop-up menu, lets the user choose a menu item, and returns the command number associated with the selected item. If the command number isn't `cmdNull`, `PopupSelect` also calls the `DoCommand` function of the pop-up menu pane, which makes the same call to its supervisor.

If you derive your own class from CPopupPane, you should override `NewMenuSelection` and `Draw`. `NewMenuSelection` is called when the user chooses a new item from the menu. `Draw` draws the menu when it's not popped up.

Data Members

CPopupPane defines one protected data member:

Data member	Type	Description
`itsMenu`	`CPopupMenu*`	The menu to pop up when this pane is clicked

Member Functions

Creation and destruction

CPopupPane

```
void CPopupPane (short menuID,
   Boolean fAutoSelect, Boolean fMultiSelect,
   Boolean fRadioStyle,
   CView *anEnclosure, CBureaucrat *aSupervisor,
   short aWidth = 0, short aHeight = 0,
   short aHEncl = 0, short aVEncl = 0,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing);
```

Constructor. Calls `MakePopupMenu`, described below, with the first four parameters to create its CPopupMenu. For more information on CPopupMenu, see Chapter 82, “CPopupMenu.”

Note

The rest of the arguments are described in Chapter 71, “CPane.”

CPopupPane

```
CPopupPane ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IPopupPane` for backward compatibility.

~CPopupPane

```
~CPopupPane ();
```

Destructor. Deletes `itsMenu`.

IPopupPane

```
void IPopupPane (short menuID,
   Boolean fAutoSelect, Boolean fMultiSelect,
   CView *anEnclosure, CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   Boolean fRadioStyle = TRUE);
```

Initialization function for backward compatibility. May not be called if constructor call has arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally to initialize a CPopupPane from a resource template. Each class derived from CView overrides this function to use its own resource template.

Accessing

GetMenu

```
CPopupMenu *GetMenu ();
```

Returns a pointer to the CPopupMenu that this pane owns.

To get the menu handle associated with that pop-up menu, call the CPopupMenu's `GetMacMenu` function.

GetHelpBalloonState

```
short GetHelpBalloonState ();
```

Returns the balloon help state:

kHMEnabledItem (0) -	Enable (or, for buttons, off)
kHMDisabledItem (1) -	Disabled (dimmed)
kHMCheckedItem (2) -	Enabled and checked (or, for buttons, on)
kHMMarkedItem (3) -	Enabled and marked (for any other purpose)

Controls and edit panes typically implement only the first two of three of the above states.

A one-to-one correspondence is not necessarily found between balloon help states and THINK Class Library variable states. The dimmed state is the most complex. Dimmed means the view would normally accept user interaction but currently does not. Note that a view must normally want clicks or allow editing to be considered dimmed when it does not want clicks or allow editing. Classes for which dimmed or marked states are meaningful, or that want to use the marked state, must override this function.

SetMenu

```
void SetMenu (struct CPopupMenu *aMenu);
```

Sets `itsMenu` to `aMenu`. Deletes any existing CPopupMenu.

Mouse

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handles a click by popping up `itsMenu`. Uses the `CalcPopupPoint` function to determine where to place the menu, and the `CalcPopupCmd` function to determine the initially selected menu command.

Menu selection

NewMenuSelection

```
void NewMenuSelection (short itemSelected);
```

Called when the user selects a new command in the menu. The default function does nothing. Override this function if you need to act on this information—for instance, by updating text that is dependent upon the selected menu item.

Appearance

Activate

```
void Activate ();
```

Activates this pane. Calls `Refresh` to redraw the pane as visually enabled on the next update event.

Deactivate

```
void Deactivate ();
```

Deactivates this pane. Calls `DrawAll` to immediately redraw the pane as visually disabled, without waiting for an update event.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the pane in which the user clicks to get the pop-up menu. When the pane is enabled, this function does not draw any representation of the menu or its selection. When the pane is disabled, it grays out what's there. Your derived class must override this function. Your function can draw the pane and then call `CPopup::Draw` to gray the pane out if it's disabled.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

MakePopupMenu

```
void MakePopupMenu (short menuID,
   Boolean fAutoSelect, Boolean fMultiSelect,
   Boolean fRadioStyle);
```

Creates the CPopupMenu that this pane will own and sets `itsMenu` equal to it. The supervisor of the CPopupMenu is the CPopupPane. `menuID` is the resource ID of the `'MENU'` resource. `fAutoSelect` controls whether the menu automatically checks or unchecks menu items when selected. `fMultiSelect` controls whether multiple menu items may be checked. `fRadioStyle` controls whether the CPopupMenu object keeps track of which single item is checked, independent of all other uses of this menu. See "Using CPopupMenu" in Chapter 82, "CPopupMenu."

Override this function if you create a derived class of CPopupMenu.

Location

CalcPopupPoint

```
Point CalcPopupPoint (Point hitPt,
   short modifierKeys, long when);
```

Returns the upper-left point at which the pop-up menu should appear. The point is expressed in global coordinates. This function returns `hitPt` converted to global coordinates.

CalcPopupCmd

```
long CalcPopupCmd ();
```

Returns the menu command to select initially when the popup menu appears. This function returns `cmdNull`, which tells `itsMenu` to determine the appropriate menu item itself.

Provider

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void *info);
```

Called when the CPopupMenu `itsMenu` changes. If the menu selection changes, this function calls `NewMenuSelection`. It passes the `ProviderChanged` call up the chain of command; if the menu selection changes, it calls `BroadcastChange` with itself as the provider.

83 CPopupPane

CPrinter 84

Introduction

CPrinter is a class that handles standard Macintosh printing dialogs and calls the appropriate Print Manager routines.

Heritage

Base Class	None
Derived Classes	None

Using CPrinter

The printer object manages communication between a document and the Macintosh Print Manager. Every document object can have a printer object associated with it. The document's `PrintPageOfDoc` function actually does the printing.

If your document is long, you may need to paginate it (that is, divide it into pages). CPrinter lets you subdivide your document into strips, each of which is a row or column of pages. At the intersection of a

vertical strip and a horizontal strip, there is a page. Figure 84-1 shows how you might paginate graphic and text documents.

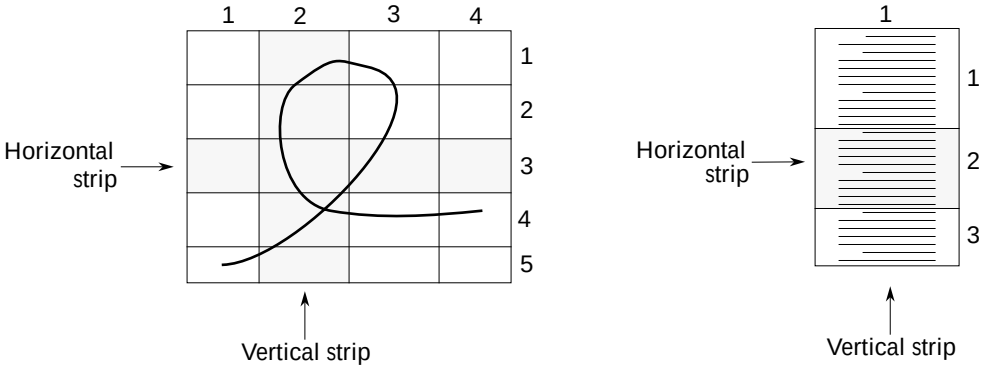


Figure 84-1 Paginating graphic and text documents

Note that strips don't have to be all the same size. Also, notice that each strip has a number, which is used to identify it when setting its size or position individually.

CPrinter lets you set the size and position of strips in a variety of ways, such as:

- To make all pages the size chosen by the user in the **Page Setup** dialog, use the `SetStrips` function described later in this chapter.
- To insert a page break at a specific place in a document, use the `SetHorizPageBreak` and `SetVertPageBreak` functions described later in this chapter.
- To make all pages the same height or width, use the `SetAllStripWidths` or `SetAllStripHeights` functions described later in this chapter.
- To make a specific strip a specific height or width, use the `SetStripWidth` or `SetStripHeight` functions described later in this chapter.

If your document has more than one horizontal and vertical strip, you can choose the order in which to print the pages: horizontally (left to right first) or vertically (top to bottom first), as shown in Figure 84-2. Just call the `CPrinter` member function

SetPrintDir with either printHoriz or printVert as the argument.

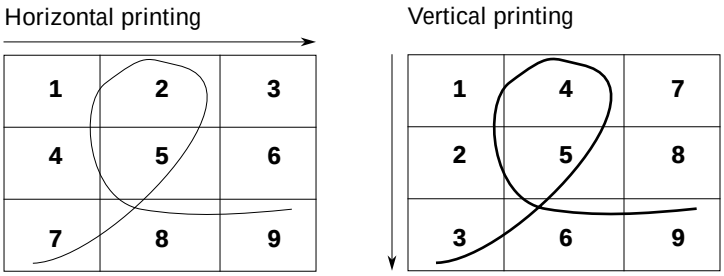


Figure 84-2 Printing horizontally and vertically

You can preserve defaults by storing a Macintosh print record with your document. You can pass a handle to this record to the CPrinter constructor when you create your document.

Ordinarily, you won't need to derive a new class from CPrinter.

Data Members

CPrinter defines the following data members:

Data member	Type	Description
itsDocument	CDocument	Document using this printer
macTPrint	THPrint	Toolbox print record
macPrintPort	TPPrPort	Port to be used for printing
printDirection	tPrintDirection	The direction of printing when there is more than one strip
printMgrOpen	Boolean	TRUE, if the Print Manager is open
printDocOpen	Boolean	TRUE, if this printer is currently printing the document
printPageOpen	Boolean	TRUE, if this printer is currently printing a page

Data member	Type	Description
savedResFile	short	The resource file that this application used before CPrinter opened the Print Manager
itsStripWidths	CRunArray*	List of strip widths
itsStripHeights	CRunArray*	List of strip heights
printChanged	Boolean	TRUE if print record changed during construction
wantstoPrint	Boolean	TRUE if user clicked OK in Print Dialog

Member Functions

Creation and destruction

CPrinter

```
CPrinter (CDocument *aDocument,
   THPrint aMacTPrint);
```

Constructor. aDocument is the document with which the printer object is associated. aMacTPrint is a Macintosh print record handle. If aMacTPrint is NULL, this function creates a new print record. It sets printChanged to TRUE if it had to update the aMacTPrint record because it was incompatible. The document's constructor initializes a printer object automatically if the printable parameter to CDocument is TRUE.

CPrinter

```
CPrinter ();
```

Default constructor. Implicitly called when an object is created by new_by_name. Can also be used in combination with IPrinter for backward compatibility.

~CPrinter

```
~CPrinter ();
```

Destructor. Disposes the Toolbox printer record and other associated memory.

Dispose

```
void Dispose ();
```

Calls `delete this`. Only available if TCL_USE_DISPOSE is defined.

IPrinter

```
Boolean IPrinter (CDocument *aDocument,
   THPrint aMacTPrint);
```

Initialization function for backward compatibility. May not be called if constructor call has arguments. This function returns `TRUE` if it had to update the `aMacTPrint` record because it was incompatible.

GetPrintChanged

```
Boolean GetPrintChanged();
```

Returns the value of `printChanged`. Allows program to test whether the constructor had to update the Toolbox print record.

Accessing

OpenPrintMgr

```
Boolean OpenPrintMgr (Boolean fCheckFailure);
```

Opens the Print Manager to get ready to print. If `fCheckFailure` is `TRUE` and this function could not open the Print Manager, it displays an alert telling the user to choose a printer with the Chooser, and then returns `FALSE`. It returns `TRUE` if there was no error or if `fCheckFailure` is `FALSE`.

ClosePrintMgr

```
void ClosePrintMgr ();
```

Closes the Print Manager. This function aborts the printing of the current page and document, if this printer is currently printing.

SetPrintDir

```
void SetPrintDir (tPrintDirection aPrintDir);
```

Sets the direction of printing in case there is more than one horizontal and vertical strip. `tPrintDirection` can be `printHoriz` or `printVert`.

HavePagination

```
Boolean HavePagination ();
```

Returns TRUE if this printer has any pagination information.

ResetPagination

```
void ResetPagination ();
```

Clears the current pagination, setting the number of horizontal and vertical strips to 0.

SetStrips

```
void SetStrips (short numHStrips,
   short numVStrips);
```

Clears the current pagination and initializes it. This function sets the number of strips to `numHStrips` and `numVStrips`, and sets the width (or height) of each strip to the width (or height) of the page size set in **Page Setup**.

SetHorizPageBreak

```
void SetHorizPageBreak (short vStripNum,
   long hPos);
```

Sets a horizontal page break. `vStripNum` is the strip that will be changed. `hPos` is the location of the page break in the frame coordinates of this printer's document.

SetVertPageBreak

```
void SetVertPageBreak (short hStripNum,
   long vPos);
```

Sets a vertical page break. `hStripNum` is the strip that will be changed. `vPos` is the location of the page break in the frame coordinates of this printer's document.

SetAllStripWidths

```
void SetAllStripWidths (short aStripWidth);
```

Sets the width of all vertical page strips to `aStripWidth`.

SetAllStripHeights

```
void SetAllStripHeights (short aStripHeight);
```

Sets the height of all horizontal page strips to `aStripHeight`.

SetStripWidth

```
void SetStripWidth (short pageNum,
   short aStripWidth)
```

Sets the width of the vertical strip `pageNum` to `aStripWidth`.

SetStripHeight

```
void SetStripHeight (short pageNum,
   short aStripHeight)
```

Sets the height of the horizontal strip `pageNum` to `aStripheight`.

GetStripCount

```
void GetStripCount (short *hStrips,
   short *vStrips);
```

Gets the number of horizontal and vertical page strips.

PageNumToStrips

```
void PageNumToStrips (short pageNum,
   short *hStrip, short *vStrip)
```

Gets the horizontal and vertical strips in which page `pageNum` falls.

GetPageStart

```
void GetPageStart (short pageNum,
   LongPt *startPos)
```

Gets the starting position of page `pageNum` in the frame coordinates of this printer's document.

GetPageArea

```
void GetPageArea (short pageNum,
   LongRect *pageArea)
```

Gets the area of page `pageNum`.

GetPrintRecord

```
THPrint GetPrintRecord ();
```

Returns the Toolbox print record. This function calls `PrValidate` to update the record. You should treat the value that this function returns as read-only.

GetPageInfo

```
void GetPageInfo (Rect *paperRect,
   Rect *pageRect, short *hRes, short *vRes);
```

Gets information about the paper size and printable area of the page. The `paperRect` and `pageRect` are specified in dots. `hRes` and `vRes` specify the number of dots per inch.

DoPageSetup

```
Boolean DoPageSetup ();
```

Responds to a **Page Setup** menu command. Displays the standard job setup dialog; returns `TRUE` if you made changes and clicked OK.

WantsToPrint

```
Boolean wantsToPrint ();
```

Returns to `wantstoPrint` data member. `wantstoPrint` is `TRUE` unless the user clicked Cancel in the **Print** dialog.

Printing

DoPrint

```
void DoPrint ();
```

Responds to a **Print** menu command. Displays the standard print job dialog. If the user clicks OK, this function calls the `PrintPageRange` function.

PrintPageRange

```
void PrintPageRange (short firstPage,
   short lastPage);
```

Prints the specified range of the document associated with this printer object. This function is usually called by `DoPrint`. Your application can bypass the print dialogs and call this function directly, but this practice is discouraged.

This function calls your document's `AboutToPrint` function to give it an opportunity to adjust the page range. Then, for each page in the page range, it calls your document's `PrintPageofDoc` function.

CProperty 85

Introduction

CProperty lets you represent Apple event object properties during the processing of an event.

Heritage

Base Class	CAppleEventObject
Derived Classes	None

Using CProperty

When your Apple event object's `AccessObject` member function is called to access a property, usually you should create a new CProperty object to represent the property and to return a token pointing to that object. For example:

```
void CApplication::AccessObject(...)
{
   switch (desiredClass)
   {
      case cProperty:
         switch (GetProperty(keyData))
         {
            case pName:
               (new CProperty(this, pName,
                  cIntlText))->
                     MakeToken(theToken);
               return;
            ....
         }
      ...
   }
   CAppleEventObject::AccessObject(...);
}
```

The `AccessObject` member function should always call its base class's `AccessObject` function for cases it does not handle.

A CProperty object can describe itself as a token or as an object specifier. It can respond to Get Data and Set Data events by calling its owner object's GetPropertyData and SetPropertyData member functions.

The object owning the property is responsible for fetching and—for modifiable properties—storing property values. For example:

```
void CApplication::DoPropertyGetDataEvent(
            DescType property,
            const AEDesc *requestedType,
            AEDesc *data)
{
   switch (property)
   {
      case pName:
         unsigned char *appName =
            LMGetCurApName();

         CreateDesc(typeChar, &appName[1],
            appName[0], data);
         CoerceDescList(requestedType, data);
         break;
      ...
      default:
         CAppleEventObject::
            DoPropertyGetDataEvent(...);
         break;
   }
}
```

The `DoPropertyGetDataEvent` member function should always call its base class's `DoPropertyGetDataEvent` function for cases it does not handle.

Data Members

CProperty has the following protected data members:

Data member	Type	Description
`itsObject`	`CAppleEventObject*`	Pointer to object owning property
`itsProperty`	`DescType`	Property ID
`itsType`	`DescType`	Default type of property

Member Functions

Creation and destruction

CProperty

```
CProperty (CAppleEventObject *object,
   DescType propertyID, DescType type);
```

Constructor. `object` is the object whose property is being represented. `propertyID` identifies the property, for example, `pName`. `type` is the property's default type, for example, `typeIntlText`.

~CProperty

```
~CProperty ();
```

Destructor.

Event handling

DoGetDataEvent

```
void DoGetDataEvent (CAppleEvent *theEvent,
   AEDesc *result);
```

Calls `itsObject`'s `DoPropertyGetDataEvent` member function.

DoSetDataEvent

```
void DoSetDataEvent (CAppleEvent *theEvent,
   AEDesc *result);
```

Calls `itsObject`'s `DoPropertySetDataEvent` member function.

Object information

GetContainer

```
CAppleEventObject *GetContainer ();
```

Returns `itsObject`.

GetClassID

```
DescType GetClassID ();
```

Returns the constant `cProperty`.

GetDefaultType

```
DescType GetDefaultType ();
```

Returns `itsType`.

Object access

AccessObject

```
void AccessObject (DescType desiredClass,
   const AEDesc *containerToken,
   DescType containerClass, DescType keyForm,
   const AEDesc *keyData, AEDesc *theToken,
   long theRefcon);
```

Fails with `errAEEventNotHandled`. If your property has elements or subproperties, you must create a derived class and override this function. (No standard property has subproperties, but the capability is available; for example, some developers treat text properties as text containers.)

MakeSelfDescriptor

```
void MakeSelfDescriptor (DescType* keyForm,
   AEDesc* keyData);
```

Creates a `formPropertyID` descriptor with `itsProperty` as data.

CPtrArray 86

Introduction

CPtrArray implements a variable-sized array of object pointers.

Heritage

Base Class	CVoidPtrArray
Derived Classes	CList

Using CPtrArray

Use a CPtrArray to maintain a list or array of object pointers.

CPtrArray is a template class, declared as:

```
template <class T>
class CPtrArray : public CVoidPtrArray
{ ... };
```

To use a CPtrArray, specify the class of the pointers to be stored in the array. For example, to declare an array of pointers to objects of class `MyClass`, you must write:

```
CPtrArray<MyClass>   *arrayOfMyClass;
```

For each CPtrArray class you use, you must expand the template class explicitly. The simplest way to do this is to follow the example of the THINK Class Library template classes. For instance, to expand the class `CPtrArray<MyClass>`, add to your project a file named `CPtrArray_MyClass.cp` that contains these statements:

```
#include "CPtrArray.h"
#include "MyClass.h"

#pragma template_access public

#pragma template CPtrArray<MyClass>

TCL_DEFINE_CLASS_M1(CPtrArray<MyClass>,
                    CVoidPtrArray);

#include "CPtrArray.tem"
```

See Chapter 87, "CPtrArrayIterator," for how to loop through the elements of a CPtrArray.

Data Members

CPtrArray has no data members.

Member Functions

In the member functions below, the class of the object pointers in the array is indicated by `T*`. When the template is expanded, `T` is replaced by the actual class you specify.

Except for the constructor, `DisposeItems`, and `DisposeAll`, all CPtrArray member functions are inline.

Creation and destruction

CPtrArray

```
CPtrArray (short blockSize = 3);
```

Constructor. Each time the array must get more memory, it allocates a block that can hold `blockSize` many pointers.

~CPtrArray

```
~CPtrArray ();
```

Destructor. Sets the `items` handle (inherited from CVoidPtrArray) to `NULL`.

Copy

```
void *Copy ();
```

Returns a copy of `this`.

DisposeItems

```
void DisposeItems ();
```

Calls operator `delete` for each object pointer in the array and removes it from the array. Does not delete the array itself. If you call this function, it is essential that there be no duplicate pointers in the array.

DisposeAll

```
void DisposeAll ();
```

Calls `DisposeItems`, then deletes the array.

Accessing items

NthItem

```
T *NthItem (long index);
```

Returns the pointer stored in location `index`. Arrays are indexed from `1` through the number of items in the array.

FirstItem

```
T *FirstItem ();
```

Returns the pointer stored in location 1.

LastItem

```
T *LastItem ();
```

Returns the pointer stored in location `numItems`.

Adding items

InsertAt

```
void InsertAt (T *ptr, long index);
```

Inserts the `ptr` at the specified `index` position, increasing the size of the array by one item and shifting pointers previously stored at `index` or higher up by one position.

Append

```
void Append (T *ptr);
```

Adds the `ptr` to the end of the array. Same as:

```
InsertAt(ptr, GetNumItems()+1)
```

Note

A common programming mistake is attempting to append an item to an array by calling:

```
InsertAt(ptr, GetNumItems());
```

Instead, this inserts the item imediately before the last item. Use the `Append` function to append correctly.

Add

```
void Add (T *ptr);
```

Same as `Append`. Provided for compatibility with previous releases.

Prepend

```
void Prepend (T *ptr);
```

Inserts the `ptr` at the start of the array. Same as:

```
InsertAt(ptr, 1)
```

InsertAfter

```
void InsertAfter (T *ptr, T *afterPtr);
```

Locates the first occurrence of `afterPtr` in the array and inserts the `ptr` in the next higher location. Same as:

```
InsertAt(ptr, FindIndex(afterPtr)+1).
```

Removing items

The basic way to remove a pointer from a CPtrArray is to call the `Remove` function inherited from CArray:

```
Remove(i);
```

This removes the pointer at position *i* from the array, shifting all pointers at higher index locations down by one position and reducing the size of the array by 1 item.

Remove

```
void Remove (T *ptr);
```

Overloads the inherited `Remove`, discussed above. Locates the first occurrence of `ptr` in the array and removes it from the array. Same as:

```
DeleteItem(FindIndex(afterPtr)).
```

Testing

Includes

```
Boolean Includes (T *theObject);
```

Returns `TRUE` if `theObject` is stored in the array, `FALSE` if it is not.

Finding

FindIndex

```
long FindIndex (T *ptr);
```

Locates the first occurrence of `ptr` in the array and returns its index.

FirstSuccess

```
T *FirstSuccess (vTestFunc testFunc);
```

Starting from item 1 and searching forward through the array, this function applies `testFunc` to each item until `testFunc` returns `TRUE` or the end of the array is reached. If `testFunc` returns `TRUE` for an item, `FirstSuccess` returns that item; otherwise, it returns `NULL`. `testFunc` must be declared as:

```
Boolean testFunc(T *ptr);
```

The same search can be performed using an iterator:

```
CPtrArrayIterator<T> iter(myArray
                          kStartAtBeginning);
T *item;

while (iter.Next(item))
   if (testFunc(item))
   {
      item = NULL;
      break;
   }
// item contains the desired pointer or NULL
```

When an iterator is used, a separate function `testFunc` is not required; the test can be written inline. The iterator has the additional advantage that the index position of the located item can be determined without further searching: a call to `iter.GetCursor()` obtains the index.

FirstSuccess1

```
T *FirstSuccess1 (vTestFunc1 testFunc,
   long param);
```

Starting from item 1 and searching forward through the array, this function applies `testFunc` to each item and the `param` until `testFunc` returns `TRUE` or the end of the array is reached. If `testFunc` returns `TRUE` for an item, `FirstSuccess1` returns that item; otherwise, it returns `NULL`. `testFunc` must be declared as:

```
Boolean testFunc(T *ptr, long param);
```

LastSuccess

```
T *LastSuccess (vTestFunc testFunc);
```

Same as `FirstSuccess`, except the function searches the array backward starting from the last item. Note that the same search can be performed using an iterator:

```
CPtrArrayIterator<T> iter(myArray
                          kStartAtEnd);
T *item;

while (iter.Prev(item))
   if (testFunc(item))
   {
      item = NULL;
      break;
   }
// item contains the desired pointer or NULL
```

When an iterator is used, a separate function `testFunc` is not required; the test can be written inline. The iterator has the additional advantage that the index position of the located item can be determined without further searching: the index position is `iter.GetCursor()+1`.

LastSuccess1

```
T *LastSuccess1 (vTestFunc1 testFunc,
   long param);
```

Same as `FirstSuccess1`, except that it searches the array backward starting from the last item.

FindItem

```
T *FindItem (vTestFunc testFunc);
```

Same as `FirstSuccess`. Provided for backward compatibility.

FindItem1

```
T *FindItem1 (vTestFunc1 testFunc,
   long param);
```

Same as `FirstSuccess1`. Provided for backward compatibility.

Looping

The following functions let you iterate through the pointers in an array and apply an external function to each pointer.

DoForEach

```
void DoForEach (vEachFunc eachFunc);
```

Starting from item 1 and proceeding forward through the array, this function applies `eachFunc` to each item. `eachFunc` must be declared as:

```
void eachFunc(T *ptr);
```

The same loop can be performed using an iterator:

```
CPtrArrayIterator<T> iter(myArray
                          kStartAtBeginning);
T *item;

while (iter.Next(item))
   eachFunc(item))
```

The iterator form lets you perform the operations inline without requiring an external function.

DoForEach1

```
void DoForEach1 (vEachFunc1 eachFunc
   long param);
```

Starting from item 1 and proceeding forward through the array, this function applies `eachFunc` to each item and the `param`. `eachFunc` must be declared as:

```
void eachFunc(T *ptr, long param);
```

Offset

```
long Offset (T *ptr);
```

Same as `FindIndex`, except that it returns the index position minus one.

Moving items

The following functions permute the array by moving a single item to a different position and shifting other items up or down as appropriate.

The `MoveItemToIndex` function inherited from CArray:

```
MoveItemToIndex(i, j);
```

moves the item at position `i` to position `j`.

MoveToIndex

```
void MoveToIndex (T *ptr, long index);
```

Locates the first occurrence of `ptr` in the array and moves that item to the `index` position.

BringFront

```
void BringFront (T *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item to the start of the array. Same as:

```
MoveToIndex(FindIndex(ptr), 1).
```

SendBack

```
void SendBack (T *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item to the end of the array. Same as:

```
MoveToIndex(FindIndex(ptr), GetNumItems()).
```

MoveUp

```
void MoveUp (T *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item forward one position, if possible. Same as:

```
long i = FindIndex(ptr);
if (i < GetNumItems())
   MoveToIndex(i, i+1);
```

MoveDown

```
void MoveDown (T *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item backward one position, if possible. Same as:

```
long i = FindIndex(ptr);
if (i > 1)
   MoveToIndex(i, i-1);
```

CPtrArrayIterator 87

Introduction

CPtrArrayIterator lets you iterate through the elements of a CPtrArray or CList array.

Heritage

Base Class	CArrayIterator
Derived Classes	None

Using CPtrArrayIterator

CPtrArrayIterator provides a straightforward and error-resistant way to loop through the items in a CPtrArray.

CPtrArrayIterator is a template class, declared as:

```
template <class T>
class CPtrArrayIterator: public CArrayIterator
{ ... };
```

To use a CPtrArrayIterator, specify the class of the pointers to be stored in the array that will be iterated through. For example, to declare an array of pointers to objects of class `MyClass` and an iterator for that array, you must write:

```
CPtrArray<MyClass>   *array;

CPtrArrayIterator<MyClass> iter(array,
                                kStartAtBeginning);
```

For each CPtrArrayIterator class you use, you must expand the template class explicitly. The simplest way to do this is to follow the example of the THINK Class Library template classes. For instance, to expand the class `CPtrArrayIterator<MyClass>`, add to

your project a file named CPtrArray_MyClass.cp that contains these statements:

```
#include "CPtrArrayIterator.h"
#include "MyClass.h"

#pragma template_access public

#pragma template CPtrArrayIterator<MyClass>

TCL_DEFINE_CLASS_M1(
               CPtrArrayIterator<MyClass>,
               CArrayIterator);

#include "CPtrArrayIterator.tem"
```

Here is an example of a CPtrArrayIterator:

```
CPtrArray<MyClass> *array;
...
CPtrArrayIterator<MyClass> iter(array,
                          kStartAtBeginning);
MyClass *item;

while (iter.Next(item))
{
   item->MyFunc();
}
```

The Next function fetches an object pointer from the array and advances the cursor (see below). Successive calls to Next will traverse the entire array. When the cursor is at the end of the array, Next returns FALSE.

You could simply loop through the array with an index, but that sort of code is more error-prone: If the loop inserts or deletes elements in the array, it usually fails.

More than one iterator can operate on the same array at the same time. For example, the loop shown on the next page removes duplicate pointers from the array above.

```
CPtrArrayIterator<MyClass> iter(array,
                                kStartAtBeginning);
CPtrArrayIterator<MyClass> check(array, 0);
MyClass *p, *dup;

while (iter.Next(p))
{
   check.MoveTo(iter.GetCursor());
   while (check.Next(dup))
      if (dup == p)
         array->DeleteItem(check.GetCursor());
}
```

The cursor ranges from 0 to the number of elements in the array. It is logically positioned at the start of the array, at the end, or between two array items. For example, if an array has 5 items, a cursor with value 3 is positioned between items 3 and 4, as illustrated in Figure 87-1. `Next` returns item 4 and `Prev` returns item 3.

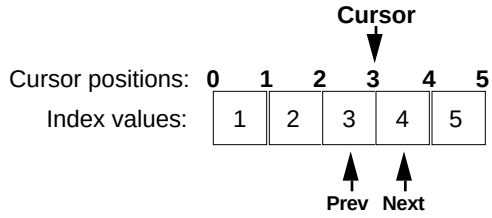


Figure 87-1 Relationship of cursor position to index

CPtrArrayIterator inherits the `GetCursor` and `MoveTo` functions from CArrayIterator, which let you fetch the current cursor position and change the cursor position.

Unlike a loop index, there is no requirement that an iterator go only forward or backward; by calling `Next, Prev`, and `MoveTo,` you may go forward, backward, or jump around in an arbitrary way during the life of the iterator.

If the array associated with the iterator is deleted, the cursor is set to `0` and the `Next` and `Prev` functions returns `FALSE`.

CPtrArrayIterator is implemented in `CPtrArrayIterator.h`.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CPtrArrayIterator

```
CPtrArrayIterator (CPtrArrayT *array,
   CursorPosition start);
```

Constructor. `array` specifies the array to iterate through and `start` specifies where iteration begins. `start` may be specified as `kStartAtBeginning`, `kStartAtEnd`, `StartBefore`*(index)*, or `StartAfter`*(index)*, where *index* is the index of an array item, from 1 to the number of items in the array.

Advancing and retreating

Next

```
Boolean Next (T *&objptr);
```

If there are more items beyond the cursor, this function fetches the next item pointer, replacing the value of `objptr`; then it increments the cursor and returns TRUE. Otherwise, it returns FALSE.

Prev

```
Boolean Prev (T *&itemptr);
```

If there are items before the cursor, this function fetches the previous item pointer, replacing the value of `objptr`; then it decrements the cursor and returns TRUE. Otherwise, it returns FALSE.

CRadioControl 88

Introduction

CRadioControl implements a standard Macintosh radio button.

Note

Earlier versions of the THINK Class Library used a different scheme to handle radio buttons.

Heritage

Base Classes	CButton
	CGroupButton
Derived Classes	None

Using CRadioControl

CRadioControl is a class that implements the standard Macintosh radio button. Since radio buttons always come in groups, a button must be a part of a button group.

Button groups are implemented by the CGroupButton base class. Each button has a group ID. All buttons with the same non-zero group ID are members of the same button group. When a radio button is turned on, all other radio buttons and check boxes in the same group are turned off. When a check box is turned on, other check boxes are unaffected, but all radio buttons in the same group are turned off. Button groups are confined to a single window.

A radio button is a member of button group 1 initially. If there is more than one button group in the window, call `SetGroupID` to explicitly set each button to the desired group.

Like any other button, a radio button can have a command associated with it. Use the `SetClickCmd` function to set a radio

button's command. You can also use any of the other CControl functions to manipulate the radio button.

For compatibility with the previous release, a CRadioControl can also be placed in a "group" by enclosing it in a CRadioGroupPane. See CRadioGroupPane for a discussion. This usage is obsolete.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CRadioControl

```
CRadioControl (short CNTLid,
   CView *anEnclosure,CBureaucrat *aSupervisor);
```

Constructor. Initializes a radio button from a 'CNTL' resource. CNTLid is the resource ID for the radio button. anEnclosure is the pane in which the radio button appears. aSupervisor is the supervisor of the radio button. Initially, the radio button is a member of button group 1.

CRadioControl

```
CRadioControl (short aWidth,
   short aHeight, short aHEncl, short aVEncl,
   StringPtr title, Boolean fVisible,
   short procID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Constructor. Initializes a radio button from the parameters in the argument list. aWidth and aHeight are the width and height of the button in pixels. aHEncl and aVEncl are the horizontal and vertical positions of the button within its enclosure. title is the text to write beside the button. If fVisible is TRUE, the window is drawn immediately after it's created. procID specifies the CDEF used for the control (CDEF resource ID times 16); for a standard radio button, specify radioProc. anEnclosure is the pane in which the radio button appears. aSupervisor is the supervisor of the radio button. Initially, the radio button is a member of button group 1.

CRadioControl

```
CRadioControl ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IRadioControl` for backward compatibility. Initially, the radio button is a member of button group 1.

~CRadioControl

```
~CRadioControl ();
```

Destructor. Broadcasts `radioButtonGoingAway`.

IRadioControl

```
void IRadioControl (short CNTLid,
   CView *anEnclosure,CBureaucrat *aSupervisor);
```

Initialization function included for backward compatibility. May not be called if constructor call has arguments.

INewRadioControl

```
void INewRadioControl (short aWidth,
   short aHeight, short aHEncl, short aVEncl,
   StringPtr title,Boolean fVisible,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialization function included for backward compatibility. May not be called if constructor call has arguments.

Mouse

DoGoodClick

```
void DoGoodClick (short whichPart);
```

When the user clicks and releases the mouse on a radio button and the radio button was off, this function calls `SetValue` to turn on the radio button. Note that `SetValue` in CControl calls `BroadcastChange`. The `ProviderChanged` function in CRadioGroupPane takes care of turning off a button that was on.

Button Group

TurnOff

```
void TurnOff ();
```

Called when a button in the same group is turned on. Calls `SetValue(0)`.

SetValue

```
void SetValue (short aValue);
```

Override this function so that it calls `TellTurningOn` when `aValue` is non-zero and differs from the previous `value`.

IsRadioButton

```
Boolean IsRadioButton ();
```

Returns `TRUE`.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CRadioGroupPane 89

Introduction

CRadioGroupPane is a class that manages a group of radio buttons. This class is obsolete.

Note

This version of the THINK Class Library uses a different scheme to handle radio buttons. See Chapter 59, “CGroupButton.”

Heritage

Base Class	CPane
Derived Classes	None

Using CRadioGroupPane

A radio group pane is a pane specifically designed for grouping radio buttons of class CRadioControl. The radio group pane helps you ensure that only one radio button in a group is on. The button that is on in a radio group is called the station.

To add a button to a radio group pane, simply use the group as the button’s supervisor when you create the button. The group pane assumes that all its subviews are radio buttons. When a radio button is selected, it calls `BroadcastChange` with `reason` set to `controlValueChanged`. The group pane turns off the previously selected button and sets `currentStation` to the newly selected button.

You can respond to clicks in radio buttons in two ways. You could give each radio button a unique ID and use `SetStationID` to set the initial radio button. After that, you would let the radio group

pane manage the radio buttons. When you wanted to find out which button was the station, you would use `GetStationID`.

Alternatively, you could give each radio button its own command by using the `SetClickCmd` function that `CRadioControl` inherits from `CButton`. When you click on a radio button to turn it on, `CRadioControl`'s `DoGoodClick` function calls the `DoCommand` function of the button's supervisor. If the radio button's supervisor is a plain `CRadioGroupPane`, the command should be handled by the group pane's supervisor. Or you can derive a class from `CRadioGroupPane` that provides a `DoCommand` function to handle clicks in radio buttons.

Data Members

Only `CRadioGroupPane` and its derived classes can access the following protected data member:

Data member	Type	Description
`currentStation`	`CRadioControl*`	The currently selected radio button

Member Functions

Creation and destruction

CRadioGroupPane

```
CRadioGroupPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth = 0, short aHeight = 0,
   short aHEncl = 0, short aVEncl = 0,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY);
```

Constructor. `aSupervisor` is the bureaucrat that owns the radio group pane. Typically, the supervisor is a pane or a window.

Note

The other arguments are described in Chapter 71, "CPane."

CRadioGroupPane

```
CRadioGroupPane ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IRadioGroupPane` for backward compatibility.

IRadioGroupPane

```
void IRadioGroupPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialization function compatible with previous release. May not be called if constructor call has arguments.

IViewTemp

```
void IViewTemp (cView *anEncl,
   CBureaucrat *aSupervisor, Ptr viewData);
```

Initializes a `CRadioGroupPane` from a resource template pointed to by `viewData`.

Accessing

SetStationID

```
void SetStationID (long aStationID);
```

Changes the current selection to the button with the specified ID number.

GetStationID

```
long GetStationID ();
```

Returns the ID number of the currently selected radio button. Returns `0` if no station is selected.

Change notification

ProviderChanged

```
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

A radio button in this radio group pane has just been selected. Its `BroadcastChange` function has been called with `reason` set to `controlValueChanged`. This function turns off the previously selected button and sets `currentStation` to the newly selected button.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CRectOvalButton 90

Introduction

CRectOvalButton is a rectangle or oval shape. It can be drawn filled or unfilled, with various pen sizes and styles, as specified in the associated environment object.

A CRectOvalButton object can also function as a button. See Chapter 111, “CSwissArmyButton,” for more information about button styles and behavior.

Heritage

Base Class	CShapeButton
Derived Classes	None

Data Members

Variable	Type	Description
`drawingKind`	`char`	One of `kROBRectKind` or `kROBOvalKind`.

Member Functions

Creation and destruction

CRectOvalButton

```
CRectOvalButton(CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHeight, short aWidth,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY,
   short aButtonkind = kSAPushButton,
   short aHiliteStyle = kSADimHilight,
   short anOnStyle = kSABorderOn,
   Boolean aNeedsUpdate = kSANeedsUpdate,
   short aDrawingKind = kROBRecKind);
```

Constructor. The `anEnclosure`, `aSupervisor`, `aHEncl`, `aVEncl`, `aHSizing`, and `aVSizing` arguments have their standard CPane meanings. Set `aNeedsUpdate` to `TRUE` if the shape is overlapped by another graphic. `aDrawingKind` must be set to either `kROBRectKind` or `kROBOvalKind`, meaning the shape is rectangular or oval, respectively.

CRectOvalButton

```
CRectOvalButton ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`.

Drawing

EraseShape

```
void EraseShape (Rect *theFrame);
```

Erases the shape.

FrameShape

```
void FrameShape (Rect *theFrame);
```

Frames (draws the outline of) the shape.

InvertShape

```
void InvertShape (Rect *theFrame);
```

Inverts the shape for highlighting.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the button to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the button from the stream.

CResFile 91

Introduction

CResFile is a class you use when working with Macintosh resource files.

Heritage

Base Class	CFile
Derived Classes	None

Using CResFile

To learn about Macintosh resources and resource files, see Inside Macintosh I, *Chapter 5*, Inside Macintosh Volume IV, *Chapter 3, and Macintosh TechNote 214.*

If your application reads and writes resources, create a derived class of CResFile and give it member functions to access the resources. CResFile gives you functions to open and close resource files and to set the current resource file.

Since CResFile inherits its behavior from CFile, you need to use one of CFile's specification methods to indicate which file the `Open` method will open.

Data Members

CResFile defines this data member:

Data member	Type	Description
`refNum`	`short`	File system reference number of opened file

Member Functions

CResFile

```
CResFile ();
```

Constructor. Initializes `refNum` to 0.

IResFile
```
void IResFile ();
```
Initialization function for backward compatibility. Does nothing.

Open
```
void Open (SignedByte);
```
Opens the resource file with the specified permission. If the file can't be opened, this function calls `FailResError`. Remember that you need to use one of CFile's specification methods to specify which file to open.

Close
```
void Close ();
```
Closes this resource file. Calls `FailOSErr` if a problem occurs while closing the file.

MakeCurrent
```
void MakeCurrent ();
```
Makes the resource file the current resource file.

IsOpen
```
Boolean IsOpen ();
```
Returns `TRUE` if the resource fork of this file is open.

Update
```
void Update ();
```
Updates this resource file by writing out the changed resource.

HasResFork
```
Boolean HasResFork ();
```
Returns `TRUE` if this file has a resource fork. The file must have been previously specified.

CreateNew
```
void CreateNew (OSType creator, OSType fType);
```
Creates a new resource file. If the file already exists but has no resource fork, this function gives it a resource fork.

CRoundRectButton 92

Introduction

CRoundRectButton is a QuickDraw rounded rectangle shape. It can be drawn filled or unfilled, with various pen sizes and styles, as specified in the associated environment object.

A CRoundRectButton object can also function as a button. See Chapter 111, “CSwissArmyButton,” for more information about button styles and behavior.

Heritage

Base Class	CShapeButton
Derived Classes	None

Using CRoundRectButton

To use a CRoundRectButton object as a graphic, simply construct an instance in an appropriate enclosure. See Chapter 111, “CSwissArmyButton,” for information on using a CRoundRectButton as a button.

Data Members

Data member	Type	Description
`ovalWidth`	`short`	The corner oval width
`ovalHeight`	`short`	The corner oval height

Member Functions

Creation and destruction

CRoundRectButton

```
CRoundRectButton (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHeight, short aWidth,
   short aHEncl, short aVEncl,
   SizingOption aHSizing,
   SizingOption aVSizing,
   short aButtonKind,
   short aHiliteStyle, short anOnStyle,
   Boolean aNeedsUpdate, short anOvalWidth,
   short anOvalHeight);
```

Constructor. The `anEnclosure`, `aSupervisor`, `aHeight`, `aWidth`, `aHEncl`, `aVEncl`, `aHSizing`, and `aVSizing` arguments have their standard CPane meanings. Sets `aNeedsUpdate` to `TRUE` if the shape is overlapped by another graphic. `anOvalWidth` and `anOvalWidth` specify the corner oval width and height.

CRoundRectButton

```
CRoundRectButton ();
```

Default constructor. Implicitly called when object is created by `new_by_name`.

Drawing

EraseShape

```
void EraseShape (Rect *theFrame);
```

Erases the shape.

FrameShape

```
void FrameShape (Rect *theFrame);
```

Frames (draws the outline of) the shape.

InvertShape

```
void InvertShape (Rect *theFrame);
```

Inverts the shape for highlighting.

Accessing

SetOval

```
void SetOval (short aWidth, short aHeight);
```

Sets the corner oval dimensions. If the width and height are set to `MAXINT`, the standard button rounding will be used (half the frame height).

GetOval

```
void GetOval (short *aWidth, short *aHeight);
```

Gets the corner oval dimensions.

Internal

CalcRounding

```
void CalcRounding (Rect *theFrame,
   short *aWidth, short *aHeight);
```

Calculates the oval size.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the shape to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the shape from the stream.

CRunArray 93

Introduction

CRunArray implements a dynamic array of long integers that can conserve space.

Heritage

Base Class	CArray
Derived Classes	None

Using CRunArray

CRunArray implements a dynamic array of long integers. It conserves space if your array contains many sequences of entries with the same value. These sequences are called runs. A run array consists of runs, each of which is represented by a value and the number of consecutive entries that have that value. For example, Figure 93-1 shows how a traditional array and a run array would store the same values.

Traditional Array

Index	Value
1	2
2	64
3	64
4	64
5	45

Run Array

Run	#Entries	Value
1	1	2
2	3	64
3	1	45

Figure 93-1 A traditional array and a run array

To put an entry into the array, you can choose between `SetValue` and `InsertValue`. `SetValue` replaces an entry with a new entry.

InsertValue inserts a run of values into the array. To delete an entry, use DeleteValue. To sum a range of entries in the array, use the functions SumRange and FindSum.

When used with a run array, many CArray functions operate on runs, not on individual values. This table lists CArray functions for which there are similar CRunArray functions that you should use instead.

CArray function	CRunArray function
GetItem	GetValue
SetItem	SetValue
InsertAtIndex	InsertValue
DeleteItem	DeleteValue

In a CRunArray, the data member numItems contains the number of runs in the array. To find the number of entries, use the function GetNumItems instead.

Data Members

CRunArray defines these protected data members:

Data member	Type	Description
itemCount	long	Number of entries in the array.
hRuns	tRunHndl	Handle to runs. Same as CArray::hItems.

Member Functions

Creation and destruction

CRunArray

```
CRunArray (short blockSize);
```

Constructor. blockSize specifies the number of elements to add to the array each time more memory is allocated. The default value is 3. The number of items and runs is set to 0 (zero).

CRunArray

```
CRunArray (CRunArray& source);
```

Copy constructor.

IRunArray

```
void IRunArray ();
```

Initialization function for backward compatibility. Does nothing.

Accessing

GetNumItems

```
long GetNumItems ();
```

Returns the number of items in the array.

Insertion and deletion

InsertValue

```
void InsertValue (long item, long value,
   long count);
```

Inserts a run of values into the array. `item` is the index at which to start the run. If you specify an index beyond the end of the array, the run is added to the end of the array. `value` is the value for all entries in this run. `count` is the length of the run.

SetValue

```
void SetValue (long index, long value);
```

Sets the value of the entry at `index` to `value`.

DeleteValue

```
void DeleteValue (long index);
```

Deletes the entry at `index`. All the entries following `index` move up one position.

DisposeAll

```
void DisposeAll ();
```

Deletes all the entries in this array.

Membership

GetValue

```
long GetValue (long index);
```

Returns the value of the entry at `index`.

Summing

SumRange

```
long SumRange (long startIndex, long endIndex);
```

Returns the sum of the items between `startIndex` and `endIndex`, inclusive.

FindSum

```
long FindSum (long aSum);
```

Returns the index of the first entry such that all the entries from 1 to that entry have a sum equal to or greater than `sum`.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

Run handling

CRunArray uses these protected member functions to manipulate runs. You should only need to use them if you are implementing a derived class of CRunArray.

FindRun

```
void FindRun (long itemIndex, long *runIndex,
   long *firstInRun);
```

Returns the number of the run that contains the entry at `itemIndex`. This function sets `runIndex` to the run number and `firstInRun` to the index of the first entry in the run. If `itemIndex` is not in the array, this function sets `runIndex` and `firstInRun` to `BAD_INDEX`.

InsertRun

```
void InsertRun (long index, long runLength,
   long value);
```

Inserts a new run into the array. `index` is the index of the first entry in the run. `runLength` is the number of entries in the run. `value` is the value of all the entries in the run.

DeleteRun

```
void DeleteRun (long runIndex);
```

Deletes the run at number `runIndex` from the array.

CSaver 94

Introduction

CSaver is a CDocument derived class that automates standard Open, Close, Save, and Revert processing. CSaver uses Object I/O to save and restore the objects pointed to by the `itsContents` data member.

Heritage

Base Class	CDocument
Derived Classes	None

Using CSaver

CSaver is a template class. To use CSaver as the base class for your document class, you must specify the class of CSaver's `itsContents` data member, for example:

```
class MyDoc : public CSaver<MyData>
```

You must also add a file to your project to explicitly expand the template. By convention, you should name your source file `CSaver_MyData.cpp`. The file should contain at least the following:

```
#include <CSaver.h>
#include <MyClass.h>

#pragma template_access public
#pragma template CSaver<MyClass>

#include <CSaver.tem>
```

See Chapter 8, "Using Object I/O," for more information.

Data Members

CSaver defines the following data member:

Data member	Type	Description
`itsContents`	`any`	Your document contents

Member Functions

Creation and destruction

CSaver

```
CSaver (Boolean printable);
```

Constructor. Sets `printable` to `TRUE` if the document's main window can be printed.

CSaver

```
CSaver ();
```

Default constructor. Implicitly called when object is created by `new_by_name`.

~CSaver

```
~CSaver ();
```

Destructor. Deletes `itsContents`.

New/Open

NewFile

```
void NewFile ();
```

Initializes a new document by calling `MakeNewWindow`, `MakeNewContents`, and `ContentsToWindow`. Leaves `itsFile` equal to `NULL`, indicating that the document has not been saved.

OpenFile

```
void OpenFile (SFReply *macSFReply);
```

Initializes a new document by opening the specified file, reading `itsContents` using object I/O, and calling `MakeNewWindow` and `ContentsToWindow`.

Save/SaveAs/Revert

DoSave
```
Boolean DoSave ();
```
Saves by calling `WindowToContents` and writing `itsContents` to the file using object I/O. If `itsFile` is `NULL`, the function calls `DoSaveFileAs` first. Returns `TRUE` if file is saved successfully; otherwise, returns `FALSE`.

DoSaveAs
```
Boolean DoSaveAs (SFReply *macSFReply);
```
Creates a new file on disk, and then calls `DoSave` to fill it. Returns the value returned by `DoSave`.

DoRevert
```
void DoRevert ();
```
Reverts by throwing away existing window and its contents. Then makes a new window and calls `ReadDocument` or `NewFile`, as appropriate, to fill it.

Member Functions: Protected

Input/Output

MakeNewWindow
```
void MakeNewWindow ();
```
Pure virtual function. Derived classes must override this function.

ReadDocument
```
void ReadDocument ();
```
Reads `itsContents` from file and calls `ContentsToWindow` to display the contents.

WriteDocument
```
void WriteDocument ();
```
Calls `WindowToContents` and then writes `itsContents` to file.

MakeNewContents
```
void MakeNewContents ();
```
Pure virtual function. Derived classes must override this function.

ReadContents
```
void ReadContents (CFileStream *aStream);
```
Reads `itsContents` from stream. Derived classes should override this function if more than `itsContents` needs to be read.

WriteContents
```
void WriteContents (CFileStream *aStream);
```
Writes `itsContents` to stream. Derived classes should override this function if more than `itsContents` needs to be written.

ContentsToWindow
```
void ContentsToWindow ();
```
Derived classes should override this function. Displays `itsContents` in the document's window.

WindowToContents
```
void WindowToContents ();
```
Derived classes may override this function to transfer the information in the document's window to `itsContents`.

Utility

ExistingFile
```
void Existing File ( );
```
Called when a new file is created with the same name as an existing file and the user has already approved the replacement. The default function throws out the existing file and makes a new one of the same type. Derived classes may override this function—for example, in order to carry along resources or other data from an existing file.

NewFileType
```
OSType NewFileType ();
```
Returns the file type for the document. The default function returns the first file type in the application's `sfFileTypes` array. Derived classes may override this function.

PositionWindow
```
void PositionWindow ();
```
Positions a new window on the screen. The default function staggers the window. Derived classes may override this function.

MakeWindowName

```
void MakeWindowName (Str255 name);
```

Returns a window title. The default function returns the file name if the document already has one. If the document does not have a file name yet `MakeWindowName` creates a new name by appending `-<untitled window count>` to either the title of the `'WIND'` resource, or to the string from the `strUNTITLED` index of the `'STR#'` resource `STRcommon`. Derived classes may override this function.

CScrollBar 95

Introduction

CScrollBar implements a standard Macintosh scroll bar.

Heritage

Base Class	CControl
Derived Classes	None

Using CScrollBar

This class implements a Macintosh scroll bar. To make scroll bars easier to use, this class distinguishes between a mouse click in an indicator (the scroll box or the “thumb”) and a click in any other part of the scroll bar. Both behaviors are implemented in the `DoClick` function of the CControl class.

When the user clicks any part other than an indicator, `DoClick` calls the Toolbox routine `TrackControl` with an action procedure, or action proc. The action procedure is called repeatedly as long as the user holds down the mouse button in the stationary part of a control. There is no default action procedure. In most cases, the scroll bar controls a panorama. To set the action proc, use the `SetActionProc` function inherited from CControl.

When the user clicks the indicator, `DoClick` calls the `DoThumbDragged` function of the scroll bar, which calls a thumb function that you provide. The thumb function is called after the user has moved the thumb of the scroll bar. There is no default thumb function. Thumb functions are unique to scroll bars. Use `SetThumbFunc` to set the thumb function.

Usually, you use a scroll bar to control a panorama. The CScrollPane class handles the usual cases of horizontal and vertical scroll bars, so

you don't have to explicitly create a CScrollBar object, or provide an action proc or a thumb function.

Data Members

CScrollBar defines these data members:

Data member	Type	Description
`theOrientation`	`Orientation`	The way scroll bar lies, horizontal or vertical
`theThumbFunc`	`ThumbFuncType`	Function to call after a thumb drag

Member Functions

Creation and destruction

CScrollBar

```
CScrollBar (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   Orientation anOrientation,
   short aLength, short aHEncl, short aVEncl);
```

Constructor. `anEnclosure` is the pane or window to which the scroll bar belongs. `anEnclosure` is usually a CScrollPane object. `aSupervisor` is the scroll bar's supervisor in the chain of command. Orientation is either `HORIZONTAL` or `VERTICAL`. `aLength` is the length of the scroll bar. `aHEncl` and `aVEncl` are the horizontal and vertical position of the upper-left corner of the scroll bar.

CScrollBar

```
CScrollBar ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IScrollBar` for backward compatibility.

IScrollBar

```
void IScrollBar (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   Orientation anOrientation,
   short aLength, short aHEncl, short aVEncl);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

IScrollBarX

```
void IScrollBarX (Orientation anOrientation,
   short aLength);
```

Completes scroll bar initialization.

Accessing

SetThumbFunc

```
void SetThumbFunc (ThumbFuncType aThumbFunc);
```

Sets `aThumbFunc` to be the scroll bar's thumb function. The default `DoClick` function for controls calls the control's `DoThumbDragged` function when the user moves an indicator in a control. The `DoThumbDragged` function for scroll bars calls the thumb function. You should declare thumb function as follows:

```
void MyThumbFunc (CControl *theControl,
   short delta);
```

`theControl` is the control whose indicator was moved. `delta` is the amount by which the value has changed. To get the current value of the control, you can call its `GetValue` function.

Appearance

Offset

```
void Offset (long hOffset, long vOffset,
   Boolean redraw);
```

Moves the control by `hOffset`, `vOffset` pixels. If `redraw` is `TRUE`, redraws the control after moving it.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the scroll bar. If the scroll bar is active, this function draws it the normal way. If the scroll bar is inactive, only the frame of the scroll bar is drawn.

Activate

```
void Activate ();
```

Activates the scroll bar.

Deactivate

```
void Deactivate ();
```

Deactivates the scroll bar.

Click response

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handles a click in the scroll bar. If the scroll bar's enclosure is a scroll pane, this function calls `AdjustScrollMax`.

DoThumbDragged

```
void DoThumbDragged (short delta);
```

If the scroll bar has an associated thumb function, call it with the scroll bar and `delta` as arguments. The default `DoClick` for controls calls the control's `DoThumbDragged` function when the user moves an indicator in a control. See `SetThumbFunc`, earlier in this chapter.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CScrollPane 96

Introduction

CScrollPane implements a pane with scroll bars that control a panorama.

Heritage

Base Class	CPane
Derived Classes	None

Using CScrollPane

A scroll pane uses scroll bars to control what is displayed in a panorama.

To use a scroll pane, install a panorama with the `InstallPanorama` function. The scroll pane uses the scale of the panorama for the values of the scroll bars.

The scroll bars and the panorama do not communicate directly. Mouse clicks in the scroll bars are reported to the scroll pane, which then tells the panorama how to scroll or shift its image. Similarly, changes in the panorama that affect the scroll bars are reported to the scroll pane, which then adjusts the scroll bars.

Many applications use a scroll pane that occupies most of a window. After creating a scroll pane, you can call its `FitToEnclosure` function to make it as big as the window. Then, create the panorama with the scroll pane as its enclosure, call the panorama's `FitToEnclosure` function to resize the panorama to fit within the scroll pane, and call `InstallPanorama` to complete initialization of the scroll pane.

Data Members

CScrollPane defines the following data members:

Data member	Type	Description
itsPanorama	CPanorama*	The scrollable view. The "content" of the scroll pane.
itsHorizSBar	CScrollBar*	The scroll pane's horizontal scroll bar.
itsVertSBar	CScrollBar*	The scroll pane's vertical scroll bar.
itsSizeBox	CSizeBox*	The scroll pane's size box.
hExtent	long	For internal use.
vExtent	long	For internal use.
hUnit	short	For internal use.
vUnit	short	For internal use.
hSpan	short	For internal use.
vSpan	short	For internal use.
hStep	short	Number of horizontal units to scroll by when the user clicks an arrow.
vStep	short	Number of vertical units to scroll by when the user clicks an arrow.
hOverlap	short	Number of units to overlap when the user clicks a page region.
vOverlap	short	Number of units to overlap when the user clicks a page region.

Member Functions

Creation and destruction

CScrollPane

```
CScrollPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
```

```
   Boolean hasHoriz, Boolean hasVert,
   Boolean hasSizeBox);
```

Constructor. All but the last three arguments are the same as the arguments used with the Cpane constructor.

If `hasHoriz` is `TRUE`, the scroll pane has a horizontal scroll bar. If `hasVert` is `TRUE`, the scroll pane has a vertical scroll bar. If `hasSizeBox` is `TRUE`, the scroll pane draws a size box in the lower-right corner of the pane. The other arguments are described in Chapter 71, "CPane."

CScrollPane

```
CScrollPane ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IScrollPane` for backward compatibility.

IScrollPane

```
void IScrollPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   Boolean hasHoriz, Boolean hasVert,
   Boolean hasSizeBox);
```

Initialization function included for backward compatibility. May not be called if constructor has arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally to initialize from a template. Each derived class of CView overrides this function to use its own template.

To initialize a scroll pane from a resource file, use a `'ScPn'` resource.

IScrollPaneX

```
void IScrollPaneX (Boolean hasHoriz,
   Boolean hasVert, Boolean hasSizeBox);
```

Creates the scroll bars and size box, if any.

Accessing

InstallPanorama

```
void InstallPanorama (CPanorama *aPanorama);
```

Establishes `aPanorama` as the panorama associated with this scroll pane. Calls the scroll pane's `AdjustScrollMax` and `Calibrate` functions.

SetSteps

```
void SetSteps (short aHStep, short aVStep);
```

Sets the amount to scroll when the user clicks the arrows of a scroll bar. The units are in the panorama's units.

GetSteps

```
void GetSteps (short *theHStep, short *theVStep);
```

Gets the amount to scroll when the user clicks the arrows of a scroll bar. The units are in the panorama's units.

SetOverlaps

```
void SetOverlaps (short aHOverlap,
   short aVOverlap);
```

Sets the amount of overlap when the user clicks the page (gray) regions of the scroll bar. The units are in the panorama's units.

GetInterior

```
void GetInterior (LongRect *theInterior);
```

Gets the interior of the scroll pane. The interior excludes the space that the scroll bars occupy.

Scroll bar

AdjustScrollMax

```
void AdjustScrollMax ();
```

Adjusts the maximum value of the scroll bars from the extent and frame size of the panorama.

Calibrate

```
void Calibrate ();
```

Adjusts the scroll bar's thumb when the position of the frame of the panorama changes.

ChangeSize

```
void ChangeSize (Rect *delta, Boolean redraw);
```

Changes the size of a scroll pane. Each component of the `delta` rectangle specifies how each side changes. Positive values mean down and to the right; negative values mean up and to the left. If redraw is `TRUE`, the scroll pane is redrawn on the next update event.

Scroll performance

DoHorizScroll

```
void DoHorizScroll (short whichPart);
```

Scrolls horizontally. `whichPart` specifies which part of the scroll bar was hit. This function calls `DoScroll` (see below).

DoVertScroll

```
void DoVertScroll (short whichPart);
```

Scrolls vertically. `whichPart` specifies which part of the scroll bar was hit. This function calls `DoScroll` (see below).

DoThumbDrag

```
void DoThumbDrag (short hDelta, short vDelta);
```

Adjusts the panorama when the scroll box (thumb) has been dragged. This function calls the panorama's `DoScroll` function.

DoScroll

```
void DoScroll (long hDelta, long vDelta);
```

Scrolls the panorama belonging to this pane by `hDelta` units horizontally and `vDelta` units vertically. The units are given in the

panorama's coordinates. All the other functions in this class call this function to handle scrolling.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Helper Functions

Note that the following are not member functions.

SBarActionProc

```
pascal void SBarActionProc (ControlHandle
   macControl, short whichPart);
```

The Toolbox `TrackControl` routine calls this function continuously while the mouse is down in any part of the scroll bar except the scroll box (thumb). This function calls the `DoHorizScroll` and `DoVertScroll` functions of the scroll bar's supervisor (the scroll pane), which actually scroll the panorama.

SBarThumbFunc

```
pascal void SBarThumbFunc (CScrollBar *theSBar,
   short delta);
```

The CScrollBar's `DoThumbDrag` function calls this routine after the thumb of a scroll bar has been moved. The `DoClick` function calls `DoThumbDrag` when the user moves the indicator of a control. `SBarThumbFunc` calls the `DoThumbDrag` function of the scroll bar's supervisor (the scroll pane).

Member Functions: Private

CScrollPaneX

```
void CScrollPaneX ( );
```

Performs common initialization.

CSelector 97

Introduction

CSelector is an abstract class for drawing panes with several items from which users can choose. A tool palette is an example of a selector.

Heritage

Base Class	CPanorama
Derived Class	CGridSelector

Using CSelector

CSelector is an abstract class that defines the basic behavior of a pane that lets the user choose from several items. A good example of a selector is a tool palette or a pattern palette. The THINK Class Library includes CGridSelector, a class for implementing those kinds of palettes, and its descendants CPatternGrid, for displaying the standard patterns, and CCharGrid, for displaying characters in a grid. You can use these classes directly for pattern palettes and tool palettes. To implement a different kind of selector, you need to create a derived class of CSelector.

A selector works like a menu without command numbers. Each selector has a command base, which is like a menu ID. You set the command base when you create a selector, or you can set it after the selector has been created. A selector contains items. You specify the number of items when you create the selector.

Every selector must have a `Draw` function to display its items. How the items appear in the selector and how they're arranged is up to you. You also need to override the `FindItem` function to determine which item was clicked, as well as the `HiliteItem` function to highlight the selected item. If you want your selector to respond to double-clicks, override the `DoDoubleClick` function as well.

Data Members

CSelector defines the following data members:

Data member	Type	Description
numItems	short	The number of items from which to choose
selection	short	The currently selected item
commandBase	short	The base value for converting selections into command numbers

Member Functions

Creation and destruction

CSelector

```
CSelector (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY,
   short aNumItems = 0, short aSelection = 0,
   short aCommandBase = 0);
```

Constructor. All but the last three arguments to this function are identical to those used with the CPane's constructor.

`aNumItems` specifies the number of items in this selector. `aSelection` is the initial item. `aCommandBase` is the base value for converting the selected item into a command number. The other arguments are described in Chapter 71, "CPane."

CSelector

```
CSelector ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `ISelector` for backward compatibly.

ISelector

```
void ISelector (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
```

```
short aNumItems, short aSelection,
short aCommandBase);
```

Initialization function provided for backward compatibility. May not be called if constructor has arguments.

Mouse

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

This function converts a click into a command number, which it passes to the `DoCommand` function of the selector's supervisor. `DoClick` calls the selector's `FindItem` function to find out which item was clicked.

Once it has an item, `DoClick` uses the same scheme as CBartender to build a command number. It puts the command base in the high word of a long integer and the item number in the low word, and negates the resulting long integer.

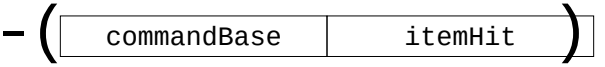


Figure 97-1 How DoClick builds the command number

If the new selection is not the same as the current selection, `DoClick` calls the selector's `ChangeHilite` function. If the new selection is the same as the current selection, and there was a double-click, `DoClick` calls the selector's `DoDoubleClick` function.

Since most of the work is done in member functions that you must override, your CSelector-derived class shouldn't need to override this function.

HitSamePart

```
Boolean HitSamePart (Point pointA, Point pointB);
```

Returns `TRUE` if the mouse went down in the same item, `FALSE` otherwise. The default function checks to see if `FindItem(pointA)` and `FindItem(pointB)` return the same item. Your CSelector-derived class should not need to override this function.

Accessing

ChangeSelection

```
void ChangeSelection (short aSelection);
```

Changes the selection from the current selection to `aSelection`. If `aSelection` is not the same as the current selection, this function turns off the highlighting of the current selection and turns on the highlighting of the new selection. Your derived class should not need to override this function.

GetSelection

```
short GetSelection ();
```

Returns the current selection. You shouldn't need to override this function.

SetCommandBase

```
void SetCommandBase (short aCommandBase);
```

This function sets the selector's command base. `CSelector` uses the command base to build a command number from the item hit. Your derived class should not need to override this function.

GetCommandBase

```
short GetCommandBase ();
```

Returns the value of the selector's command base. Your derived class should not need to override this function.

HiliteItem

```
void HiliteItem (short theItem,
   HiliteState state);
```

Highlights the specified item. `HiliteItem` can take one of three values for the `state` parameter, as shown here:

HiliteState value	Behavior
`hiliteOFF`	Turns highlighting off for the item
`hiliteON`	Turns highlighting on for the item
`hiliteDYNAMIC`	Flashes the selected item without selecting or deselecting

You can decide whether to highlight the items of your selector. In most cases, inverting the rectangle that encloses the specified item is sufficient. When highlighting is on, the item should be inverted.

When highlighting is off, the item should appear normally. Dynamic highlighting is used when the selector is used as a menu. See Chapter 99, “CSelectorMDEF.”

FindItem

```
short FindItem (Point hitPt);
```

Determines which item corresponds to a mouse down at a specified point. You decide how your selector arranges and displays its items. Typically, items are arranged in a grid or a table. Your derived class must override this function.

DoDoubleClick

```
void DoDoubleClick ();
```

Responds to a double-click. The first click sets the selection, so the double-click pertains to that item. If your selector-derived class responds to double-clicks, you should override this function.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

97 CSelector

CSelectorDirector 98

Introduction

This class is retained for backward compatibility.

Heritage

Base Class	CTearOffMenu
Derived Classes	CPICTGridDirector

Using CSelectorDirector

You should not have to create instances of this class, or derive classes from it directly.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CSelectorDirector

```
CSelectorDirector (short WINDid,
   tDragBar dragBar);
```

Constructor. Calls the CTearOffMenu constructor with the argument `WINDid`, then sets margins using `dragBar`.

CSelectorDirector

```
CSelectorDirector ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `ISelectorDirector` for backward compatibility.

~CSelectorDirector

```
~CSelectorDirector ();
```

Destructor.

ISelectorDirector

```
void ISelectorDirector (short WINDid,
   tDragBar dragBar);
```

Initialization function provided for backward compatibility. If no constructor arguments are specified, new object must be further initialized by calling `ISelectorDirector`. If any constructor arguments are specified, `ISelectorDirector` must not be called. See constructor for descriptions of the arguments.

Commands

SelectTool

```
void SelectTool (short theTool);
```

Sets the current tool selection to `theTool`.

CSelectorMDEF 99

Introduction

CSelectorMDEF is a class that lets you use descendants of CSelector as custom menus.

Heritage

Base Class	CPaneMDEF
Derived Classes	CGridMDEF

Using CSelectorMDEF

CSelectorMDEF is a class that handles menu selection for custom menus based on CSelector panes. The pane passed to the `CSelectorMDEF` constructor must be derived from CSelector; it can be used as a custom menu. CSelectorMDEF's `ChooseItem` function takes care of selecting items from the CSelector. If you pass `CSelectorMDEF` a descendant of CTearOffMenu, you can use your custom menu as a tear-off menu.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CSelectorMDEF

```
CSelectorMDEF (short MDEFid, CPane *aPane,
   CTearOffMenu *aTearOffMenu);
```

Constructor. Passes the arguments to CPaneMDEF's constructor. `aPane` must be a selector pane descended from CSelector. `aTearOffMenu` must be a descendant of CTearOffMenu. If `NULL` is specified, the menu cannot be torn off.

CSelectorMDEF

```
CSelectorMDEF ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `ISelectorMDEF` for backward compatibility.

ISelectorMDEF

```
void ISelectorMDEF (short MDEFid, CPane *aPane,
   CTearOffMenu *aTearOffMenu);
```

Initialization function for backward compatibilty. May not be called if constructor has arguments.

ChooseItem

```
void ChooseItem (MenuHandle macMenu,
   Rect *menuRect, Point hitPt,
   short *whichItem);
```

This function handles menu selection for custom menus.

If the `hitPt` is within the `menuRect`, this function sets up the QuickDraw drawing environment to match the pane's drawing environment. The QuickDraw origin is set so that the point (0, 0) is the upper-left corner of the pane. `ChooseItem` then calls the member function `FindItem` of the pane associated with this object.

CShapeButton 100

Introduction

CShapeButton is an abstract base class for the rectangle, oval, and rounded-rectangle shapes.

Heritage

Base Class	CSwissArmyButton
Derived Classes	CPolyButton
	CRectOvalButton
	CRoundRectButton

Using CShapeButton

CShapeButton is not used directly, but provides common functions for its derived classes.

Data Members

CShapeButton defines the following data member:

Data member	Type
`pen`	`Point`

Member Functions

Creation and destruction

CShapeButton

```
CShapeButton (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHeight, short aWidth,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY,
   short aButtonKind = kSAPushButton,
```

```
short aHiliteStyle = kSADimHilight,
Short anOnStyle = kSABorderOn,
Boolean aNeedsUpdate = kSANeedsUpdate);
```

Constructor. The `anEnclosure`, `aSupervisor`, `aHEncl`, `aVEncl`, `aHSizing`, and `aVSizing` arguments have their standard CPane meanings. This function sets `aNeedsUpdate` to `TRUE` if the shape is overlapped by another graphic.

CShapeButton

```
CShapeButton ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IShapeButton` for backward compatibility.

Accessing

SetPen

```
void SetPen (Point aPen);
```

Sets the pen width and height.

GetPen

```
Point GetPen ();
```

Returns the pen width and height.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the shape by calling `EraseShape`, `FrameShape`, and `InvertShape`.

DrawButton

```
void DrawButton (Boolean hilite);
```

Draws the button normally or highlighted.

EraseShape

```
void EraseShape (Rect *theFrame);
```

Derived classes must override this function to erase the shape.

FrameShape

```
void FrameShape (Rect *theFrame);
```

Derived classes must override this function to frame (draw the outline of) the shape.

InvertShape

```
void InvertShape (Rect *theFrame);
```

Derived classes must override this function to invert the shape for highlighting.

Tracking

InButton

```
Boolean InButton (Point aPoint);
```

Returns `TRUE` if `aPoint` is within this button.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the button to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the button from the stream.

Member Functions: Protected

IShapeButton

```
void IShapeButton (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aHeight, short aWidth,
   short aHEncl, short aVEncl,
   SizingOption aHSizing,
   SizingOption aVSizing,
   short aButtonKind, short aHilightStyle,
   short anOnStyle, Boolean aNeedsUpdate);
```

Initialization function called by derived classes that need to calculate arguments during construction.

CSizeBox 101

Introduction

CSizeBox implements a Macintosh grow icon.

Heritage

Base Class	CPane
Derived Classes	None

Using CSizeBox

In most cases, you do not need to use this class yourself. It is used by CScrollPane to draw a Macintosh size box at the lower-right corner of any pane, not just in the lower-right corner of a window.

The first version of the THINK Class Library used an `'SICN'` resource to draw the grow icon. With System 7, using this is no longer a good idea. If the `useSICN` flag is `TRUE`, the `Draw` function uses the resource; otherwise, it calls the Toolbox routine `DrawGrowIcon`.

Resource	Description
`'SICN' 200`	The grow icon

For System 7 compatibility, `useSICN` should remain `FALSE`.

Data Members

CSizeBox defines one data member:

Data member	Type	Description
`useSICN`	`Boolean`	`TRUE`, if this size box uses a `'SICN'` resource instead of calling `DrawGrowIcon`

Member Functions

Creation and destruction

CSizeBox

```
CSizeBox (CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. Places the size box at the lower right of its enclosure. The sizing options for a size box are `sizFIXEDRIGHT` and `sizFIXEDBOTTOM`. By default, `useSICN` is `FALSE`.

CSizeBox

```
CSizeBox ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `ISizeBox` for backward compatibility.

ISizeBox

```
void ISizeBox (CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialization function provided for backward compatibility. May not be called if constructor has arguments.

Appearance

Draw

```
void Draw (Rect *area);
```

Draws a size box. If the pane is active, this function draws the size box; if the pane is inactive, it draws a white rectangle. If `useSICN` is `FALSE`, this function uses the Toolbox's grow icon; otherwise, it uses a `'SICN'` resource to draw the grow icon.

Activate

```
void Activate ();
```

Calls the `Draw` function of the size box. Called when the enclosure to which the size box belongs is becoming active.

Deactivate

```
void Deactivate ();
```

Calls the `Draw` function of the size box. Called when the enclosure to which the size box belongs becomes inactive.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CStaticText 102

Introduction

CStaticText displays text that cannot be edited or selected.

Heritage

Base Class	CEditText
Derived Classes	None

Using CStaticText

Use this class exactly as you would CEditText, with the difference that editing and selection are disabled.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CStaticText

```
CStaticText (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth =0, short aHeight =0,
   short aHEncl =0, short aVEncl =0,
   SizingOption aHSizing = sizELASTIC,
   SizingOption aVSizing = sizELASTIC,
   short aLineWidth = -1);
```

Constructor. All of the arguments are identical to those of the CEditText constructor. `aLineWidth` specifies how wide the lines should be. If it's less than zero, the width is the same as the Macintosh TE record's viewRect. `aScrollHoriz` is `TRUE` if the text should scroll horizontally when the line width is greater than the

frame width and the cursor moves outside the frame. A CStaticText object is initialized to be neither editable nor selectable.

CStaticText

```
CStaticText ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IStaticText` for backward compatibility.

IStaticText

```
void IStaticText (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initialization function provided for backward compatibility. May not be called if constructor has arguments.

CStdPopupPane 103

Introduction

CStdPopupPane implements a pop-up menu pane that displays the menu's title and currently selected item. A pop-up menu pane is a pane that owns a CPopupMenu and pops up the menu when clicked.

Heritage

Base Class	CPopupPane
Derived Classes	None

Using CStdPopupPane

CStdPopupPane is a derived class of CPopupPane that follows Apple's *Human Interface Guidelines* for a pop-up menu. It displays a title and a box with the current selection and an arrow, as shown in Figure 103-1. Clicking the menu pops it up, inverting the current selection and the menu title. Clicking the title does nothing.

Figure 103-1 A CStdPopupPane

CStdPopupPane lets you select only one item, and automatically checks the item when it pops up the menu. (In other words, it sets `radioStyle` to `TRUE`.) If you want to select more than one item or you need your program to check the item, you can either use the CArrowPopupPane described in Chapter 22, "CArrowPopupPane," or change the values of the `multiSelect`, `autoSelect` in and `radioStyle` flags in the associated pop-up menu.

Setting menu font and size

CPopupMenu draws the menu in the current grafport's font and size. There are two ways to draw a CStdPopupPane in other than the system default font and size. You can set the pane's `textFont` and `textSize` data members by calling the `SetTextFont` member function. Or, as with other panes, you can set the pane's `itsEnvironment` data member to point to a CTextEnvirons object with the desired characteristics. (The latter approach works for any derived class of CPopupPane.) If `itsEnvironment` is non-`NULL`, the environment is honored and the `textFont` and `textSize` are ignored.

Data Members

CStdPopupPane defines the following protected data members:

Data member	Type	Description
`textFont`	`short`	The font for the menu's title and selection. It is the system font.
`textSize`	`short`	The size of the font for the menu's title and selection. It is the system's default font size.
`titleWidth`	`short`	The width of the menu's title.
`fontAscent`	`short`	The maximum distance above the baseline for `textFont`.
`itsPopupBox`	`CPane*`	The CPaneBorder for drawing the menu's box and drop shadow.
`popupPt`	`Point`	Where the menu pops up.
`sicnPt`	`Point`	Where the Down Arrow is drawn.
`hiliteRect`	`Rect`	The rectangle to highlight when the menu pops up. It includes the menu title and selection.

Member Functions

Creation and destruction

CStdPopupPane

```
CStdPopupPane (short menuID,
   CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   Boolean fAutoSelect, Boolean fMultiSelect,
   Boolean fRadioStyle);
```

Constructor. The `menuID` specifies the `'MENU'` resource displayed in the pane. To have the `CalcDimensions` member function calculate the pane's size from the contents of the menu (the only way to conform to the Apple *Human Interface Guidelines*), set `aWidth` and `aHeight` to `kAutoSize`. The default value of `fRadioStyle` is `TRUE`, and the default values of `fAutoSelect` and `fMultiSelect` are `FALSE`; hence, you can select only one item. The menu automatically checks that item when it pops up, holding the selected item internally so that two uses of the same `'MENU'` resource do not interfere with each other.

The remaining arguments are the same as for the CPane constructor.

CStdPopupPane

```
CStdPopupPane ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IStdPopupPane` for backward compatibility.

IStdPopupPane

```
void IStdPopupPane (short menuID,
   CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   Boolean fAutoSelect, Boolean fMultiSelect,
   Boolean fRadioStyle);
```

Initialization function for backward compatibility. Must not be called if constructor has arguments. The default value of `fAutoSelect` is `TRUE`, and the default values of `fRadioStyle` and `fMultiSelect` are `FALSE`. Note that for compatibility, these defaults are different from those of the constructor.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

Used internally to initialize a pop-up menu pane from a template. Each derived class of CView overrides this function to use its own template.

PreloadStdPopup

```
static void PreloadStdPopup (short menuID);
```

Preloads a menu that one or more standard pop-up panes will use later. Call this function when you must initialize a menu (by calling `AddResMenu`, for example) before using it.

Mouse

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handles a click in the pop-up menu pane. This function pops up the menu only when you click the menu (not the menu title), and it highlights the title while the menu is popped up. Override this function if your derived class handles clicks differently.

Command

DoKeyDown

```
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handles a key-down event. If the event corresponds to an item in the menu, `DoKeyDown` selects it. If you want this function to be called automatically, you must put the pop-up menu in the chain of command. Override this function if your derived class handles key-down events differently.

NewMenuSelection

```
void NewMenuSelection (short itemSelected);
```

This function puts the new item's text in the menu. It is called when the user selects a new item in the menu.

Activate
```
void Activate ();
```
Activates the menu pop-up pane and sets it to be redrawn with a solid black pattern at the next update event.

Deactivate
```
void Deactivate ();
```
Deactivates the menu pop-up pane and draws it with a solid gray pattern.

CalcPopupBox
```
void CalcPopupBox ();
```
Recalculates the dimensions of the pop-up box. When you add, delete, or change the text of menu items or change font characteristics, call this function to ensure that the width of the pop-up box matches the width of the menu.

Drawing

Prepare
```
void Prepare ();
```
Prepares the port and coordinate system before drawing. This function sets the text characteristics for the pop-up menu pane. Override this function if your derived class needs to perform other preparations.

Draw
```
void Draw (Rect *area);
```
Draws the pop-up menu title, currently selected item, and Down Arrow. The pop-up menu box is a subpane that is drawn automatically.

Object I/O

PutTo
```
void PutTo (CStream& aStream);
```
Writes to the stream.

GetFrom
```
void GetFrom (CStream& aStream);
```
Reads from the stream.

Member Functions: Protected

IStdPopupPaneX

```
void IStdPopupPaneX (Boolean fFirstInstall);
```

Performs common initialization; calls `InstallHook` if appropriate.

HookMenu

```
void HookMenu (MenuHandle menu);
```

Installs a hook into the menu defproc of the pop-up menu that forces the menu to be wide enough to contain a standard Down Arrow. To disable the hook, comment out this line at the top of the file `CStdPopupPane.cp`:

```
#define __INSTALL_MENU_HOOK__
```

InstallHook

```
void InstallHook ();
```

Installs a hook into the menu defproc of the pop-up menu by calling `HookMenu`.

CalcDimensions

```
void CalcDimensions ();
```

Calculates the pane's size and other dimensions. If you called `CStdPopupPane` with `aWidth` set to `kAutoSize`, this makes the pane wide enough to fit the title, the longest menu item, and the Down Arrow. If you set `aHeight` to `kAutoSize`, this makes the pane at least `kStdPopupHeight` points high and big enough for the font.

MakePopupBox

```
void MakePopupBox ();
```

Creates a CPaneBorder for drawing the pop-up menu and its drop shadow, and sets `itsPopupBox` equal to it.

Appearance

InvertTitle

```
void InvertTitle ();
```

Inverts the pop-up title. Called when the user clicks the menu pane or presses a Command-key for a menu item.

SetTextFont

```
void SetTextFont (short aFont, short aSize);
```

Sets the pane's `textFont` and `textSize` data members and calls `CalcPopupBox`.

Member Functions: Private

CStdPopupPaneX

```
void CStdPopupPaneX ();
```

Performs common construction.

CStream 104

Introduction

CStream is a base class for streaming object I/O.

CStream allows you to read and write binary streams of objects and primitive data. CStream can read and write object data structures recursively, so an entire data structure can be read or written with a single program statement. CStream implements duplicate checking, allowing data structures to be self-referential or to have multiple pointers to the same object.

Through its derived classes, CHandleStream and CFileStream, CStream allows you to read and write resources and data files.

Heritage

Base Class	None
Derived Classes	CBufferedStream
	CCountingStream

Using CStream

See Chapter 8, “Using Object I/O,” to learn how to use streams.

Since CStream is an abstract base class, you never use it directly. (CStream-derived classes may be input- or output-only.)

Data Members

CStream defines these protected data members:

Data member	Type	Description
checkList	CvoidPtrArray*	List of pointers to objects written while duplicate checking is on
itsMode	TCLStreamMode	One of ClosedStream, ReadStream, WriteStream, ReadWriteStream

Member Functions

Creation and destruction

CStream

```
CStream (Boolean check = TRUE);
```

Constructor. If check is TRUE, duplicate checking is enabled.

~CStream

```
~CStream ();
```

Destructor. Delete checkList.

Open and close

Open

```
void Open (TCLStreamMode mode);
```

Opens a stream for ReadStream, WriteStream, or ReadWriteStream access.

Close

```
void Close ();
```

Closes a stream, returning it to ClosedStream mode. The same stream may be opened and closed multiple times.

GetMode

```
TCLStreamMode GetMode ();
```

Returns the current stream mode.

Positioning

Position

```
long Position ();
```

Pure virtual function that returns the current stream position. The stream position is a long integer ranging from zero to the number of bytes in the stream minus one. The first byte of the next `Get` or `Put` will come from or go to the current position.

MoveTo

```
void MoveTo (long newPosition);
```

Pure virtual function that changes the current stream position.

Size

```
long Size ();
```

Pure virtual function that returns the number of bytes in the stream.

Truncate

```
void Truncate (long newSize);
```

Pure virtual function that reduces the number of bytes in the stream.

AtEnd

```
Boolean AtEnd ();
```

Returns `TRUE` if the current position is the end of the stream. `AtEnd` is always `TRUE` for streams opened in `WriteStream` mode, and may be used to test streams opened in `ReadStream` mode for end-of-stream.

Writing

Put

```
void Put (const void *bytes, long n);
```

Pure virtual function that puts `n` bytes to the stream.

PutThru

```
void PutThru (const void *bytes, char c);
```

Pure virtual function that puts data to the stream starting with the first character of `bytes` and ending with the first occurrence of the character `c`.

PutStr

```
void PutStr (const char *string);
```

Puts a zero-byte delimited C string to the stream.

PutStr255

```
void PutStr255 (const unsigned char *string);
```

Puts a Pascal string to the stream (an unsigned length byte followed by up to 255 characters of data).

PutStr255AsLine

```
void PutStr255AsLine (const unsigned char *line);
```

Puts a Pascal string to the stream followed by an `'\r'` character.

PutChar

```
void PutChar (char value);
```

Puts a single character to the stream.

PutBoolean

```
void PutBoolean (Boolean value);
```

Puts a Boolean value to the stream.

PutShort

```
void PutShort (short value);
```

Puts a 16-bit short integer value to the stream.

PutInt

```
void PutInt (int value);
```

Puts an `int` to the stream. Depending on the setting of the 4-byte ints compiler option, an `int` may be a 16- or 32-bit value.

PutLong

```
void PutLong (long value);
```

Puts a 32-bit long integer value to the stream.

PutFloat

```
void PutFloat (float value);
```

Puts a 32-bit IEEE floating-point value to the stream.

PutDouble

```
void PutDouble (SHORT_DOUBLE value);
```

Puts a 64-bit IEEE floating-point value to the stream.

PutHandle

```
void PutHandle (Handle aHandle);
```

Puts the relocatable data pointed to by the `aHandle` argument to the stream. If `aHandle` is `NULL`, a special `NULL` indication is written to the stream.

PutPtr

```
void PutPtr (Ptr aPtr);
```

Puts the non-relocatable data pointed to by the `aPtr` argument to the stream. If `aPtr` is `NULL`, a special `NULL` indication is written to the stream.

Reading

Get

```
void Get (void *bytes, long n);
```

Pure virtual function that gets `n` bytes from the stream.

GetThru

```
void GetThru (void *bytes, char c);
```

Gets data from the stream starting with the first character of `bytes` and ending with the first occurrence of the character `c`.

GetThruN

```
long GetThruN (void *bytes, char c, long n);
```

Pure virtual function that gets data from the stream starting with the first character of `bytes` and ending with the first occurrence of the character `c`, the end-of-stream, or the first `n` characters—whichever is first.

`GetThruN`, unlike most Get functions, does not fail if it is called at end-of-stream. Instead, it returns a length of zero. `GetStrAsLine`, which calls `GetThruN`, also works like this.

GetStr

```
void GetStr (char *string);
```

Gets a zero-byte delimited C string from the stream.

GetStr255

```
void GetStr255 (unsigned char *string);
```

Gets a Pascal string from the stream (an unsigned length byte followed by up to 255 characters of data).

GetCString

```
void GetCString (CString& string);
```

Gets an unknown length CString class object safely from the stream.

GetLineAsStr255

```
void GetLineAsStr255 (unsigned char *line);
```

Gets a Pascal string from the stream, returning all characters up to the next `'\r'` character in the stream or the end-of-stream or 255 characters, whichever occurs first.

`GetLineAsStr255` can be called in a loop to retrieve successive lines from a text file. It returns a null string at end-of-stream. This can be distinguished from a true empty line by calling `AtEnd` before `GetLineAsStr255`.

GetChar

```
char GetChar ();
```

Gets a single character from the stream.

GetBoolean

```
Boolean GetBoolean ();
```

Gets a Boolean from the stream.

GetShort

```
short GetShort ();
```

Gets a 16-bit short integer value from the stream.

GetInt

```
int GetInt ();
```

Gets an `int` from the stream. Depending on the setting of the 4-byte ints compiler option, an `int` may be a 16- or 32-bit value.

GetLong

```
long GetLong ();
```

Gets a 32-bit long integer value from the stream.

GetFloat

```
float GetFloat ();
```

Gets a 32-bit IEEE floating-point value from the stream.

GetDouble

```
SHORT_DOUBLE GetDouble ();
```

Gets a 64-bit IEEE floating-point value from the stream.

GetHandle

```
Handle GetHandle ();
```

Creates a new relocatable block on the heap and fills it with data previously written to the stream with `PutHandle`. If a `NULL` handle was written, a `NULL` handle is returned by `GetHandle`.

GetPtr

```
Ptr GetPtr ();
```

Creates a new non-relocatable block on the heap and fills it with data previously written to the stream with `PutPtr`. If a `NULL` pointer was written, a `NULL` pointer is returned by `GetPtr`.

Reading views

GetView

```
CView *GetView (CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Calls `GetBureaucrat` for a CView object to be enclosed and supervised as specified by the arguments. `GetView` can be used to read an entire window and all its subviews previously saved to the stream by `PutObject`.

See Chapter 8, “Using Object I/O,” for a discussion of `GetView`.

GetBureaucrat

```
CBureaucrat *GetBureaucrat (CBureaucrat
   *aSupervisor);
```

Calls `GetObject` for a CBureaucrat object to be supervised as specified by the argument. `GetBureaucrat` can be used for other things, but its primary purpose is to be called by `GetView`.

Duplicate checking

CheckDuplicates

```
void CheckDuplicates (Boolean startChecking);
```

Turns duplicate checking on or off for the stream. When duplicate checking is on, all objects written by `PutObject` are remembered in a list. If `PutObject` is called to write the same object twice, a reference to the first occurrence of the object is written to the stream instead of writing the same object twice. This reference is not a C++ reference, but a long integer that uniquely identifies the object. Likewise, on input, when `GetObject` is called and encounters an object reference, it uses an auxiliary list to locate the first occurrence of the object and returns a pointer to that object instead of allocating memory to the same object twice.

`CheckDuplicates` should always be used when reading and writing views, because a view data structure has multiple pointers to the same objects.

Utility functions

PutReference

```
Boolean PutReference (void *anObject);
```

Writes a duplicate reference (a long integer uniquely identifying the object) to the stream. Called by `PutObject`. Returns `TRUE` if `anObject` has already been written to the stream and a reference was written. Returns `FALSE` if this is the first time `anObject` has been seen and no reference was written.

GetReference

```
void *GetReference ();
```

Reads an object from the stream that was identified by a duplicate reference. Called by `GetObject`.

AddReference

```
void AddReference (void *anObject);
```

Remembers that an object has been written to or read from a stream. Called by `GetObject` and `PutObject`.

GetClassName

```
void GetClassName (char *className,
   size_t maxStrLen);
```

Reads a class name from the stream of no more than `maxStrLen` characters. Called by `GetObject`.

PutClassName

```
void PutClassName (const char *className);
```

Remembers that an object has been written to or read from a stream. Called by `GetObject` and `PutObject`.

PutObjectReference

```
template<class T>
void PutObjectReference (CStream& stream,
                         T* anObject);
```

Puts a `long` that uniquely identifies the object pointed to by `anObject` to the stream. The unique identifier is a relative pointer to an object previously written to the stream. If the object has not been previously written to the stream, `PutObjectReference` writes a `NULL` pointer indication. Duplicate Checking must be on.

`PutObjectReference` is used in special cases where an object pointer may or may not point to an object in the data structure currently being written. See "PutTo" in Chapter 27, "CBureaucrat," for an example. You probably won't need to call this function.

Member Functions: Protected

The following member functions are provided to allow you to customize or extend the way objects are encoded in the stream or the way duplicate checking is performed. Most programs do not need to call them.

ObjectToReference

```
long ObjectToReference (CObject **anObject,
   short *flag);
```

Encodes a reference to an object if the object was previously written to the stream with `DuplicateChecking` on. `ObjectToReference` allows derived classes to customize reference encoding.

ReferenceToObject

```
CObject *ReferenceToObject (long index,
   short flag);
```

An object reference is encoded in the stream as a flag byte with a particular value followed by a long integer index to a previously read object. This function uses the index to look up the object in the list kept by duplicate checking. `ReferenceToObject` allows derived classes to extend the reference encoding scheme to address other needs. Only derived classes may call this function.

Utility Functions

The following function is not a member function:

FailStream

```
void FailStream (short why);
```

Calls `Failure`. The `why` argument is an index into the `'STR#' 29000` resource containing object I/O error message text.

Struct Macros

PutStruct

```
#define PutStruct (S) Put(&S, sizeof S)
```

Puts the `struct` argument to the stream. `PutStruct` is a convenience; that is, `PutStruct(rec)` can be used instead of `Put(rec, sizeof(rec))`. Warning: Streams written using the `PutStruct` macro may not be transportable between 68K family and PowerPC systems.

GetStruct

```
#define GetStruct (S) Get(&S, sizeof S)
```

Gets the struct argument from the stream. `GetStruct` is a convenience; that is, `GetStruct(rec)` can be used in place of `Get(rec, sizeof(rec))`.

Friend Functions

The following friend functions overload `operator <<` and `>>` to provide alternatives to the `Put` and `Get` functions:

operator <<

```
friend CStream& operator << (CStream& s,
                  CString& v);
```

Puts a CString object.

```
friend CStream& operator << (CStream& s,
                  const char *v);
```

Puts a C (null-terminated) string.

```
friend CStream& operator << (CStream& s,
                  const unsigned char *v);
```

Puts a Pascal (length byte plus data) string.

```
friend CStream& operator << (CStream& s,
                  char v);
```

Puts a `char`.

```
friend CStream& operator << (CStream& s,
                  Boolean v);
```

Puts a `Boolean`.

```
friend CStream& operator << (CStream& s,
                  short v);
```

Puts a `short`.

```
friend CStream& operator << (CStream& s,
                  int v);
```

Puts an `int`.

```
friend CStream& operator << (CStream& s,
                  long v);
```

Puts a `long`.

```
friend CStream& operator << (CStream& s,
                  float v);
```

Puts a `float` (32-bit IEEE value).

```
friend CStream& operator << (CStream& s,
                  double v);
```

Puts a `double` (64-bit IEEE value).

```
friend CStream& operator << (CStream& s,
                  Handle v);
```

Puts data pointed to by a `Handle`.

```
friend CStream& operator << (CStream& s,
                  Point& v);
```

Puts a `Point`.

```
friend CStream& operator << (CStream& s,
                  Rect& v);
```

Puts a `Rect`.

---

operator >>

```
friend CStream& operator >> (CStream& s,
                  CString& v);
```

Gets a CString object.

```
friend CStream& operator >> (CStream& s,
                  char *v);
```

Gets a C (null-terminated) string.

```
friend CStream& operator >> (CStream& s,
                  const unsigned char *v);
```

Gets a Pascal (length byte plus data) string.

```
friend CStream& operator >> (CStream& s,
                  char& v);
```

Gets a `char`.

```
friend CStream& operator >> (CStream& s,
                  Boolean& v);
```

Gets a `Boolean`.

```
friend CStream& operator >> (CStream& s,
                  short& v);
```

Gets a `short`.

```
friend CStream& operator >> (CStream& s,
                  int& v);
```

Gets an `int`.

```
friend CStream& operator >> (CStream& s,
                  long& v);
```

Gets a `long`.

```
friend CStream& operator >> (CStream& s,
                  float& v);
```

Gets a `float` (32-bit IEEE value).

```
friend CStream& operator >> (CStream& s,
                  double& v);
```

Gets a `double` (64-bit IEEE value).

```
friend CStream& operator >> (CStream& s,
                  Handle& v);
```

Gets data pointed to by a `Handle`.

```
friend CStream& operator >> (CStream& s,
                  Point& v);
```

Gets a `Point`.

```
friend CStream& operator >> (CStream& s,
                  Rect& v);
```

Gets a `Rect`.

Template Functions

The following template functions allow object I/O on objects of arbitrary class:

GetObject

```
template<class T>
CStream& GetObject (CStream& stream,
                    T*& anObject);
```

Gets an object from the stream and sets the `anObject` pointer to point to it. If the object written to the stream was `NULL`, `anObject` is set to `NULL`.

See Chapter 8, "Using Object I/O," for a discussion of `PutObject` and `GetObject`.

PutObject

```
template<class T>
CStream& PutObject (CStream& stream,
                    T* anObject);
```

Puts the object pointed to by `anObject` to the stream. If `anObject` is `NULL`, a special `NULL` indication is written to the stream.

PutObject1

```
template<class T>
void PutObject1(T* anObject, long stream);
```

A version of `PutObject` that can be used with, for example, `DoForEach1` to write all objects in a `CList`. You probably will not have to call `PutObject1`, as the `CList` class automatically reads and writes all objects in a list when the list is read or written.

CString 105

Introduction

CString contains functions for concatenating, copying, and extracting characters from CStrings. The class provides for coercion to and from the type `const char *`.

Heritage

Base Class	None
Derived Classes	None

Using CString

CString implements strings as C++ objects. It contains functions for concatenating and copying CStrings. Character extraction is supported by overloaded `operator[]`, which provides safe array indexing. Friend functions exist for equality and inequality comparisons.

CString provides for coercion to and from the type `const char *`. The two types are interchangeable as rvalues, and a CString can be constructed from a `const char *`.

Data Members

This data member is private:

Data member	Type	Description
`s`	`char *`	Internal representation of the CString as a zero-terminated C string

Member Functions

Creation and destruction

CString
```
CString (const char *str);
```
Constructor. Sets `s` to point to a copy of `str`. (The function allocates a new `char` array of size `strlen(str)+1`, sets `s` to point to that buffer, and copies `str` to `s`.)

CString
```
CString (const CString& x);
```
Copy constructor. Sets `s` to point to a copy of `x.s`. Equivalent to `CString(x.s)`.

CString
```
CString ();
```
Default constructor. Initializes `s` to the empty string, `""`.

~CString
```
~CString ();
```
Virtual destructor. Deletes the array `s`.

Member Functions: Private

Internal utility functions

CopyStr
```
CopyStr (const char *str);
```
Replaces `str` with a new copy of the string.

CatStr
```
CatStr (const char *str);
```
Concatinates `str` to `s`.

Assignment

operator=
```
CString& operator=(const char *str);
```
Deletes the array `s`, then sets `s` to point to a copy of `str`. Returns `*this`.

operator=
```
CString& operator=(const CString& x);
```
Deletes the array `s`, then sets `s` to point to a copy of `str`. Equivalent to `operator=(x.s)`. Returns `*this`.

Concatenation

operator+=
```
CString& operator+=(const char *str);
```
Appends `str` to `s`, setting `s` to point to a new buffer holding the result. The old array `s` is deleted. Does nothing if `str` is the empty string, because in that case `s` already holds the result of appending `str` to it. Returns `*this`.

operator+=
```
CString& operator+=(const CString& x);
```
Appends `x.s` to `s`, setting `s` to point to a new buffer holding the result. The old array `s` is deleted. Does nothing if `x.s` is the empty string. Equivalent to `operator+=(x.s)`. Returns `*this`.

Length

length
```
long length ();
```
Returns `strlen(s)`.

Extraction

operator[]
```
char& operator[](int i);
```
Returns `s[i]` if `i` is between `0` and `len`, where `len` equals `strlen(s)`. If `i` is less than `0`, returns `s[0]`; if `i` is greater than `len`, returns `s[len]`, which always has the value `'\0'`.

Conversion

operator const char*
```
operator const char*();
```
Returns `s`.

GetPStr

```
void GetPStr(unsigned char *pstr);
```

Converts `s` to a Pascal string, storing the result in `pstr`. The first byte of `pstr` holds the length of the string, and the succeeding bytes hold the characters of the string without a terminating zero byte. At most, 255 bytes of `s` are converted. `pstr` must be large enough to hold the result; that is, its size must be at least `length()+1`.

Friend Functions

operator==

```
int operator==(const CString& x,
                       const char *str);
```

Returns `1` if `x.s` and `str` are equivalent strings; returns `0` otherwise. Strings are equivalent if they have the same length and each of their characters is the same; their addresses may, of course, be different. Implemented as an inline that returns `strcmp(x.s, str) == 0`.

operator==

```
int operator==(const CString& x,
                       const CString& y);
```

Equivalent to `operator==(x, y.s)`.

operator==

```
int operator==(const char *str,
                       const CString& x);
```

Equivalent to `operator==(x, str)`.

operator!=

```
int operator!=(const CString& x,
                       const char *str);
```

Returns `1` if `x.s` and `str` are not equivalent strings; returns `0` if they are equivalent. Implemented as an inline that returns `strcmp(x.s, str) != 0`.

operator!=

```
int operator!=(const CString& x,
                       const CString& y);
```

Equivalent to `operator!=(x, y.s)`.

operator!=

```
int operator!=(const char *str,
                        const CString& x);
```

Equivalent to `operator!=(x, str)`.

CStyleTEClipboard 106

Introduction

CStyleTEClipboard is a subclass of CClipboard that is able to display styled text when both `TEXT` and `styl` data are in the global scrap.

Heritage

Base Class	CClipboard
Derived Classes	None

Using CStyleTEClipoard

To use this class in your application, you should override `CApplication::MakeClipboard` to set `gClipboard` to a CStyleTEClipboard.

For example:

```
void CMyApp::MakeClipboard() {
   CStyleTEClipboard*styleClip =
      new(CStyleTEClipboard);
      styleClip->IStyleTEClipboard(this, TRUE);
      gClipboard = styleClip;
   }
```

Data Members

CStyleTECClipboard defines no data members.

Member Functions

Creation and destruction

CStyleTEClipboard

```
CStyleTEClipboard(Boolean hasWindow = FALSE);
```

Constructor. `hasWindow` controls whether the Clipboard displays its contents in a window.

IStyleTEClipboard

```
void  IStyleTEClipboard(CApplication
   *aSupervisor, Boolean hasWindow);
```

Initializes a CStyleTEClipboard. Provided for backward compatability. `IStyleTEClipboard` should only be called if no arguments were passed to the constructor.

Display

MakeClipView

```
virtual CPanorama *MakeClipView(long dataType,
    Handle dataHandle);
```

Makes a panorama to display the current Clipboard data. If both `'TEXT'` and `'styl'` data are available, it makes a CStyleText panorama, otherwise it calls `CClipboard::MakeClipView` to make the view.

CStyleTEEditTask 107

Introduction

CStyleTEEditTask provides undo and redo support for typing and editing in CStyleText panes.

Heritage

Base Class	CTextEditTask
Derived Classes	None

Using CStyleTEEditTask

CStyleTEEditTask implements undo for typing and the **Cut**, **Copy**, **Paste**, and **Clear** commands in CStyleTEEditTask. A CStyleText pane automatically creates an instance of this class when you type, press Backspace or Forward Delete, or choose an editing command.

Data Members

CStyleTEEditTask defines these protected data members:

Data member	Type	Description
`macTE`	`TEHandle`	`TEHandle` of `itsTextPane`
`origStyleScrap`	`StScrpHandle`	Original styles on Clipboard
`deletedStyles`	`StScrpHandle`	Styles of original text
`insertedStyles`	`StScrpHandle`	Styles of inserted text

Member Functions

Creation and destruction

CStyleTEEditTask

```
CStyleTEEditTask();
```

Default constuctor. Implicitly called when object is created by `new_by_name`. May be used in combination with `IStyleTEEditTask` for backward compatibility.

CStyleTEEditTask

```
CStyleTEEditTask(CStyleText *aTextPane,
   long anEditCmd, short firstTaskIndex);
```

Creates and initializes a CStyleTEEditTask object. `anEditCmd` should be the command that the task is to respond to, for example, `cmdCut`, `cmdCopy`, `cmdPaste`, or `cmdClear`. If the task is for typing, then it should be `cmdNull`. `FirstTaskIndex` is the index into the `strUNDO 'STR#'`, where the text undo labels are located. These typically are "Typing," "Cut," "Copy," "Paste," "Clear," and "Formatting." If there is any `styl` data on the Clipboard, this function saves it.

~CStyleTEEditTask

```
~CStyleTEEditTask();
```

Destructor. Frees the memory used by `origStyleScrap`, `deletedStyles`, and `insertedStyles`.

IStyleTEEditTask

```
void IStyleTEEditTask(CStyleText *aTextPane,
   long anEditCmd, short firstTaskIndex);
```

Initializes a CStyleTEEditTask. Provided for backward compatability. Should not be called if arguments were passed to the constructor.

Member Functions: Protected

DoBackspace

```
void DoBackspace();
```

Processes a backspace over original (not newly typed) text. This function verifies that the style of the character about to be deleted is different from the style of the character at the start of the saved styled scrap, then it saves it if necessary.

DoPwdDelete

```
void DoFwdDelete();
```

Processes a forward delete over original text. This function verifies that the style of the character about to be deleted is different from the style of the character at the end of the saved styled scrap, then it saves it if necessary.

SaveRange

```
void SaveRange(tRangeSelector whichRange);
```

Saves the selected text before it is deleted, or the inserted text before an undo is performed. This function calls `CTextEditTask::SaveRange` to save the text, then saves the styled scrap data.

RestoreRange

```
void RestoreRange(tRangeSelector whichRange,
   Boolean killData);
```

Restores the previously saved range of either inserted or deleted text. This function does not call `CTextEditTask::RestoreRange`.

StoreToClip

```
void StoreToClip(tClipSelector whichClip);
```

Stores the correct text and style data to the Clipboard after an **Undo** or **Redo** command is processed. If `whichClip` equals `kOldClip`, the original text and style data are restored (undo). If `whichClip` equals `KNewClip`, the deleted text and style data are restored.

CheckNewStyle

```
Boolean CheckNewStyle(ScrpSTElement *scrapEl,
   StScrpHandle styleH, Boolean atStart);
```

Compares a `ScrpSTEElement` with either the first or the last element in `styleH`. If they are the same, it returns `FALSE`. If they are different, the element is inserted in `styleH` and the function returns `TRUE`.

CStyleTEStyleTask 108

Introduction

CStyleTEStyleTask provides undo and redo support for style commands in CStyleText panes.

Heritage

Base Class	CTextStyleTask
Derived Classes	None

Using CStyleTEStyleTask

CStyleTEStyleTask stores the information needed to undo font, size, style, alignment, and spacing commands in CStyleText and its subclasses. A CStyleText pane automatically creates an instance of this class when you choose a style command.

Data Members

CStyleTEStyleTask defines these protected data members:

Data members	Type	Description
`oldStyles`	`StScrpHandle`	Handle to old styles
`macTE`	`TEHandle`	Handle to Styled Text Edit record
`selStart`	`long`	Beginning of the selection
`selEnd`	`long`	End of the selection

Member Functions

Creation and destruction

CStyleTEStyleTask

```
CStyleTEStyleTask();
```

Default constuctor. Implicitly called when an object is created by `new_by_name`. May be used in combination with `IStyleTEStyleTask` for backward compatibility.

CStyleTEStyleTask

```
CStyleTEStyleTask(CStyleText *aTextPane,
   long aStyleCmd, short firstTaskIndex);
```

Constructor. Creates a CStyleTEStyleTask. `aStyleCmd` should be the formatting command that the task is to respond to. `FirstTaskIndex` is the index into the `strUNDO 'STR#'`, where the text undo labels are located. These typically are “Typing,” “Cut,” “Copy,” “Paste,” “Clear,” and “Formatting.”

~CStyleTEStyleTask

```
~CStyleTEStyleTask();
```

Destructor.

IStyleTEStyleTask

```
void IStyleTEStyleTask(CStyleText *aTextPane,
   long aStyleCmd, short firstTaskIndex);
```

Initializes a CStyleTEStyleTask. Provided for backward compatability. Should not be called if the constructor was passed arguments.

Undo

```
void Undo();
```

Saves the style information for the currently selected range of text and restores the previously saved formatting. This method handles both **Undo** and **Redo**.

Member Functions: Protected

SaveStyle

```
void SaveStyle();
```

Saves the style(s) of the currently selected range of text.

RestoreStyle

```
void RestoreStyle();
```

Restores the previously saved formatting.

CStyleText 109

Introduction

CStyleText implements a pane that displays text with one or more styles applied to it. This class uses the Styled Text Edit routines in the Macintosh Toolbox.

Heritage

Base Class	CEditText
Derived Classes	None

Using CStyleText

Use CStyleText to display text with more than one style. If the entire range of text will always be displayed with the same formatting (font, size, style, and alignment), you should use a CEditText pane. You only need to create a CStyleText pane if you need to have different formats for different ranges of the text.

A CStyleText pane usually is the panorama in a scroll pane, so you can scroll through the text. The DoCommand function of a CStyleText pane handles all the common text-editing commands such as cutting and pasting, font selection, line spacing, and so on. The `Specify` function, inherited from CAbstractText, lets you choose whether or not the user can edit and copy your pane's text.

Many of the functions of this class operate on the current selection. For example, `SetSize` changes the font size for all text within the current selection. If you want to change the formatting for a specific range of text, select the range with `SetSelection` and format that range. To do this without changing the text within the current selection, you should call `GetSelection` to identify the current selection, select the range of text you want, and format it. Then reset the selection to the original range.

To make the pane respond to typing and commands, you must place it in the chain of command. The best way to do this is to set the value of your document's `itsGopher` data member to the CStyleText pane.

CStyleText uses the Styled Text Edit routines in the Macintosh Toolbox. These routines are designed to edit small amounts of text. The maximum number of characters you can store in a CEditText record is about 32,000, but performance degradation will occur long before the text record gets that large.

If you need to display more than 32,000 characters with multiple formats, you should create a subclass of CAbstractText.

Data Members

CStyleText defines no data members.

Member Functions

Creation and destruction

CStyleText();

```
CStyleText();
```

Default constructor. It is called implicitly when one object is created by `new_by_name`. It may be used in combination with `IStyleText` for backward compatibility.

CStyleText

```
CStyleText(CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizELASTIC,
   SizingOption aVSizing = sizELASTIC,
   short aLineWidth = -1,
   Boolean aScrollHoriz = 0); FALSE
```

Constructor. Most of the arguments are identical to the CPanorama constructor. `aLineWidth` specifies the width of the lines. If `aLineWidth` is less than 0 (zero), the width is the same as the Macintosh TextEdit record's viewRect. `aScrollHoriz` is `TRUE` if the text scrolls horizontally when the line width is greater than the frame width and the cursor moves outside the frame.

IStyleText

```
void IStyleText(CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initializes a CStyleText pane. This member function is provided for backward compatibility. You should not call this function if the constructor was passed arguments.

IViewTemp

```
void IViewTemp(CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

Initializes a CStyleText from a resource template.

MakeMacTE

```
void MakeMacTE();
```

Creates a styled TextEdit record that is stored in the inherited data member `macTE`.

Commands

PerformEditCommand

```
void PerformEditCommand(long theCommand);
```

Processes edit commands (`cmdCut`, `cmdCopy`, `cmdPaste`, `cmdClear`) for CStyleTEEditTasks.

This function ensures that both `TEXT` and `styl` data are transferred during editing operations.

CStyleTEEditTask and CStyleTEStyleTask call this function to undo an edit command.

Display

SetScrollPane

```
void SetScrollPane(CScrollPane *aScrollPane);
```

Ensures that reasonable scroll steps are set up when a CStyleText pane is installed in a scroll pane.

Text specification

InsertWithStyles

```
void InsertWithStyles(Ptr text, long length,
   StScrpHandle styles);
```

Inserts the text, with associated style information, into the selection.

Text characteristics

SetFontNumber

```
void SetFontNumber(short aFontNumber);
```

Sets the font for the current selection by font number.

SetFontSize

```
void SetFontSize(short aSize);
```

Sets the font size for the current selection.

SetFontStyle

```
void SetFontStyle(short aStyle);
```

Sets the font style of the current selection. `aStyle` may be one or any additive combination of `normal`, `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, or `extended`; for example, `bold + italic`.

SetStyle

```
void SetStyle(short mode, TextStyle *newStyle,
   Boolean redraw);
```

Sets the style of the selection (or insertion point) to the given style. Possible values for `mode` and `newStyle` are described in *Inside Macintosh Volume V* and *VI*.

SetSpacingCmd

```
void SetSpacingCmd(long aSpacingCmd);
```

Sets the space between lines of text. CStyleText does not support variable spacing, so `aSpacingCmd` is ignored and the spacing is always set to `cmdSingleSpace`.

GetTheStyleScrap

```
StScrpHandle GetTheStyleScrap();
```

Returns a `StScrpHandle` that contains style information for the selection.

SetTheStyleScrap

```
void SetTheStyleScrap(long rangeStart,
   long rangeEnd, StScrpHandle styleScrap,
   Boolean redraw);
```

Applies a `StScrpHandle` to the given range.

Member Functions: Protected

IStyleTextX

```
void IStyleTextX();
```

Performs common initialization.

MakeEditTask

```
CTextEditTask *MakeEditTask(long editCmd);
```

Creates a task to handle typing or edit menu commands. The task must be derived from CTextEditTask. The default function returns a pointer to the CStyleTEEditTask.

MakeStyleTask

```
CTextStyleTask *MakeStyleTask(long styleCmd);
```

Creates a task for carrying out font, style, size, alignment, and spacing commands. The task must be a derived class of CTextStyleTask. The default function returns a CStyleTEStyleTask.

CSubviewDisplayer 110

Introduction

CSubviewDisplayer displays a named subview.

Heritage

Base Class	CPane
Derived Classes	None

Using CSubviewDisplayer

Visual Architect creates subviews as separate `'CVue'` resources. To show a subview within another view, use the Subview tool  to create a CSubviewDisplayer pane and set the Subview to name the subview.

The CSubviewDisplayer pane encloses the subview. When the subview is drawn, it is clipped to the dimensions of the CSubviewDisplayer.

If the `filled` data member of CSubviewDisplayer is `TRUE`, the subview is displayed in the pane. If it is `FALSE`, the name of the subview is drawn in the pane, but the subview itself is not displayed.

You can also create a CSubviewDisplayer object at run-time, that is, from a window's director:

```
CSubviewDisplayer *disp =
   new CSubviewDisplayer("\pSubname",
         itsWindow, this, 100, 50, 10, 10);
```

If the subview resource name is not an empty string, CSubviewDisplayer calls `Fill` to display the subview in the pane.

Data Members

CSubviewDisplayer defines two data members:

Data member	Type	Description
`resName`	`Str31`	The name of the subview `'CVue'` resource created by Visual Architect.
`filled`	`Boolean`	TRUE if currently displaying a subview.

Member Functions

Creation and destruction

CSubviewDisplayer

```
CSubviewDisplayer (Str31 resName,
   CView *anEnclosure, CBureaucrat *aSupervisor,
   short aWidth = 0, short aHeight = 0,
   short aHEncl = 0, short aVEncl = 0,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY);
```

Constructor. `resName` is the name of the `'CVue'` resource to be displayed in the CSubviewDisplayer pane. If `resName` is a null string, no subview is displayed. The remaining arguments are the same as those of the CPane constructor, see Chapter 71, “CPane.”

CSubviewDisplayer

```
CSubviewDisplayer ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`.

Accessing

SetViewName

```
void SetViewName(Str31 viewName);
```

Sets the `resName` of the `'CVue'` resource to be displayed.

GetViewName

```
void GetViewName(Str31 viewName);
```

Returns the `resName` of the `'CVue'` resource currently displayed.

Fill

```
void Fill();
```

Calls `Empty` to delete any existing subviews, loads the subview using the current `resName`, and sets the `filled` data member to `TRUE`.

Empty

```
void Empty();
```

Sets the `filled` data member to `FALSE` and deletes any existing subviews.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the scroll bar. If the scroll bar is active, this function draws it in the normal way; if the scroll bar is inactive, it draws only the frame of the scroll bar.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CSwissArmyButton 111

Introduction

CSwissArmyButton is a base class for push buttons, check boxes, and radio buttons of all kinds.

The `buttonKind` data member determines what kind of button a CSwissArmyButton is. The `hiliteStyle` member tells how the button is highlighted; the two standard choices are inverting the button or displaying with a different graphic representation. The `onStyle` member tells how the "on" state of the button is graphically distinguished from the "off" state; the two standard choices are surrounding the button with a border or switching to a different graphical state. In all, up to four different graphical representations are possible, as determined by the values of the `value` and `hiliteStyle` data members.

Button groups are implemented by the CGroupButton base class. Each button has a group ID. All buttons with the same nonzero group ID are members of the same button group. When a radio button is turned on, all other radio buttons and check boxes in the same group are turned off. When a check box is checked, other check boxes are unaffected but all radio buttons in the same group are turned off. Button groups are confined to a single window.

These rules make it possible to construct button groups such as:

Figure 111-1 Example Button Group

If any check box is checked, the radio button will be turned off. If the radio button is turned on, all check boxes will be unchecked.

Heritage

Base Class	CPane
Derived Classes	CGroupButton
	CLine
	CPictureButton
	CShapeButton

Using CSwissArmyButton

CSwissArmyButton is used through its derived classes. You can also use it as a base class for new button types of your own design.

Data Members

CSwissArmyButton defines these protected data members:

Data member	Type	Description
`clickCmd`	`long`	Command sent on good click.
`value`	`short`	Control value, 0 or 1 for standard buttons.
`buttonKind`	`char`	One of `kSAPushButton`, `kSACheckBox`, or `kSARadioButton`.
`hiliteStyle`	`char`	One of `kSAStateHilight` or `kSADimHilight`.
`onStyle`	`char`	One of `kSAStateOn` or `kSABorderOn`.
`needsUpdate`	`Boolean`	`TRUE` if this button needs to go through entire window `Update` to redraw correctly.
`hilite`	`Boolean`	`TRUE` if button is currently highlighted.
`borderPen`	`Point`	Saved value of border pen size.
`grayLine`	`Boolean`	`TRUE` if button outline is gray.

Member Functions

Creation and destruction

CSwissArmyButton

```
CSwissArmyButton (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizFIXEDSTICKY,
   SizingOption aVSizing = sizFIXEDSTICKY,
   short aButtonKind = kSAPushButton,
   short aHiliteStyle = kSAStateHilight,
   short anOnStyle = kSAStateOn,
   Boolean aNeedsUpdate = kSANoUpdate);
```

Constructor. The `anEnclosure`, `aSupervisor`, `aHEncl`, `aVEncl`, `aHSizing`, and `aVSizing` arguments have their standard CPane meaning. Specify `aButtonKind` as one of `kSAPushButton`, `kSACheckBox`, or `kSARadioButton`; `aHighliteStyle` as one of `kSAStateHilight` or `kSADimHilight`; and `anOnStyle` as one of `kSAStateOn` or `kSABorderOn`. Set `aNeedsUpdate` to `TRUE` if the shape is overlapped by another graphic.

CSwissArmyButton

```
CSwissArmyButton ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `ISwissArmyButton` for backward compatibility.

Drawing

DrawButton

```
void DrawButton (Boolean hilite);
```

Sets `hilite`, then calls `Update` or `DrawAll` depending on the value of `needsUpdate`.

CalcDrawState

```
short CalcDrawState (Boolean hilite);
```

Calculates the state of the button as an index into a four-element state array that may hold 1, 2, or 4 values.

Tracking

DoClick

```
void DoClick (Point hitPt,
   short modifierKeys,
   long when);
```

Calls `Track` to track the mouse, then sets the value based on the final mouse position and the button type.

Track

```
Boolean Track ();
```

Tracks the mouse and calls `DrawButton` as needed. Returns `TRUE` if mouse is still inside the button at the end of tracking.

InButton

```
Boolean InButton (Point aPoint);
```

Returns `TRUE` if `aPoint` is inside the button. Derived classes must override this function.

FixupBorder

```
void FixupBorder ();
```

Creates a border, if `onStyle` is `kSABorderOn`. If the button is off, sets the border pen size to zero. If on, sets to saved pen size. If you want to change the default border size or other attributes, call `GetBorder` after initializing the button and change its attributes directly.

Accessing

SetClickCmd

```
void SetClickCmd (long aClickCmd);
```

Sets the command to be sent on a good click.

GetClickCmd

```
long GetClickCmd ();
```

Gets the command.

SetValue

```
void SetValue (short aValue);
```

Sets the `value` of the button. Does a `BroadcastChange` with reason code of `controlValueChanged`, just as CControl does.

Setting the `value` of a radio button or check box to 1 affects the values of other buttons in the group, as described in "Using CSwissArmyButton" earlier in this chapter.

GetValue

```
short GetValue ();
```

Gets the current `value` of the button.

SetButtonKind

```
void SetButtonKind (short aKind);
```

Sets the button `action` — one of `kSAPushButton`, `kSACheckBox`, or `kSARadioButton`.

GetButtonKind

```
short GetButtonKind ();
```

Gets the button action.

SetGrayLine

```
void SetGrayLine (Boolean isGray);
```

Sets the `grayLine` data member. If `TRUE`, the frame of the button is drawn in gray.

GetGrayLine

```
Boolean GetGrayLine ();
```

Gets the value of `grayLine`.

Button groups

TurnOff

```
void TurnOff ();
```

Called to turn the button off when another button in the group is turned on. Affects only check boxes and radio buttons.

IsRadioButton

```
Boolean IsRadioButton ();
```

Returns `TRUE` if button is a radio button.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes the button to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads the button from the stream.

Member Functions: Protected

ISwissArmyButton

```
void ISwissArmyButton (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aButtonKind, short aHiliteStyle,
   short anOnStyle, Boolean aNeedsUpdate);
```

Initialization function for derived classes that need to calculate certain constructor arguments. Arguments are the same as for the CSwissArmyButton constructor.

Member Functions: Private

CSwissArmyButtonX

```
void CSwissArmyButtonX ();
```

Performs common construction.

ISwissArmyButtonX

```
void ISwissArmyButtonX (short aButtonKind,
   short aHiliteStyle, short anOnStyle,
   Boolean aNeedsUpdate);
```

Performs common initialization.

CSwitchboard 112

Introduction

CSwitchboard is the class that processes Macintosh Toolbox events and transforms them into calls to the appropriate member functions of THINK Class Library objects. There is normally only one instance of this class.

Heritage

Base Class	None
Derived Classes	None

Using CSwitchboard

The single instance of this class, the switchboard, handles all the Macintosh Toolbox events and calls objects' member functions. The application's `Run` function repeatedly calls the switchboard's `ProcessEvent` function to dispatch events to the objects that make up your application.

The CApplication constructor calls `MakeSwitchboard` to create the single instance of CSwitchboard. The switchboard is stored in the application's `itsSwitchboard` data member. If you need to derive a class from CSwitchboard, you should override `MakeSwitchboard` in your application class.

Note

For example, you need to derive a class from CSwitchboard if your application handles `app1Evt`, `app2Evt`, or `app3Evt` events. Then you should override the `DoOtherEvent` function.

Data Member

CSwitchboard defines these protected data members:

Data Member	Type	Description
`mouseRgn`	`RgnHandle`	Argument for `WaitNextEvent` to handle cursor adjustment

Member Function

Creation and destruction

CSwitchboard

```
CSwitchboard ();
```

Constructor. The application's `MakeSwitchboard` function creates the switchboard. This function also installs one handler for all Apple events.

ISwitchboard

```
void ISwitchboard ();
```

Initialization function provided for backward compatibility. Does nothing.

Mouse

DoMouseDown

```
void DoMouseDown (EventRecord *macEvent);
```

Called when the user presses the mouse button. This function calls the desktop's `DispatchClick` function and stores the event in the global `gLastMouseDown`.

DoMouseUp

```
void DoMouseUp (EventRecord *macEvent);
```

Called when the user releases the mouse button. This function calls the `DoMouseUp` member function of the view in which a mouse down most recently occurred. Since a mouse up always follows a mouse down, it's important that the event be dispatched to the same object that handled the mouse down, rather than to the view in which the mouse up occurred. This function stores the event in the global `gLastMouseUp`.

Key

DoKeyEvent

```
void DoKeyEvent (EventRecord *macEvent);
```

Called when the user presses or releases a key. If the user holds down the Command key and presses another key at the same time, this function uses the Toolbox routine `MenuKey` to find the menu equivalent. If there is a menu equivalent, this function calls the gopher's `DoCommand` function; if there is no menu equivalent, it calls the gopher's `DoKeyDown` function. For other key events, this function calls the `DoKeyDown`, `DoKeyUp`, or `DoAutoKey` function of the gopher.

Disk

DoDiskEvent

```
void DoDiskEvent (EventRecord *macEvent);
```

Calls the Toolbox routine `DIBadMount` to mount a disk. This is the only event that the switchboard handles directly.

Window event

DoUpdate

```
void DoUpdate (EventRecord *macEvent);
```

Calls the `Update` function of the window specified in the event record.

DoActivate

```
void DoActivate (EventRecord *macEvent);
```

Calls the `Activate` function of the window specified in the event record.

DoDeactivate

```
void DoDeactivate (EventRecord *macEvent);
```

Calls the `Deactivate` function of the window specified in the event record.

Suspend/Resume

DoSuspend

```
void DoSuspend (EventRecord *macEvent);
```

Called when the application is about to be switched to the background under MultiFinder or System 7. This function calls your application's `Suspend` function.

DoResume

```
void DoResume (EventRecord *macEvent);
```

Called when the application is about to be switched to the foreground under MultiFinder or System 7. This function calls your application's `Resume` function.

Event processing

DoOtherEvent

```
void DoOtherEvent (EventRecord *macEvent);
```

If your application handles `app1Evt`, `app2Evt`, or `app3Evt` events, override this function. The default function does nothing.

DoIdle

```
void DoIdle (EventRecord *macEvent);
```

Invoked during null events. `DoIdle` calls the application's `Idle` function. The application uses `Idle` to perform periodic tasks.

DoHighLevelEvent

```
void DoHighLevelEvent (const EventRecord*
   macEvent);
```

Handles a high-level event. This function assumes all high-level events are Apple events. If you use a high-level event that isn't an Apple event, you must override this function. Your function should call the inherited `DoHighLevelEvent` to handle Apple events.

AppleEventIdle

```
Boolean AppleEventIdle (EventRecord *macEvent,
   long *sleepTime, RgnHandle *mouseRgn);
```

The default Apple event idle procedure, `AppleEventIdleProc`, calls this function. The Toolbox functions, `AEInteractWithUser` and `AESend`, use the idle procedure to send events while waiting for the user to respond to an Apple event, or while the application awaits activation. The possible events are null, update, OS, or

activate events. If the event is a null event, the desired sleep time and mouse region should be returned in the corresponding arguments.

To use a different idle procedure, set the `idleProc` data member in the CAppleEvent object, described in Chapter 15, "CAppleEvent."

This function returns `TRUE` if the user aborted by pressing Command-. (Command-Period), and `FALSE` if the user wants to continue waiting.

ProcessEvent

```
void ProcessEvent ();
```

This function is the heart of your application's event loop. It gets an event and calls an appropriate function of the switchboard to handle it. Before processing the event, `ProcessEvent` calls the application's `DispatchCursor` function to adjust the cursor.

GetAnEvent

```
Boolean GetAnEvent (EventRecord *macEvent);
```

Gets the next event in the event queue. This function calls one of the Toolbox routines `GetNextEvent` or `WaitNextEvent` to fetch an event, and returns the result. If you need to process an event before the switchboard handles it, override this function. Your function should call the inherited `GetAnEvent` before performing any other processing.

DispatchEvent

```
void DispatchEvent (EventRecord *macEvent)
```

This function is the main event dispatcher. It may handle the event by calling itself recursively with another event argument.

CTable 113

Introduction

CTable is an abstract class for displaying information in a row and column format.

Heritage

Base Class	CPanorama
Derived Classes	CArrayPane

Using CTable

CTable is an abstract class that lets you display data in a tabular format—such as a scrolling list or spreadsheet—with up to 32,767 rows and columns. It determines which cells need to be drawn and highlighted. It also handles mouse selection, scrolling, and cursor key navigation. To use CTable, you must create a derived class that provides storage for your data and draws the contents of a cell. Figure 113-1 shows a derived class of CTable that adds support for column and row headings. It also demonstrates discontinuous selection, which CTable handles for you.

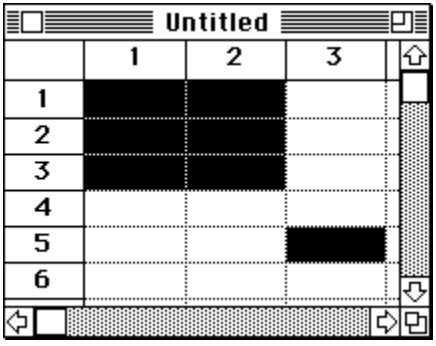


Figure 113-1 A CTable Enclosed in a CScrollPane

CTable draws each cell individually with the function `DrawCell`. The default `DrawCell` calls `GetCellText`, which returns a string to draw in the cell. If your data can be represented as a string, your derived class should override `GetCellText`. If your data is represented with graphics, such as an icon or a picture, your derived class must override `DrawCell`.

By default, CTable does not clip drawing to a cell, so there is no protection from drawing across cell boundaries. You can enable clipping by setting the `clipToCells` data member to `TRUE` in your derived class constructor. However, clipping to a cell will slow down drawing.

CTable specifies cells with the List Manager `Cell` type, which is identical to the QuickDraw `Point` type. Because cells are identical to points, and regions are collections of points, CTable can use regions to store lists of cells.

CTable uses zero-based numbering for rows and columns. The data member `tableBounds` is a rectangle indicating the cells contained in the table. If the table contains 4 rows and 5 columns, `tableBounds` would be `{0,0,4,5}`. In contrast, the inherited data member `bounds` is the number of pixels spanned by the table. The data members `itsRows` and `itsCols` are CRunArray objects that give the sizes of each row and column. The data member `itsSelection` is a `RgnHandle` that holds the selection.

Mouse interaction is handled by CTableDragger, a derived class of CMouseTask, described in Chapter 114, “CTableDragger.” You can control how your derived class handles selection with the `SetSelectionFlags` function. Also, you can handle clicks with your own mouse task by overriding the `MakeMouseTask` function.

Your derived class may draw row and column borders. Using the `SetRowBorders` and `SetColBorders` functions, you can specify a separate thickness, pen mode, and pattern for each.

CTable uses the dependency mechanism to notify its supervisor and dependents when the selection has changed. If an object (other than the table’s supervisor) needs to keep track of your table’s selection, call the object’s `DependUpon` function with the table as the provider. That object’s class should override the `ProviderChanged` function and watch for the reason code `tableSelectionChanged`.

Data Members

CTable defines the following protected data members:

Data member	Type	Description
tableBounds	Rect	Table size in cells
topLeftIndent	Point	Top and left indent of entire table
itsSelection	RgnHandle	The current selection
itsRows	CRunArray*	Runs of row heights
itsCols	CRunArray*	Runs of column widths
drawOrder	tTblDrawOrder	Whether CTable draws in row-major or column-major order
defRowHeight	short	Default height of rows
defColWidth	short	Default width of columns
selectionFlags	long	Flags for selection behavior
fontInfo	FontInfo	Font information for text tables
indent	Point	Amount to indent drawing in cell
dblClickCmd	long	Command sent if user double-clicks cell
drawActiveBorder	Boolean	If TRUE, draws a border around the table when the pane is active
clipToCells	Boolean	If TRUE, clips drawing to cell
rowBorders	tTableBorder	Structure that contains thickness, pen mode, and pattern for row borders

Data member	Type	Description
colBorders	tTableBorder	Structure that contains thickness, pen mode, and pattern for column borders
dClip	Rect	Clip rectangle in QuickDraw coordinates
saveBorder	CPaneBorder*	Border around scrollpane prior to table becoming gopher; NULL if none

CTable defines three static data members:

Data member	Type	Description
cDeselection	RgnHandle	Region containing cells to be deselected
cNewSelection	RgnHandle	Region containing cells to be newly selected
cCurrHilite	RgnHandle	Region containing cells that are currently highlighted

Member Functions

Creation and destruction

CTable

```
CTable (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing = sizELASTIC,
   SizingOption aVSizing = sizELASTIC);
```

Constructor. All the arguments are identical to those of the CPanorama constructor, described In Chapter 74, “CPanorama.”

CTable

```
CTable ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `ITable` for backward compatibility.

~CTable

```
~CTable ();
```

Destructor. Deletes the memory for the selection and deletes the memory for the arrays that store the heights and widths of the table's rows and columns.

ITable

```
void ITable (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialization function compatible with previous release. May not be called if constructor has arguments.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This function is used internally for initializing from a resource template. Each derived class of CView overrides this function to use its own resource template.

Accessing

SetDefaults

```
void SetDefaults (short colWidth,
   short rowHeight);
```

Sets the default row and column size to `colWidth` and `rowHeight`. If a new value is negative, its corresponding default is not changed.

CTable uses these values as the scroll pane steps and overlaps. In an empty table, it also uses them as the size for a new row or column. In a table that isn't empty, it uses the size of the preceding row or column as the size for a new one.

SetDrawOrder

```
void SetDrawOrder (tTblDrawOrder aDrawOrder);
```

Sets the order in which `Draw` draws the table cells: row-major order (row by row) or column-major order (column by column). If you use row-major order, `Draw` calls `DrawRow` for each row in the table. If you use column-major order, `Draw` calls `DrawCol` for each column in the table.

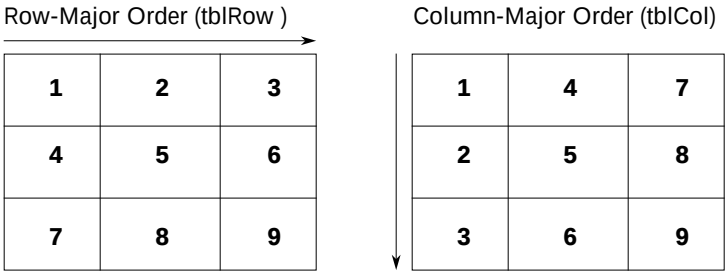


Figure 113-2 Drawing in row-major order and column-major order

Use one of these values for `aDrawOrder`:

If aDrawOrder is…	The table is drawn in…
`tblRow`	Row-major order
`tblCol`	Column-major order

A table that displays a database might use row-major order and overrides `DrawRow` to set up the current record for each row. A table that displays vectors of numbers as columns might use column-major order and override `DrawCol` to set up the current vector for each column.

SetScrollPane

```
void SetScrollPane (CScrollPane *aScrollPane);
```

Creates a scroll pane, sets `itsScrollPane` to it, initializes the step, and overlaps sizes. This function provides reasonable default step and overlaps sizes. Override it only if you need to set the sizes differently.

SetSelectionFlags

```
void SetSelectionFlags (long selFlags);
```

Lets you choose how cells are selected. It sets selectionFlags to selFlags. And together any of the following flags create a value for selFlags:

Flag	Meaning
`selOnlyOne`	Only one cell is selected at a time
`selNoDisjoint`	Disjoint selection is not allowed
`selExtendDrag`	Extend selection by adding cell dragged, not by extending rectangle
`selDragRects`	Dragging always selects rectangle

GetSelectionFlags

```
long GetSelectionFlags ();
```

Returns the selection flags.

SetDblClickCmd

```
void SetDblClickCmd (long aCmd);
```

Sets `dblClickCmd` to `aCmd`. The `dblClickCmd` command is sent to a table pane by `DoDblClick` when you double-click in a cell.

GetSelection

```
RgnHandle GetSelection ();
```

Returns the region that specifies the selected cells.

IsSelected

```
Boolean IsSelected (Cell aCell);
```

Returns `TRUE` if `aCell` is selected; `FALSE` otherwise.

GetTableBounds

```
void GetTableBounds (Rect *aTableBounds);
```

Sets `aTableBounds` to the size of the table in cells.

GetRowCount

```
short GetRowCount ();
```

Returns the number of rows in the table.

GetColCount

```
short GetColCount ();
```

Returns the number of columns in the table.

SetRowHeight

```
void SetRowHeight (short rowNum,
   short newHeight);
```

Sets the height of `rowNum` and redraws the table.

GetRowHeight

```
short GetRowHeight (short rowNum);
```

Returns the height of row `rowNum`, if `rowNum` is a row in the table; returns `-1` otherwise.

SetColWidth

```
void SetColWidth (short colNum, short newWidth);
```

Sets the width of `colNum` and redraws the table.

GetColWidth

```
short GetColWidth (short colNum);
```

Returns the width of column `colNum`, if `colNum` is a column in the table; returns `-1` otherwise.

GetRowStart

```
long GetRowStart (short rowNum);
```

Returns the pixel position of the top of `rowNum` in pane coordinates. Returns the start of the last row if the row isn't in the table.

GetColStart

```
long GetColStart (short colNum);
```

Returns the pixel position of the left side of `colNum` in pane coordinates. Returns the start of the last column if the column isn't in the table.

GetCellRect

```
void GetCellRect (Cell theCell,
   LongRect *cellRect);
```

Returns the pixels bounded by `theCell`. It sets `cellRect` to the bounds of `theCell`.

Drawing

Draw

```
void Draw (Rect *area);
```

Draws the table pane. Your derived class should not need to override this function, but should override `DrawCell` to draw each cell.

Text table

RefreshCell

```
void RefreshCell (Cell aCell);
```

Forces a cell to be redrawn by invalidating the region it occupies.

RefreshCellRect

```
void RefreshCellRect (Rect *cells);
```

Forces a rectangular region of cells to be redrawn by invalidating the region they occupy.

SetDrawActiveBorder

```
void SetDrawActiveBorder (Boolean
   fDrawActiveBorder);
```

If `fDrawActiveBorder` is `TRUE`, the table pane has a border when it is the gopher, that is, when it's active. Otherwise, the table pane has no border.

SetRowBorders

```
void SetRowBorders (short thickness,
   short penMode, ConstPatternParam penPat);
```

Sets thickness, transfer mode, and pattern for the borders between rows. The values should be such that they can be used by the Toolbox routines `PenSize`, `PenMode`, and `PenPat`.

SetColBorders

```
void SetColBorders (short thickness,
   short penMode, ConstPatternParam penPat);
```

Sets thickness, transfer mode, and pattern for the borders between columns. The values should be such that can be used by the Toolbox routines `PenSize`, `PenMode`, and `PenPat`. To have no column borders, set the thickness to 0.

Activate

```
void Activate ();
```

Makes the table pane active. Highlights the selected cells.

Deactivate

```
void Deactivate ();
```

Makes the table pane inactive. Removes the highlighting around the selected cells.

Insertion and deletion

AddRow

```
void AddRow (short numRows, short afterRow);
```

Inserts `numRows` many rows into the table. The rows are inserted following `afterRow` if `afterRow` is positive; or the rows are inserted before row 0 if `afterRow` is negative.

DeleteRow

```
void DeleteRow (short numRows, short startRow);
```

Deletes `numRows` many rows following `startRow`. If you specify rows beyond the last row, this deletes only the rows you specified in the table.

AddCol

```
void AddCol (short numCols, short afterCols);
```

Inserts `numCols` many columns into the table. The columns are inserted following `afterCol` if `afterCol` is positive; or the columns are inserted before column 0 if `afterCol` is negative.

DeleteCol

```
void DeleteCol (short numCols, short startCol);
```

Deletes `numCols` many columns following `startCol`. If you specify columns beyond the last column, this deletes only the columns you specified in the table.

Searching

FindRow

```
short FindRow (long vLoc);
```

Returns the row that contains vertical coordinate `vLoc`. `vLoc` should be in pane coordinates.

FindCol

```
short FindCol (long hLoc);
```

Returns the column that contains the horizontal coordinate `hLoc`. `hLoc` should be in pane coordinates.

NextCell

```
Boolean NextCell (Boolean hNext, Boolean vNext,
   Cell *theCell);
```

Finds the cell that follows `theCell`, using the criteria set with `hNext` and `vNext`. If this function finds the next cell, it puts the value in `theCell` and returns `TRUE`. Otherwise, it leaves `theCell` unchanged and returns `FALSE`. Use it to iterate through all the cells in a row, column, or table.

This table shows how to set `hNext` and `vNext`.

To go…	And stop at the end of…	Set these to… hNext	vNext
To the right	Row	TRUE	FALSE
Downwards	Column	FALSE	TRUE
Row by row	Table	TRUE	TRUE

If both hNext and vNext are FALSE, NextCell returns FALSE and doesn't change theCell.

This code fragment sums all the values in a column of a numeric table:

```
int   sum = 0;
Cell  currCell = { 0, 0 };
                  // The cell to start at.

do
{
   sum += GetCellValue(currCell);
                  // Sum up cell values.
} while (NextCell(FALSE, TRUE, &currCell));
                  // Iterate until the end
                  // of the column
```

GetSelect

```
Boolean GetSelect (Boolean next, Cell *theCell);
```

Determines if theCell is selected. If theCell isn't selected, this function can search for the next cell that is selected, depending on the value of next.

If theCell is…	And next is…	GetSelect does this…
Selected	TRUE or FALSE	Returns TRUE.
Not selected	FALSE	Returns FALSE.
Not selected	TRUE	Searches for the next cell that is selected. Returns TRUE if it finds one, FALSE otherwise.

GetSelect searches to the right as far as it can and then continues from the beginning of the next row. If it finds a selected cell, it sets theCell to that cell and returns TRUE. If it doesn't find a selected cell, it leaves theCell unchanged and returns FALSE.

Mouse

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handles a mouse down event in a table pane. There are three cases:

Click	Action
Click in cell	Calls the table's `MakeMouseTask` to create a mouse task, and then `TrackMouse` to select cells
Double Click in cell	Calls the table's `DoDblClick` with the cell clicked in
Click outside cells	Calls the table's `ClickOutsideBounds`

To change how your derived class selects cells, set the selection flags with `SetSelectionFlags`. To handle clicks with your own mouse task, your derived class should override `MakeMouseTask`.

DoDblClick

```
void DoDblClick (Cell hitCell,
   short modifierKeys, long when);
```

Responds to a double-click in a cell. This function calls the table's `DoCommand` function with the command `dblClickCmd` as argument. Your derived class can handle the command or let the table's supervisor handle it.

HitSamePart

```
Boolean HitSamePart(Point pointA, Point pointB);
```

Returns `TRUE` if the points are in the same cell; `FALSE` otherwise. Used to determine if two clicks are a double-click.

FindHitCell

```
void FindHitCell (LongPt *hitPt, Cell *hitCell);
```

Sets `hitCell` to the cell that contains `hitPt`. Sets `hitCell` to (`-1`, `-1`) if `hitPt` is not in a cell.

Printing

Paginate

```
void Paginate (CPrinter *aPrinter,
   short pageWidth, short pageHeight);
```

Paginates the table so that only full cells are printed on a page.

Command

DoCommand

```
void DoCommand (long aCmd);
```

Handles commands for table panes. This function handles only `cmdSelectAll` which select all the table's cells. Override it if you want your derived class to handle more commands.

UpdateMenus

```
void UpdateMenus ();
```

Updates menu items that table panes handle. This function enables only the **Select All** command. Override it if your derived class handles more menu commands.

DoKeyDown

```
void DoKeyDown(char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handles the arrow keys (left, right, up, and down) by selecting the cell in the arrow's direction from the currently selected cell. If there is no selected cell, it moves from (0, 0). If more than one cell is selected or if moving in the arrow's direction would move you off the table, the selection remains unchanged.

DoAutoKey

```
void DoAutoKey(char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handles repeated keys by calling the table pane's `DoKeyDown` function.

BecomeGopher

```
Boolean BecomeGopher (Boolean fBecoming);
```

If `fBecoming` is `TRUE`, this table pane is becoming the gopher, and this function activates the table pane. If `drawActiveBorder` is `TRUE` and the table is in a scroll pane, this function also draws a border around the scroll pane.

If `fBecoming` is `FALSE`, this table pane is no longer the gopher, and this function deactivates the table. If `drawActiveBorder` is `TRUE` and the table is in a scroll pane, this function also removes the border around the scroll pane, unless the scroll pane had a border before becoming the gopher; in that case, it restores the previous border.

Selection

SelectCell

```
void SelectCell (Cell aCell,
   Boolean keepPrevious, Boolean reDraw);
```

Selects the cell `aCell`. If `keepPrevious` is `TRUE`, it adds the new cell to the previous selection; otherwise, the new cell replaces the previous selection. If `redraw` is `TRUE`, the cell is redrawn; otherwise, it is invalidated.

SelectRect

```
void SelectRect (Rect *selectRect,
   Boolean keepPrevious, Boolean reDraw);
```

Selects the rectangular region of cells `selRect`. If `keepPrevious` is `TRUE`, it adds the new cells to the previous selection; otherwise, the new cells replace the previous selection. If `redraw` is `TRUE`, the cells are redrawn; otherwise, they are invalidated.

DeselectCell

```
void DeselectCell (Cell aCell, Boolean redraw);
```

Deselects the cell `aCell`. If `redraw` is `TRUE`, this function redraws the cell.

DeselectRect

```
void DeselectRect (Rect *deselectRect,
   Boolean redraw);
```

Deselects the rectangular region of cells `deselectR`. If `redraw` is `TRUE`, this function redraws the cells.

DeselectAll

```
void DeselectAll (Boolean redraw);
```

Deselects all the cells in the table. If `redraw` is `TRUE`, this function redraws the cells.

Scrolling

ScrollToSelection

```
void ScrollToSelection ();
```

Scrolls the current selection into view.

Conversion

PixelsToCells

```
Boolean PixelsToCells (LongRect *pixelsRect,
   Rect *cellsRect);
```

Sets `cellsRect` to the cells that contain the rectangle `pixelsRect`. Returns `TRUE` if `pixelsRect` contains any cells; returns `FALSE` otherwise. `pixelsRect` is in the view's coordinate system.

CellsToPixels

```
Boolean CellsToPixels (Rect *cellsRect,
   LongRect *pixelsRect);
```

Sets `pixelsRect` to the rectangle that contains the cells `cellsRect`. Returns `TRUE` if `cellsRect` is within the table's bounds; returns `FALSE` otherwise. `pixelsRect` is in the view's coordinate system.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Protected

DrawCell

```
void DrawCell (Cell theCell, Rect *cellRect);
```

Draws a single cell. `cellRect` is the position of the cell in short coordinates.

This function calls `GetCellText` to get a string, and then draws it. If your data can be represented in a string, override `GetCellText`. If your data is graphic, override `DrawCell`.

By default, this function indents by three points horizontally and by the font ascent height vertically. If you want to change the amount, your derived class should change the value of `indent`.

DrawRow

```
void DrawRow (short rowNum, short firstCol,
   short lastCol, Boolean doHilite);
```

Draws a row of cells. `Draw` calls this function if you asked it to draw by row-major order (that is, `drawOrder` is `tblRow`). To set the drawing order, use `SetDrawOrder`, described earlier in this chapter.

If your derived class needs to do some preparation before drawing a row, such as reading a record from a database, override this function.

DrawCol

```
void DrawCol (short colNum, short firstRow,
   short lastRow, Boolean doHilite);
```

Draws a column of cells. `Draw` calls this function if you asked it to draw by column-major order (that is, `drawOrder` is `tblCol`). To set the drawing order, use `SetDrawOrder`, described earlier in this chapter.

If your derived class needs to do some preparation before drawing a column, such as computing a vector of numbers, override this function.

DrawBorders

```
void DrawBorders (Rect *theCells);
```

Draws the row and column borders for the cells `theCells`.

HiliteCellRegion

```
void HiliteCellRegion (RgnHandle cellRgn,
   Boolean fHilite);
```

Draws or removes highlighting for a (not necessarily rectangular) region of cells. `HiliteFlag` should be `TRUE` if you're highlighting cells, `FALSE` if you're removing the highlighting. This function assumes you've already called `Prepare`. If you're highlighting the region, it assumes every cell in `cellRgn` is in `itsSelection`. Otherwise, it assumes every cell in `cellRgn` is not in `itsSelection`.

If the region is a rectangle, this function calls `Hilite`. If the region is not a rectangle, this function calls `Hilite` on each cell in the region.

Hilite

```
void Hilite (Rect *cells, Boolean hiliteFlag);
```

Draws or removes highlighting for a rectangular region of cells. `HiliteFlag` should be `TRUE` if you're highlighting the cells, `FALSE` if you're removing the highlighting. It assumes you've already called `Prepare`. If you want to handle highlighting differently, override this function.

GetCellText

```
void GetCellText (Cell aCell,
   short availableWidth, StringPtr itsText);
```

Get the text for a cell. Sets `itsText` to the text for the cell `aCell`. `availableWidth` is the cell's width. This function returns a string of the form "*rr*,*cc*", where *rr* is the cell's row number and *cc* is the cell's column number.

`DrawCell` calls this function to get a cell's text. If you can represent your table's data in a string, override this function to return that string. Otherwise, override `DrawCell`.

CreateTextEnvironment

```
void CreateTextEnvironment ();
```

Creates and initializes the text environment used for drawing text. If you need to change the default indent and row height, your derived class should override this functionAppearance

ClickOutsideBounds

```
void ClickOutsideBounds (Point hitPt,
   short modifierKeys, long when);
```

Respond to a click inside the table's frame but outside the table's bounds. Deselects all the cells in the table.

MakeMouseTask

```
CMouseTask *MakeMouseTask (short modifiers);
```

Creates a task to track the user's mouse dragging. This function creates a CTableDragger, described in Chapter 114, "CTableDragger." Override it in your derived class if you want to use your own task.

AdjustBounds

```
void AdjustBounds ();
```

Adjusts `bounds`, the size of the table in pixels, to match `tableBounds`, the size of the table in cells.

ITableX

```
void ITableX ();
```

Performs common initialization.

Member Functions: Private

CTableX

```
void CTableX ();
```

Performs common construction.

CTableDragger 114

Introduction

CTableDragger is a class that handles mouse tracking for the CTable class.

Heritage

Base Class	CMouseTask
Derived Classes	None

Using CTableDragger

CTableDragger handles mouse selection for the CTable class. When the user clicks in a table with the mouse, `CTable::MakeMouseTask` creates an instance of this class. If you create your own class to handle mouse tracking in tables, you should override `CTable::MakeMouseTask` in your CTable-derived class.

How CTableDragger selects cells

CTableDragger gives you flexibility in selecting cells. You choose which kind of selections your table allows (multiple cells, disjoint areas, and so forth) with the `SetSelectionFlags` function described in Chapter 113, "CTable." CTableDragger determines which kind of selection the user choses by determining if any modifier keys (Shift, Command, and so forth) were held down when the mouse button was clicked.

Table 114-1 lists settings of `SetSelectionFlags`.

Flag	Meaning
`selOnlyOne`	Only one cell is selected at a time.
`selNoDisjoint`	Disjoint selection is not allowed.

Table 114-1 Selection flags for CTable

Flag	Meaning
selExtendDrag	Extend selection by adding cell clicked, not by extending rectangle.
selDragRects	Dragging always selects rectangle.

Table 114-1 Selection flags for CTable *(Continued)*

CTableDragger uses three data members to keep track of how to select cells. Table 114-2 describes how these data members select cells:

Data member	If TRUE, dragging…	If FALSE, dragging…
fExtend	Adds newly selected cells to the current selection	Replaces current selection with newly selected cells
fSelectRects	Selects only rectangles	Adds individual cells to selection
fUseSense	Applies the state of the first cell clicked (selected or unselected) to all subsequent cells	Always selects cells

Table 114-2 Selection behavior data members

Table 114-3 describes how CTableDragger sets those data members, depending on the settings of the selection flags and modifier keys.

This flag…	Is TRUE if…
fUseSense	Command key is down
fSelectRects	fUseSense is FALSE, and selOnlyOne is off, and: • Shift key is down, or • selExtendDrag is on
fExtend	selOnlyOne is off, and: • fUseSense is TRUE, or • selExtendDrag is on, or • fSelectsRect is TRUE, and the Shift key is down

Table 114-3 How CTableDragger sets selection behavior flags

Data Members

CTableDragger defines the following protected data members:

Data member	Type	Description
itsTable	CTable*	The table
anchorCell	Cell	Initially clicked cell
prevCell	Cell	The cell clicked last
selectionFlags	long	A copy of table's flags
modifierKeys	short	Modifier keys from the mousedown event
fExtend	Boolean	TRUE if selections should be extended
fSelectRects	Boolean	TRUE if dragging always selects rectangles
fUseSense	Boolean	TRUE if dragging applies the state of the first cell clicked to all subsequent cells
fFirstWasSelected	Boolean	TRUE if first cell was selected

Member Functions

Creation and destruction

CTableDragger

```
CTableDragger (CTable *aTable,
   short theModifiers, long selFlags);
```

Constructor. `theModifers` should be the modifier flags from `DoClick`. `selFlags` are the table's selection flags.

This function initializes the data members `fUseSense`, `fSelectRects`, and `fExtend` based upon the table's selection flags and the modifier keys.

CTableDragger

```
CTableDragger ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `ITableDragger` for backward compatibly.

~CTableDragger

```
~CTableDragger ();
```

Destructor. Sets the data member `itsLastTask` of `itsTable` to `NULL` if that pointer is equal to this object.

ITableDragger

```
void ITableDragger (CTable *aTable,
   short theModifiers, long selFlags);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

Mouse tracking

BeginTracking

```
void BeginTracking (LongPt *startPt);
```

When mouse tracking is starting, this function calculates the anchor cell and makes the initial selection based upon the current selection settings.

KeepTracking

```
void KeepTracking (LongPt *currPt,
   LongPt *prevPt, LongPt *startPt);
```

When mouse tracking is under way, this function updates the selection and scrolls the table automatically.

Member Functions: Protected

CalcAnchorCell

```
void CalcAnchorCell (LongPt *startPt,
   Cell hitCell);
```

Determines which cell to use as the anchor cell when selecting a rectangular block of cells.

CTask 115

Introduction

CTask is an abstract class for implementing undoable actions.

Heritage

Base Class	None
Derived Classes	CMouseTask
	CTextEditTask
	CTextStyleTask

Using CTask

A task is an object of a class for implementing undoable actions. If you want your application to be able to undo an action, you need to define a task class, derived from CTask, for each type of undoable action.

You can use a task two ways. In the first way, your application can perform an action, create a task object, store enough information in it for its `Undo` function to undo the action, and call the `Notify` function of your supervisor (usually the document).

The second way is similar to the first, but you also implement a `Do` function that performs an action. Thus, you create a task, call its `Do` function to perform the action, and then call the `Notify` function of its supervisor. Your `Do` function stores enough information in the task's data members for its `Undo` function to undo the action.

When you notify a document that you've performed a task, it stores the task in its data member `lastTask`. When you choose **Undo** from the **Edit** menu, the document's `DoCommand` function calls that task's `Undo` function.

Every task class uses a string in the 'STR#' 130 resource used for the wording of the **Undo/Redo** command. Every task has a data member that is the index of this string in the 'STR#' resource. The document's UpdateUndo function takes care of the wording of the **Undo/Redo** command.

For example, suppose you've derived a class from CTask to change the font in an edit text pane. Before passing the command on to the edit pane's DoCommand function, you create a task and store the current font in a task data member. After you pass the font change command to the edit text pane, you pass the task to the document by calling the latter's Notify function. Your Undo function simply passes the font change command to the document using the font saved in the task's data member. Since the command goes through the regular command chain, your DoCommand function creates a task to let you undo what you were undoing.

Data Members

CTask defines these data members:

Data member	Type	Description
nameIndex	short	Index of the **Undo/Redo** string in the 'STR#' 130 resource.
undone	Boolean	TRUE if this task has been undone.

Member Functions

Creation and destruction

CTask

```
CTask (short aNameIndex = 0);
```

Constructor. aNameIndex is the index of the task's **Undo** string in the 'STR#' 130 resource; the default value is zero.

~CTask

```
~CTask ();
```

Virtual destructor.

ITask
```
void ITask (short aNameIndex);
```
Initialization function for backward compatibility. Need not be called if the constructor has an argument.

Dispose
```
void Dispose ();
```
Dispose function for backward compatibility. Calls `delete this`. You must have `TCL.USEDISPOSE` defined to use this function.

Accessing

GetNameIndex
```
short GetNameIndex ();
```
Gets the task's index. Used by the document's `UpdateUndo` function.

IsUndone
```
Boolean IsUndone ();
```
Returns whether or not this task is undone.

Action

Do
```
void Do ();
```
Performs a task. The default function does nothing. If you want to use a task to implement an action that is not necessarily undoable, your derived class should override this function. The **Undo/Redo** mechanism doesn't call `Do`.

Undo
```
void Undo ();
```
Undoes a task. The default function toggles the value of `undone`. Your derived class must store enough information to be able to undo an action. This is the function in which you implement the undo.

Redo
```
void Redo ();
```
Redoes a task that was undone. The default function calls the task's `Undo` function. This function assumes that a redo is the same as

undoing the undo. If your application implements **Redo** differently, you need to override this function.

Class Resources

Resource	Description
`'STR#' 130`	List of strings for the wording of the **Undo**/**Redo** command. For instance, if you're implementing a "move" action, your string would be "Move." Each task contains the index of its string in this resource, the data member `nameIndex`.

CTearChore 116

Introduction

CTearChore is a chore that notifies a tear-off menu that it has been torn off.

Heritage

Base Class	CChore
Derived Classes	None

Using CTearChore

CTearChore is used in CTearOffMenu's `TornOff` function to inform a menu that it has been torn off from the menu bar. `TornOff` creates a tear chore and assigns it as an urgent chore to the application. Your application should not have to use CTearChore directly, but you may find it useful as an example chore.

Data Members

CTearChore defines one data member:

Data member	Type	Description
`itsTearOffMenu`	`CTearOffMenu*`	The tear-off menu that has been torn off.

Member Functions

Creation and destruction

CTearChore

```
CTearChore (CTearOffMenu *aTearOffMenu);
```

Constructor. `aTearOffMenu` is stored in `itsTearOffMenu`. The default value is `0`. CTearOffMenu's `TornOff` function creates a tear chore and assigns it as an urgent chore.

ITearChore

```
void ITearChore (CTearOffMenu *aTearOffMenu);
```

Initialization function compatible with previous release. Should not be called if constructor has an argument.

Perform

```
void Perform (long *maxSleep);
```

Calls the `MoveToCorner` function of `itsTearOffMenu`. This function does not change `maxSleep`.

CTearOffMenu 117

Introduction

CTearOffMenu is an abstract class that implements a Macintosh tear-off menu.

Heritage

Base Class	CFloatDirector
Derived Classes	CSelectorDirector

Using CTearOffMenu

CTearOffMenu is a director that holds the pane of a menu torn off from the menu bar. To use a CTearOffMenu, first create a derived class, and then create a pane in the derived class constructor. Next, create an MDEF object (CSelectorMDEF or another descendant of CPaneMDEF) and pass both the pane and the CTearOffMenu-derived object to the constructor of the MDEF object.

The window that CTearOffMenu uses to display the tear-off menu is a floating window.

The Art Class example provided with the THINK Class Library uses two derived classes of CTearOffMenu for the tool palette and the pattern palette.

Data Members

CTearOffMenu has no data members.

Member Functions

CTearOffMenu

```
CTearOffMenu (short WINDid);
```

Constructor. A tear-off menu's supervisor is always the application. `WINDid` is the resource ID of the window that the tear-off menu appears in. Tear-off menus should use the window ID `'WDEF'`. The source for this `'WDEF'` as well as the `'WDEF'` resource itself is in the `FW/Tearoffs` folder.

`CTearOffMenu` sets `itsPane` to `NULL`, so your derived class needs to create a pane and set `itsPane` to it. This should be the same pane that you pass to CPaneMDEF's constructor.

CTearOffMenu

```
CTearOffMenu ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `ITearOffMenu` for backward compatibility.

ITearOffMenu

```
void ITearOffMenu (CApplication *aSupervisor,
   short WINDid);
```

Initialization function compatible with previous release. May not be called if the constructor has arguments.

TornOff

```
void TornOff(Point aCorner);
```

The menu has been torn off to the point `aCorner`. This function creates a tear chore (see Chapter 116, "CTearChore") that moves the window to the appropriate location.

CTextEditTask 118

Introduction

CTextEditTask provides undo support for typing and editing in CAbstractText-derived classes.

Heritage

Base Class	CTask
Derived Classes	CStyleTEEditTask

Using CTextEditTask

CTextEditTask implements undo for typing and the **Cut**, **Copy**, **Paste**, and **Clear** commands in CAbstractText-derived classes. A text pane automatically creates an instance of CTextEditTask when you type, press Backspace, Forward Delete, or choose an editing command.

You may need to create a derived class of CTextEditTask if you create your own derived class of CAbstractText, especially if your CAbstractText-derived class allows text to have multiple styles or if it doesn't store its text in a single contiguous block of memory. For example, a CStyleText text pane uses a task of type CStyleTEEditTask to deal with the style scrap format.

Data Members

The following are protected data members:

Data member	Type	Description
itsTextPane	CAbstractText	Text pane that this task acts on
editCmd	long	Command being performed, cmdNull if user is typing
inserted	tTextRange	Info about the inserted text
deleted	tTextRange	Info about the deleted text
originalScrap	Handle	Contents of text scrap, before this task was created
stillTyping	Boolean	TRUE if user is typing
doText	Boolean	TRUE if this task changes the text in the text pane
doClip	Boolean	TRUE if this task changes the clipboard
typingEvent	EventRecord	Event record for last keystroke

tTextRange is defined as follows:

```
typedef struct
{
   Handle text;
   long start;
   long end;
   long selStart;
   long selEnd;
} tTextRange;
```

Member Functions

Creation and destruction

CTextEditTask

```
CTextEditTask (CAbstractText *aTextPane,
   long anEditCmd, short firstTaskIndex);
```

Constructor. `anEditCmd` is the command that this task is responding to, such as `cmdCut`, `cmdCopy`, `cmdPaste`, or `cmdClear`. If the task is responding to typing, `anEditCmd` is `cmdNull`. `aTextPane` is the text pane for this task. `firstTaskIndex` is the index of the first text-edit Undo string in `STR# 130`. The text-edit Undo strings are typically “Typing,” “Cut,” “Copy,” “Paste,” and “Clear.” This function sets `doClip` to `TRUE` if this command changes the contents of the Clipboard, sets `doText` to `TRUE` if this command adds or deletes text to the text pane, and saves the currently selected range.

CTextEditTask

```
CTextEditTask ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `ITextEditTask` for backward compatibility.

~CTextEditTask

```
~CTextEditTask ();
```

Destructor. Deletes the memory for this object.

ITextEditTask

```
void ITextEditTask (CAbstractText *aTextPane,
   long anEditCmd, short firstTaskIndex);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

Accessing

CanStillType

```
Boolean CanStillType ();
```

Returns `TRUE` if the user’s typing doesn’t start a new task. The user can type if the user hasn’t tried to undo this task and `stillTyping` is `TRUE`.

Action

Do

```
void Do ();
```

Performs an **Edit** menu command by calling the text pane's `PerformEditCommand` function. This function then stores the resulting selection.

DoTyping

```
void DoTyping (char theChar, short keyCode,
   EventRecord *macEvent);
```

Types a character. Depending on the character, this function calls `DoBackspace`, `DoFwdDelete`, or `DoNormalChar`. This function is called only when performing a command, not when undoing one.

Undo

```
void Undo ();
```

Undoes this task. This function saves the text the user inserted, removes the inserted text, and restores the text the user deleted. If you choose **Cut** or **Copy**, it also restores the old Clipboard.

Redo

```
void Redo ();
```

Does this task after it's been undone. This function removes the text the user deleted and restores the text the user inserted. If you chose **Copy** or **Cut**, it restores the new Clipboard.

CancelTyping

```
void CancelTyping ();
```

Stops accumulating characters that the user types. This function is called if this task was created to handle typing and the user has just stopped to move the cursor or to select **Undo**. After this function is called, you can undo the typing until you start editing again or initiate another task.

SelectionChanged

```
void SelectionChanged ();
```

When the selection has changes this function calls `CancelTyping`.

Member Functions: Protected

DoNormalChar

```
void DoNormalChar (char theChar);
```

Handles a key that is not a Backspace or Forward Delete key, usually a cursor key or a character key. This function calls the `TypeChar` function of this object's text pane.

DoBackspace

```
void DoBackspace ();
```

Handles the Backspace key. If you're performing this task, this function saves the character you're deleting.

DoFwdDelete

```
void DoFwdDelete ();
```

Handles the Forward delete key. If you're performing this task, this function saves the character you're deleting.

SaveRange

```
void SaveRange (tRangeSelector whichRange);
```

If `whichRange` is `kDeletedText`, this function saves the text the user is about to delete. If `whichRange` is `kInsertedText`, it saves the text the user just inserted.

DeleteRange

```
void DeleteRange (tRangeSelector whichRange);
```

If `whichRange` is `kDeletedText`, this function deletes the text the user deleted. If `whichRange` is `kInsertedText`, it deletes the text the user inserted.

ReportChange

```
void ReportChange (Boolean isDo);
```

Calls `BroadcastChange` with the reason `textValueChanged`.

RestoreRange

```
void RestoreRange (tRangeSelector whichRange,
   Boolean killData);
```

If `whichRange` is `kDeletedText`, this function restores the text the user deleted. If `whichRange` is `kInsertedText`, it restores the text the user inserted. If `killData` is `TRUE`, it deletes this object's copy of the restored text.

StoreToClip

```
void StoreToClip (tClipSelector whichClip);
```

If `whichClip` is `kOldClip`, this function stores the original scrap text. If `whichClip` is `kNewClip`, it stores the deleted text to the Clipboard.

CTextEnvirons 119

Introduction

CTextEnvirons maintains a Quickdraw text-drawing environment for any pane.

Heritage

Base Class	CEnvironment
Derived Classes	CColorTextEnvirons

Using CTextEnvirons

Every pane has an `itsEnvironment` data member. If this member points to a descendant of CEnvironment, the `Prepare` function calls `Restore` to set up the drawing environment for the pane.

You can use CTextEnvirons to ensure that a pane's text-drawing characteristics are set up correctly. CTextEnvirons maintains the font, the size of the font, the font style, and the drawing transfer mode.

Suppose you have a pane that lets the user set the font and size of a text display. When you create your pane, you read the settings into a `TextInfoRec`, create a `CTextEnvirons` object, then set the `itsEnvironment` data member to point to that object. Whenever the pane needs to be drawn, the `Prepare` function calls `Restore` so that the drawing mode is set up correctly.

Here's an example of setting up a CTextEnvirons object for a pane:

```
CSomeDisplayPane::CSomeDisplayPane (
         CView *anEnclosure,
         CBureaucrat *aSupervisor)
{
   TextInfoRec aTextInfo;
   ...
   aTextInfo.fontNumber = ReadStoredFont();
   aTextInfo.theSize = ReadStoredSize();
   aTextInfo.theStyle = 0;
   aTextInfo.theMode = srcCopy;
   itsEnvironment = new CTextEnvirons;
   ((CTextEnvirons *) itsEnvironment)->
      SetTextInfo(&aTextInfo);
   ...
}
```

Data Members

CTextEnvirons defines one data member:

Data member	Type	Description
`textInfo`	`TextInfoRec`	Text characteristics

The `TextInfoRec` contains these members:

`short`	`fontNumber`	The number of the font.
`short`	`theSize`	Size of the font.
`style`	`theStyle`	Style of the font. In THINK C, this type is `short`.
`short`	`theMode`	Text transfer mode.

Member Functions

CTextEnvirons

```
CTextEnvirons ();
```

Constructor. Initializes every field of the `textInfo` record to zero. These settings correspond to the default system font, the default system size, plain style, and the `srcCopy` transfer mode.

ITextEnvirons

```
void ITextEnvirons ();
```

Initialization function for backward compatibility. Does nothing.

Restore

```
void Restore ();
```

Sets the Quickdraw text-drawing characteristics to the values previously stored with `SetTextInfo`. This function uses the standard QuickDraw text-setting routines: `TextFont`, `TextSize`, `TextFace`, and `TextMode`. This function also calls the Toolbox routine `PenNormal`.

SetTextInfo

```
void SetTextInfo (TextInfoRec *aTextInfo);
```

Sets the text-drawing characteristics to the values in `aTextInfo`. The next time the pane is redrawn, the QuickDraw text-drawing characteristics will be set to the values supplied.

GetTextInfo

```
void GetTextInfo (TextInfoRec *aTextInfo);;
```

Gets the current text-drawing characteristics from the object.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

CTextStyleTask 120

Introduction

CTextStyleTask provides undo support for style commands in CAbstractText-derived classes.

Heritage

Base Class	CTask
Derived Classes	CStyleTEStyleTask

Using CTextStyleTask

CTextStyleTask provides undo support for font, size, style, alignment, and spacing commands in CAbstractText-derived classes. A text pane automatically creates an instance of CTextStyleTask when you choose a style command.

You may need to create a derived class of CTextStyleTask if you create your own derived class of CAbstractText, especially if your CAbstractText-derived class allows text to have multiple styles or if it doesn't store its text in a single contiguous block of memory. For example, a CStyleText text pane uses a task of type CStyleTEStyleTask that can deal with the style scrap format.

Data Members

The following are protected data members:

Data member	Type	Description
`itsTextPane`	`CAbstractText*`	Text pane that this task acts on
`oldStyle`	`TextStyle`	Style before this task
`oldAlignCmd`	`long`	Alignment before this task

Data member	Type	Description
oldSpacingCmd	long	Spacing before this task
styleCmd	long	Command this task performs
styleAttribute	short	Style attributes affected by this task

Member Functions

Creation and destruction

CTextStyleTask

```
CTextStyleTask (CAbstractText *aTextPane,
   long aStyleCmd, short taskIndex);
```

Constructor. aStyleCmd is the command that this task is responding to. aTextPane is the text pane for this task. taskIndex is the index of this command's Undo string in 'STR#' 130.

CTextStyleTask

```
CTextStyleTask ();
```

Default constructor. Implicitly called when object is created by new_by_name. Can also be used in combination with ITextStyleTask for backward compatibility.

~CTextStyleTask

```
~CTextStyleTask ();
```

Destructor. Sets the pointer to this task that the text pane maintains to NULL.

ITextStyleTask

```
void ITextStyleTask (CAbstractText *aTextPane,
   long aStyleCmd, short taskIndex);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

Action

Do

```
void Do ();
```

Saves the original formatting, then performs the user's formatting command.

Undo

```
void Undo ();
```

Saves the current formatting and restores the previously saved formatting. This method handles both **Undo** and **Redo**.

Member Functions: Protected

SaveStyle

```
void SaveStyle ();
```

Saves the style of the text one of two ways, depending on the type of the text pane. It saves the style for the whole text pane, as in the case of CEditText, or it saves the style for the current selection, as in the case of CStyleText.

RestoreStyle

```
void RestoreStyle ();
```

Restores the previously saved formatting.

CView 121

Introduction

CView is an abstract class for implementing objects that have a visual representation. Every object in the visual hierarchy is derived from this class.

Heritage

Base Class	CBureaucrat
Derived Classes	CDesktop
	CPane
	CWindow

Using CView

CView is an abstract class for implementing objects with a visual representation. In other words, anything you can see on the screen is derived from CView. Views respond to visual commands involving the mouse. Because CView is a CBureaucrat, a view can also be a link in the chain of command.

The standard classes include three derived from CView. These are the desktop, windows, and panes. Most of the time, you deal with panes. Keep in mind that all functions that apply to views also apply to panes and classes derived from CPane.

Views and the visual hierarchy

The top of the visual hierarchy, the desktop, is the only view that does not have an enclosure. Every other view has an enclosure that specifies its place in the hierarchy. Each view can enclose multiple subviews. The desktop encloses all the windows in your application. Each window encloses one or more panes. Panes can enclose other panes.

The desktop handles some visual commands, such as mouse clicks, and forwards them to the appropriate window. The switchboard forwards window-related events, such as updates and activates, directly to a window, which forwards them to its subviews.

By default, a view does not process mouse clicks. If a view can respond to mouse clicks, you need to call `SetWantsClicks(TRUE)` when you create it to alert `DispatchClick` and `AdjustCursor` that they should look for clicks in that view.

Views and the chain of command

Each view has a supervisor, which is the view's boss in the chain of command. The desktop's supervisor is always the application. A window's supervisor is always its director or document. A pane's supervisor is usually its director or document.

The desktop and windows are almost never the first in the chain of command; they're almost never the gopher. Panes, on the other hand, are frequently made the gopher. For instance, you need to make an edit pane the gopher so that it can respond to typing and menu commands.

If a view can be a gopher, you need to call `SetCanBeGopher(TRUE)` so that `DispatchClick` can make the view the gopher and let the previous gopher know that it's not the gopher anymore. Although you can force a view to be the gopher by setting the `gGopher` global variable, it is better to rely on the gopher-setting mechanism of `BecomeGopher`.

Using Balloon Help with views

Any view can have a help balloon associated with it. The THINK Class Library uses `'hrct'` resources to specify help balloons although the Macintosh Help Manager, described in the "Help Manager" chapter of *Inside Macintosh: More Macintosh Toolbox*, allows a combination of `'hwin'` and `'hrct'` resources to display help balloons for non-resizable windows. You should not use `'hwin'` resources with the THINK Class Library.

Each window has a data member that can hold a resource ID to an `'hrct'` resource. If you do not specify a resource ID, then the class constructor supplies `kDefaultHelpResID` as the default ID.

If you want to provide help for a pane, set its `helpResIndex` data member, inherited from CView, as an index into the `'hrct'` resource. Otherwise, set `helpResIndex` to 0.

The `DispatchCursor` function uses `GetBalloonInfo` and `ShowHelpBalloon` to look for the help resource and to display it.

Data Members

CView defines the following public data members:

Data members	Type	Description
`macPort`	`GrafPtr`	Macintosh drawing port for the view.
`itsEnclosure`	`CView*`	View that totally encloses this one.
`itsSubviews`	`CViewList*`	Views contained within this view.
`visible`	`Boolean`	Is the view visible?
`active`	`Boolean`	Is the view active?
`wantsClicks`	`Boolean`	Does the view handle mouse clicks?
`canBeGopher`	`Boolean`	Can this view be the gopher?
`ID`	`long`	The identifier for this view.
`usingLongCoord`	`Boolean`	`TRUE` if using 32-bit coordinates.
`helpResIndex`	`short`	Index into `'hrct'` resource for balloon help.

The following three data members are static. CView uses these for tracking help balloons and for optimizing calls to `Prepare`. You usually don't need to use them.

Data member	Type	Description
`cCurrHelpView`	`CView*`	Used in `DispatchCursor` to determine whether a help balloon was displayed.
`cLastHelpView`	`CView*`	The view that is showing a help balloon.
`cPreparedView`	`CView*`	Currently prepared view.

Member Functions

Creation and destruction

CView

```
CView (CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Constructor. Views start out invisible, inactive, with no port and no subviews. By default, views don't want clicks. `anEnclosure` is the view that completely encloses this view. `aSupervisor` is the bureaucrat that gets the commands that this view can't handle.

CView

```
CView ();
```

Default constructor. Implicitly called when object is created by `new_by_name`. Can also be used in combination with `IView` for backward compatibility.

~CView

```
~CView ();
```

Destructor. Deletes all enclosed views and removes itself from its enclosure's subview list.

IView

```
void IView (CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialization function for backward compatibility. May not be called if constructor has arguments.

IViewRes

```
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initializes a view from a resource template. Each derived class of CView overrides this function and uses its own resource template. `rType` is the resource type for the CView-derived class you want to initialize. `resID` is the resource ID of the resource. `anEnclosure` is the view that completely encloses this view. `aSupervisor` is the bureaucrat that gets commands this view can't handle.

To initialize a view from a resource file, use a `'View'` resource.

IViewTemp

```
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

The `IViewRes` function calls `IViewTemp` to initialize a view from a template. Most derived classes of CView override this function so they can be initialized from templates.

Accessing

IsVisible

```
Boolean IsVisible ();
```

Returns `TRUE` if the view is visible.

IsActive

```
Boolean IsActive ();
```

Returns `TRUE` if the view is active.

ReallyVisible

```
Boolean ReallyVisible ();
```

Returns `TRUE` if the view and its enclosure are visible.

GetMacPort

```
GrafPtr GetMacPort ();
```

Gets the `GrafPort` associated with this view.

GetOrigin

```
void GetOrigin (long *theHOrigin,
   long *theVOrigin);
```

Gets the origin of the view (the upper-left corner). The default function returns (0, 0).

GetFrame

```
void GetFrame (LongRect *theFrame);
```

Gets the frame of the view. The default function does nothing. CView-derived classes (like CPane) must override this function.

GetInterior

```
void GetInterior (LongRect *theInterior);
```

Gets the interior of the view. The default function returns whatever `GetFrame` returns. If the interior of a particular derived class is not the same as the frame, the class must override this function.

GetAperture

```
void GetAperture (LongRect *theAperture);
```

Gets the aperture of the view. (The aperture is the visible portion of a view.) The default function does nothing. Derived classes must override this function.

Contains

```
Boolean Contains (Point thePoint);
```

Returns `TRUE` if the view contains `thePoint`. The default function always returns `FALSE`.

SetWantsClicks

```
void SetWantsClicks (Boolean aWantsClicks);
```

If `aWantsClicks` is `TRUE`, the view reports clicks in itself. By default, views don't want clicks.

GetWantsClicks

```
Boolean GetWantsClicks ();
```

Returns `TRUE` if this view wants to receive clicks.

SetCanBeGopher

```
void SetCanBeGopher (Boolean fCanBeGopher)
```

If `fCanBeGopher` is `TRUE`, this view can become the gopher. By default, a view can't become the gopher.

CanBeGopher

```
Boolean CanBeGopher ()
```

Returns `TRUE` if this view can become the gopher.

SetID

```
void SetID (long anIdentifier)
```

Sets this view's ID to be `anIdentifier`. By default, the ID is 0. It is up to you to provide IDs and to guarantee that they're unique. View IDs let you identify particular views within the view hierarchy. You can use Macintosh-style identifiers like `'MyVu'` or plain numeric constants.

GetID

```
long GetID ()
```

Returns the ID for this view.

UseLongCoordinates

```
void UseLongCoordinates (Boolean fUsing);
```

Specifies whether the view should use 32-bit coordinates (long coordinates) or 16-bit QuickDraw coordinates. If `fUsing` is `TRUE`, the view is using 32-bit coordinates.

Appearance

Show

```
void Show ();
```

Makes a view visible.

Hide

```
void Hide ();
```

Hides a view. If the view is the gopher, this function makes the supervisor the gopher.

Activate

```
void Activate ();
```

Makes a view active. Activates all the subviews as well.

Deactivate

```
void Deactivate ();
```

Makes a view inactive. Deactivates all the subviews as well. If the view is the gopher, this function makes the supervisor the gopher.

Mouse

DispatchClick

```
void DispatchClick (EventRecord *macEvent);
```

Finds out which subview got the click and calls its `DispatchClick` function. If there are no subviews, the click is for this view. If the view can be the gopher, `DispatchClick` calls `BecomeGopher(TRUE)` to make the view the gopher. Then `DispatchClick` calls `Prepare` to set up the drawing environment. Finally, `DispatchClick` calls the view's `DoClick` function to handle the click. Ordinarily, you should not need to override this function.

DoClick

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

This function is called when the mouse goes down in this view. If the view is using long coordinates, `hitPt` is given in QuickDraw coordinates. Otherwise, it is in frame coordinates. You can use CPane's `QDToFrame`, and `QDToFrameR` to convert from long coordinates to frame coordinates and `FrameToQD` and `FrameToQDR` to convert from long coordinates to QuickDraw coordinates. Derived classes must override this function.

These conversion routines are described in "QDToFrame" in Chapter 71, "CPane."

HitSamePart

```
Boolean HitSamePart (Point pointA, Point pointB);
```

Checks whether two points hit the same part of the view. The default function always returns `TRUE`. Derived classes that override this function should decide what constitutes a part and how close is close.

DoMouseUp

```
void DoMouseUp (EventRecord *macEvent);
```

This function is called when the mouse goes up in a view. The default function does nothing.

Cursor

DispatchCursor

```
void DispatchCursor (Point where,
   RgnHandle mouseRgn);
```

Finds which view the cursor is in. If the cursor is in this view, this function calls the `AdjustCursor` function of this object. If the cursor is in a subview, this function calls the subview's `DispatchCursor` function. `DispatchCursor` calls these member functions only for views that want clicks.

If the balloon help system is available, if the application has help resources, and if no other view has already displayed the help text, then `DispatchCursor` displays the help balloon associated with this view.

Your application should not need to override this function.

AdjustCursor

```
void AdjustCursor (Point where,
   RgnHandle mouseRgn);
```

Adjusts the cursor. A view's `AdjustCursor` function is called when the cursor moves into it and the view wants clicks. The default function sets the cursor to an arrow. Your derived class should override this function to set the cursor to whatever is appropriate for the view. See the `AdjustCursor` function in Chapter 14, "CAbstractText," for an example.

If you use only one cursor within a view, you can ignore the two parameters to this function. If you want to use different cursor shapes within a pane, you need to use these two parameters.

You are unlikely to want multiple cursors for a view that's not a pane.

The `where` parameter tells you where the cursor is in window coordinates. You can use the `WindToFrame` pane function to convert that `Point` to frame coordinates. The `mouseRgn` parameter is the region in which the cursor shape stays the same. In other words, your pane's `AdjustCursor` function is not called again until the cursor leaves this region. The `mouseRgn` is specified in global coordinates.

For example. suppose you're displaying a map of the United States in a pane and you want the cursor to be a star when it's over Texas. You use the `where` parameter, converted from window coordinates to frame coordinates, to determine that you're in Texas, and you change the cursor to a star. When you leave Texas, you want to set the cursor shape to something else. Since your pane's `AdjustCursor` function is called again only when the cursor leaves the `mouseRgn`, you have to change the `mouseRgn`. You create a region with the shape of Texas, convert this region to global coordinates, and make it the `mouseRgn`.

Note

The mouse region should be the intersection of the original mouse region and the one you're specifying. This way, you preserve any clipping boundaries that the original mouse region had established.

The `AdjustCursor` function for this example might look like this:

```
void MapPane::AdjustCursor(Point where,
   RgnHandle mouseRgn);
{
   Point       locWhere;
   RgnHandle   TexasRgn = NewRgn();
   Rect        TexasRect;
   Rect        TempRect;

   locWhere = where;
   WindToFrame(&locWhere);

      /* If we're not in Texas, */
      /* use the default.       */
   if (!ptInTexas(locWhere)) {
      CPane::AdjustCursor(where,
         mouseRgn);
      return ;
   }

      /* Set the star cursor. */
   SetCursor(star);

      /* Calculate the mouse region */
      /* in frame coords.           */
   OpenRgn();
   DrawTexas();
   CloseRgn(TexasRgn);
      /* Convert it to global coords. */
   TexasRect = (**TexasRgn).rgnBBox;
   tmpRect = TexasRect;
   FrameToGlobalR(&tempRect);
   OffsetRgn(TexasRgn,
      tempRect.left - TexasRect.left,
      tempRect.top - TexasRect.top)

      /* Set the mouse region. */
   SectRgn(mouseRgn, TexasRgn, mouseRgn);
}
```

GetBalloonInfo

```
void GetBalloonInfo (struct HMMessageRecord
   *helpData, Point *tip, RectPtr alternateRec,
   Ptr *tipProc, short *theProc, short *variant,
   short *method);
```

Sets up the parameters needed to show a help balloon. If `helpResIndex` is greater than 0, use it as an index into the `'hrct'` resource associated with this view. If the view is a pane, the enclosing window holds the resource ID of the `'hrct'` resource;

otherwise, it uses the 'hrct' resource with the ID kDefaultHelpResID (128).

This function calls the Help Manager's HMGetIndHelpMsg routine to get the information from the 'hrct' resource.

ShowHelpBalloon

```
void ShowHelpBalloon (struct HMMessageRecord
   *helpData, Point tip,
   RectPtr altRect, Ptr tipProc,
   short theProc, short variant, short method);
```

Uses the data from GetBalloonInfo and calls the Help Manager routine HMShowBalloon to display the help balloon. If there is an error, this function calls Failure. helpData comes from the 'hrct' resource. Tip, as returned from GetBalloonInfo, is set to the center of the view in global coordinates. altRect is the aperture of the view. tipProc is NULL. theProc is 0, the standard definition function. variant and method are both read from the 'hrct' resource.

GetHelpResID

```
short GetHelpResID ();
```

Returns the resource ID of the 'hrct' resource that has the balloon help information for this view. This function returns the ID of the THINK Class Library's default 'hrct' resource: kDefaultHelpResID.

Subview management

AddSubview

```
void AddSubview (CView *theSubview);
```

Adds theSubview to the view's itsSubviews list.

RemoveSubview

```
void RemoveSubview (CView *theSubview);
```

Removes theSubview from the itsSubviews list.

FindSubview

```
CView* FindSubview (Point hitPt);
```

Finds the subview that wants clicks and contains hitPt. hitPt is in window coordinates. This function finds the topmost subview. You should not override this function.

FindViewByID

```
CView* FindViewByID (long anID);
```

Finds the subview whose ID is `anID`. If more than one view has the same ID, this function finds the topmost subview.

MatchView

```
CView* MatchView (MatchViewProc matchProc,
   void *matchData)
```

Finds the first subview that returns `TRUE` for `matchProc`. This function finds the topmost subview.

`MatchProc` is a pointer to function of the form:

```
Boolean matchProc (CView *aView,
   void *matchData);
```

`MatchData` can be anything that your `matchProc` needs. For an example of `MatchView`, see the definition of `FindViewByID` in the file `CView.cp`.

SubpaneLocation

```
void SubpaneLocation (long hEncl, long vEncl,
   long *hLocation, long *vLocation);
```

Returns the position of a subview at `hEncl`, `vEncl` in the view's coordinates and at `hLocation` and `vLocation` in window coordinates.

Prepare

```
void Prepare ();
```

Prepares to draw or to do something after a click. The default function sets the static data member `cPreparedView` to `this`. The static data member `cPreparedView` points to the most recently prepared view. CPane uses `cPreparedView` to avoid redundant calls to `Prepare`.

View classes must override this function. The overriding functions should set `cPreparedView` to `this` or call the inherited function. See the definition of `Prepare` in CPane, for an example.

ForceNextPrepare

```
static void ForceNextPrepare ();
```

Clears `cPreparedView` to disable the optimization on the next call to `Prepare`. If you change the port, the clipping region, or the origin of a view, you must call this function to disable the `Prepare` optimization.

FrameToGlobalR

```
void FrameToGlobalR (LongRect *frameRect,
   Rect *globalRect);
```

Converts `frameRect` from frame coordinates to global coordinates and puts the converted coordinates into `globalRect`. The default function does nothing. View-derived classes must override this function.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Private

CViewX

```
void CViewX ();
```

Performs common initialization.

Class Resources

Defining resources for initializing views

The `IViewRes` function lets you initialize any view from a resource template.

Resource	Class
`'Pane'`	CPane
`'Pano'`	CPanorama
`'PctP'`	CPicture

Table 121-1 Resource templates for initializing views

Resource	Class
'ScPn'	CScrollPane
'AbTx'	CAbstractText, CEditText
'StTx'	CStaticText, CEditText
'View'	CView

Table 121-1 Resource templates for initializing views

When you use ResEdit to create view resource templates, keep in mind these values for sizing and clipping mnemonics.

Sizing values		Clipping values	
sizFIXEDLEFT	= 0	clipAPERTURE	= 0
sizFIXEDRIGHT	= 1	clipFRAME	= 1
sizFIXEDTOP	= 2	clipPAGE	= 2
sizFIXEDBOTTOM	= 3		
sizFIXEDSTICKY	= 4		
sizELASTIC	= 5		

Table 121-2 Sizing and clipping values

Installing the TMPL resources into ResEdit

You can use ResEdit to create the resource templates for each class. Symantec C++ includes a file TCL TMPLs that contains 'TMPL' resource templates that you can install into ResEdit. These 'TMPL's let you create and edit the resources above. Follow these steps to install the 'TMPL' resources into ResEdit:

1. Make a duplicate copy of ResEdit.
2. Double-click the copy of ResEdit that you just made.
3. Open the file TCL TMPLs.
4. Select the item TMPL.
5. Choose **Copy** from the **Edit** menu.
6. Open the original copy of ResEdit.
7. Choose **Paste** from the **Edit** menu.
8. Choose **Quit** from the **File** menu. When ResEdit asks you to confirm the changes, click Yes.
9. Delete your copied version of ResEdit.

ResEdit is included in your Symantec C++ package in the Utilities folder. To install it, follow the instructions in the *THINK C User's Guide*.

CVoidPtrArray 122

Introduction

CVoidPtrArray implements a variable-sized array of `void*` pointers.

Heritage

Base Class	CArray
Derived Class	CPtrArray

Using CVoidPtrArray

Use a CVoidPtrArray to maintain a list or array of untyped pointers, such as Macintosh `Handle` or `Ptr` values. If you want a list of objects, use CPtrArray or CList, instead.

See Chapter 123, "CVoidPtrArrayIterator," for a discussion of how to loop through the elements of a CVoidPtrArray.

Data Members

This class has a single data member:

Data member	Type	Description
`items`	`LongHandle`	A copy of CArray's `hItems` handle

Member Functions

Creation and destruction

CVoidPtrArray

```
CVoidPtrArray (short blockSize = 3);
```

Constructor. Each time the array must get more memory, this function allocates a block that can hold `blockSize` many pointers; the default value of `blockSize` is `3`.

CVoidPtrArray
```
CVoidPtrArray (CVoidPtrArray&source)
```
Copy constructor for CVoidPtrArray.

~CVoidPtrArray
```
~CVoidPtrArray ();
```
Destructor. Sets the `items` handle to `NULL`.

Copy
```
void *Copy ();
```
Returns a copy of the array.

Accessing items

NthItem
```
void *NthItem (long index);
```
Returns the pointer stored in location `index`. Arrays are indexed from 1 through the number of items in the array.

FirstItem
```
void *FirstItem ();
```
Returns the pointer stored in location 1.

LastItem
```
void *LastItem ();
```
Returns the pointer `LastItem` in the array.

Adding items

InsertAt
```
void InsertAt (void *ptr, long index);
```
Inserts the `ptr` at the specified `index` position, increases the size of the array by one item, and shifts pointers previously stored at `index` or higher up by one position.

Append
```
void Append (void *ptr);
```
Adds the `ptr` to the end of the array. Same as:

```
InsertAt(ptr, GetNumItems()+1)
```

Note

A common programming mistake is attempting to append an item to an array by calling:

```
InsertAt(ptr, GetNumItems());
```

Instead, this inserts the item immediately before the last item. Use the `Append` function to append correctly.

Add

```
void Add (void *ptr);
```

Same as `Append`. Provided for backward compatibility.

Prepend

```
void Prepend (void *ptr);
```

Inserts the `ptr` at the start of the array.

InsertAfter

```
void InsertAfter (void *ptr, void *afterPtr);
```

Locates the first occurrence of `afterPtr` in the array and inserts the `ptr` in the next higher location. Same as:

```
InsertAt(ptr, FindIndex(afterPtr)+1).
```

Removing items

The basic way to remove a pointer from a CVoidPtrArray is to call the `DeleteItem` function inherited from CArray:

```
DeleteItem
```

This removes the pointer at position `i` from the array, shifting all pointers at higher index locations down by one position, and reducing the size of the array by 1 item.

Remove

```
void Remove (void *ptr);
```

Locates the first occurrence of `afterPtr` in the array and removes it from the array. Same as `DeleteItem(ptr, FindIndex(afterPtr)).`

Testing

Includes

```
Boolean Includes (void *ptr);
```

Returns TRUE if `ptr` is stored in the array, FALSE if it is not.

Finding

FindIndex

```
long FindIndex (void *ptr);
```

Locates the first occurrence of `afterPtr` in the array and returns its index.

FirstSuccess

```
void *FirstSuccess (vTestFunc testFunc);
```

Starting from item 1 and searching forward through the array, this function applies `testFunc` to each item until `testFunc` returns TRUE or the end of the array is reached. If `testFunc` returns TRUE for an item, `FirstSuccess` returns that item; otherwise, it returns NULL. `testFunc` must be declared as:

```
Boolean testFunc(void *ptr);
```

The same search can be performed using an iterator:

```
CVoidPtrArrayIterator iter(myArray
                           kStartAtBeginning);
void *item;

while (iter.Next(item))
   if (testFunc(item))
   {
      item = NULL;
      break;
   }
// item contains the desired pointer or NULL
```

When an iterator is used, a separate function `testFunc` is not required; the test can be written inline. The iterator has the additional advantage that the index position of the located item can be determined without further searching: a call to `iter.GetCursor()` obtains the index.

FirstSuccess1

```
void *FirstSuccess1 (vTestFunc1 testFunc,
   long param);
```

Starting from item 1 and searching forward through the array, this function applies `testFunc` to each item and the `param` until `testFunc` returns `TRUE` or the end of the array is reached. If `testFunc` returns `TRUE` for an item, `FirstSuccess1` returns that item; otherwise, it returns `NULL`. `testFunc` must be declared as:

```
Boolean testFunc(void *ptr, long param);
```

LastSuccess

```
void *LastSuccess (vTestFunc testFunc);
```

Same as `FirstSuccess`, except that it searches the array backward starting from the last item. The same search can be performed using an iterator:

```
CVoidPtrArrayIterator iter(myArray
                           kStartAtEnd);
void *item;

while (iter.Prev(item))
   if (testFunc(item))
   {
      item = NULL;
      break;
   }
// item contains the desired pointer or NULL
```

When an iterator is used, a separate function `testFunc` is not required; the test can be written inline. The iterator has the additional advantage that the index position of the located item can be determined without further searching: the index position is `iter.GetCursor()`+1.

LastSuccess1

```
void *LastSuccess1 (vTestFunc1 testFunc,
   long param);
```

Same as `FirstSuccess1`, except that it searches the array backward starting from the last item.

FindItem

```
void *FindItem (vTestFunc testFunc);
```

Same as `FirstSuccess`. Provided for backward compatibility.

FindItem1

```
void *FindItem1 (vTestFunc1 testFunc,
   long param);
```

Same as `FirstSuccess1`. Provided for backward compatibility.

Looping

The following functions let you iterate through the pointers in an array and apply an external function to each pointer.

DoForEach

```
void DoForEach (vEachFunc eachFunc);
```

Starting from item 1 and proceeding forward through the array, this function applies `eachFunc` to each item. `eachFunc` must be declared as:

```
void eachFunc(void *ptr);
```

The same loop can be performed using an iterator:

```
CVoidPtrArrayIterator iter(myArray
                           kStartAtBeginning);
void *item;

while (iter.Next(item))
   eachFunc(item);
```

The iterator form lets you perform the operations inline without requiring an external function.

DoForEach1

```
void DoForEach1 (vEachFunc1 eachFunc)
   long param);
```

Starting from item 1 and proceeding forward through the array, this function applies `eachFunc` to each item and the `param`. `eachFunc` must be declared as:

```
void eachFunc(void *ptr, long param);
```

Moving items

The following functions permute the array by moving a single item to a different position in the array and shifting other items up or down as appropriate.

The `MoveItemToIndex` function inherited from CArray:

```
MoveItemToIndex(i, j);
```

moves the item in position `i` to position `j`.

MoveToIndex

```
void MoveToIndex (void *ptr, long index);
```

Locates the first occurrence of `ptr` in the array and moves that item to the `index` position.

BringFront

```
void BringFront (void *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item to the start of the array. Same as:

```
MoveToIndex(FindIndex(ptr), 1);
```

SendBack

```
void SendBack (void *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item to the end of the array. Same as:

```
MoveToIndex(FindIndex(ptr), GetNumItems());
```

MoveUp

```
void MoveUp (void *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item forward one position, if possible. Same as:

```
long i = FindIndex(ptr);
if (i < GetNumItems())
   MoveToIndex(i, i+1);
```

MoveDown

```
void MoveDown (void *ptr);
```

Locates the first occurrence of `ptr` in the array and moves that item backward one position, if possible. Same as:

```
long i = FindIndex(ptr);
if (i > 1)
   MoveToIndex(i, i-1);
```

Internal functions

Offset

```
long Offset (void *ptr);
```

Protected function. Same as `FindIndex`, except that it returns the index position minus one.

CVoidPtrArrayIterator 123

Introduction

CVoidPtrArrayIterator lets you iterate through the elements of a CVoidPtrArray array.

Heritage

Base Class	CArrayIterator
Derived Classes	None

Using CVoidPtrArrayIterator

CVoidPtrArrayIterator provides a straightforward and error-resistant way to loop through the items in a CVoidPtrArray. For example:

```
CVoidPtrArray *resArray;
...
CVoidPtrArrayIterator iter(array,
                           kStartAtBeginning);
Handle item;

while (iter.Next(item))
{
   short rID;
   ResType rType;
   Str255 rName;
   GetResInfo(item, &rID, &rType, rName);
   ...
}
```

The `Next` function fetches a handle from the array and advances the cursor (see below) past the item. Successive calls to `Next` traverse the entire array. When the cursor is at the end of the array, `Next` returns `FALSE`.

You could simply loop through the array with an index, but that sort of code is more error-prone: if the loop inserts or deletes elements in the array, it usually fails.

More than one iterator can operate on the same array at the same time. For example, here is a loop that removes duplicate handles from the array `resArray`:

```
CVoidPtrArrayIterator iter(resArray,
                           kStartAtBeginning);
CVoidPtrArrayIterator check(resArray, 0);
Handle orig, dup;

while (iter.Next(orig))
{
   check.MoveTo(iter.GetCursor());
   while (check.Next(dup))
      if (dup == orig)
         array->DeleteItem(check.GetCursor());
}
```

The cursor ranges from 0 to the number of elements in the array. It is logically positioned at the start of the array, or *between* two array items. For example, if an array has 5 items, a cursor with value 3 is positioned between items 3 and 4, as illustrated in Figure 123-1. `Next` returns item 4 and `Prev` returns item 3.

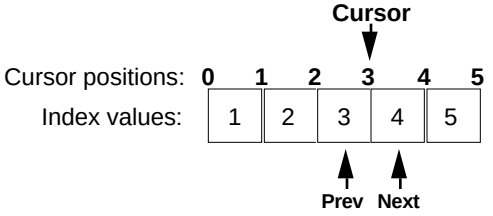


Figure 123-1 Relationship of cursor position to index

CVoidPtrArrayIterator inherits the `GetCursor` and `MoveTo` functions from CArrayIterator, which let you fetch the current cursor position and change the cursor position.

Unlike a loop index, an iterator is not required to go only forward or only backward; by calling `Next`, `Prev`, and `MoveTo`, you may go forward, backward, or jump around in an arbitrary way during the life of the iterator.

If the array associated with the iterator is deleted, the cursor is set to 0 and the `Next` and `Prev` functions return `FALSE`.

This class is implemented in `CVoidPtrArrayIterator.h`.

Data Members

This class has no data members.

Member Functions

Creation and destruction

CVoidPtrArrayIterator

```
CVoidPtrArrayIterator (CVoidPtrArray *array,
   CursorPosition start);
```

Constructor. `array` specifies the array to iterate through and `start` specifies where iteration begins. `start` may be specified as `kStartAtBeginning`, `kStartAtEnd`, `StartBefore(`*index*`)`, or `StartAfter(`*index*`)`, where *index* is the index of an array item, from 1 to the number of items in the array.

Advancing and retreating

Next

```
Boolean Next (void *&voidptr);
```

If there are items beyond the cursor, this function fetches the next item pointer, replacing the value of `voidptr`; then it increments the cursor and returns `TRUE`. Otherwise, it returns `FALSE`.

Prev

```
Boolean Prev (void *&voidptr);
```

If there are items before the cursor, this function fetches the previous item pointer, replacing the value of `voidptr`; then it decrements the cursor and returns `TRUE`. Otherwise, it returns `FALSE`.

CWatchDesc 124

Introduction

CWatchDesc lets you ensure that Apple-event descriptors created by a function without exception handlers are nevertheless disposed when the function returns or when an exception is thrown.

Heritage

Base Class	None
Derived Classes	None

CWatchDesc is declared as `TCL_AUTO_DESTRUCT_OBJECT`.

Using CWatchDesc

CWatchDesc is a class whose sole purpose is to ensure that Apple-event descriptor memory is properly managed.

When an object is allocated on the stack, its destructor is called when the block in which the object was declared is exited or (for objects of autodestruct classes) when an exception is thrown that terminates the block. See Chapter 9, “Exception Handling and RTTI,” for a discussion of autodestruct classes.

The behavior of autodestruct classes lets you create objects on the stack that use and properly delete heap objects. Thus, you can write

functions that clean up when exceptions are thrown without using try blocks and catch handlers. For example:

```
void CAppleEventObject::MakeSelfSpecifier(
         AEDesc *result)
{
   AEDesc   spec;
   AEDesc   desc;
   DescType keyForm;
   CAppleEventObject *container =
      GetContainer();

   MakeSelfDescriptor(&keyForm, &desc);
   CWatchDesc watchDesc(desc);

   container->MakeSelfSpecifier(&spec);
   CWatchDesc watchSpec(spec);

   FailOSErr(CreateObjSpecifier(GetClassID(),
      &spec, keyForm, &desc, FALSE, result));
}
```

Even though the function creates an Apple-event descriptor and object specifier, each of which allocates a handle on the heap, the function cleans up the temporary `desc` and `spec` when it returns or if an exception is thrown during its execution.

Data Members

CWatchDesc has the following private data member:

Data member	Type	Description
watched	`AEDesc*`	Pointer to descriptor to be disposed

Member Functions

Creation and destruction

CWatchDesc

```
CWatchDesc (AEDesc *watch);
```

Constructor. Sets `watched` to the value of `watch`.

CWatchDesc

```
CWatchDesc (AEDesc& watch);
```

Constructor. Sets `watched` to point to the `watch` descriptor.

~CWatchDesc

```
~CWatchDesc ();
```

Destructor. If `watched` is not `NULL`, this function disposes the descriptor to which it points.

Keeping the descriptor

Keep

```
void Keep ();
```

Sets `watched` to `NULL`. Call this function if the descriptor pointed to by `watched` is to be preserved beyond the lifetime of this CWatchDesc object.

CWindow 125

Introduction

CWindow implements a class necessary to manipulate a Macintosh window.

Heritage

Base Classes	CView CAppleEventObject CGroupButtonEnclosure
Derived Classes	None

Using CWindow

Virtually every application you write on the Macintosh uses windows. The CWindow class implements functions to manipulate Macintosh windows. In the THINK Class Library, window objects are merely visual entities. They're not designed to interact directly with the application. The class CDirector manages communication between a window and the application. The CDocument class, a derived class of CDirector, manages communication between the application, a window, and a file.

Objects of class CWindow usually belong to directors. Under normal circumstances, you should not need to define a derived class of CWindow or to manipulate one directly. All you really need to do to a window is create it and set its options.

You can specify that a window be modal. If a modal window is the frontmost window, CDesktop's `DispatchClick` function will not let a click in another window deactivate the modal window, although mouse clicks in the menu bar are handled. Make sure that your program provides a way to close modal windows.

The Macintosh windows associated with CWindow objects have a window kind of `OBJ_WINDOW_KIND`. If your application uses windows that are not associated with a CWindow object—from a utility library, for instance—you should: override the CDesktop functions `DispatchClick` and `DispatchCursor`; override CSwitchboard functions `DoUpdate` and `DoActivate`; and call `DoDeactivate` to check the window kind of the Macintosh windows that these functions deal with.

The enclosure for all windows must be `gDesktop`.

Data Members and Global Variables

Global variables

CWindow uses the following global variable:

Global variable	Type	Description
`gDesktop`	`CDesktop`	The global desktop that acts as the top of the visual hierarchy and the enclosure for all windows.

Data members

CWindow defines the following data members:

Data member	Type	Description
`procID`	`short`	Window definition ID used to create the window.
`sizeRect`	`Rect`	Minimum and maximum size of the window.
`floating`	`Boolean`	`TRUE` if this is a floating window.
`isColor`	`Boolean`	`TRUE` if this is a color window.
`isModal`	`Boolean`	TRUE if currently modal.
`actClick`	`Boolean`	TRUE if windows should process mouse click that activate window.
`location`	`Point`	Current window location, used when “hiding” a window while suspended.
`helpResID`	`short`	Resource ID of `'hrct'` help resource for panes in this window.
`hiding`	`Point`	Location of window used when “hiding” a floating window.

Member Functions

Creation and destruction

CWindow

```
CWindow (short WINDid, Boolean aFloating,
   CDirector *aSupervisor);
```

Constructor. Initializes a window object from a `'WIND'` resource. If Color QuickDraw is available, this function creates a color window.

`WINDid` is the ID of the `'WIND'` resource. The enclosure of a window is always the desktop. If `aFloating` is `TRUE`, the window is a floating window. In this case, the desktop must be a floating window desktop.

`aSupervisor` is the window's supervisor. A window's supervisor is the director it belongs to. `CWindow` calls the window's `MakeMacWindow` function to actually allocate space in memory for the window.

By default, a window is not modal. To make a window modal, use the `SetModal` function described later in this chapter.

By default, `helpResID` is set to `kDefaultResID`. `helpResID` is the resource ID of the `'hrct'` resource associated with this window. To change it, use the `SetHelpResID` function described later in this chapter. For more information about using Balloon Help with panes, see "Using Balloon Help with views" in Chapter 121, "CView."

CWindow

```
CWindow (Rect *bounds, Boolean fVisible,
   short aProcID, Boolean fFloating,
   Boolean fHasGoAway, CDirector *aSupervisor);
```

Alternate constructor to initialize a window from arguments rather than from a `'WIND'` resource. If Color QuickDraw is available, this function creates a color window.

`bounds` is a rectangle that describes the size and position of the window. `aProcID` is a `short` that describes that type of the window.

If `fVisible` is `TRUE`, the window is drawn immediately after it's created. If `fHasGoAway` is `TRUE`, the window has a close box.

The other parameters are identical to `CWindow`'s parameters above.

CWindow

```
CWindow ();
```

Default constructor. Implicitly called when an object is created by `new_by_name`. Can also be used in combination with `IWindow` for backward compatibility.

~CWindow

```
~CWindow ();
```

Destructor. Disposes of window's subviews and grafport, and removes the window from the desktop.

IWindow

```
void IWindow (short WINDid, Boolean aFloating,
   CDesktop *anEnclosure,
   CDirector *aSupervisor);
```

Initialization function compatible with previous release. May not be called if constructor has arguments.

INewWindow

```
void INewWindow (Rect *bounds,
   Boolean fVisible, short aProcID,
   Boolean fFloating, Boolean fHasGoAway,
   CDesktop *anEnclosure,
   CDirector *aSupervisor);
```

Initialization function compatible with previous release. May not be called if constructor has arguments.

IWindowX

```
void IWindowX ();
```

Common initialization.

MakeMacWindow

```
void MakeMacWindow (short WINDid);
```

Creates a Macintosh window from a `'WIND'` resource. This function lets the Window Manager allocate memory itself. If you want to create your windows differently, override this function. Bear in mind that floating windows should be in front (-1) and that non-floating windows should be behind.

MakeNewMacWindow

```
void MakeNewMacWindow (Rect *bounds,
   short aProcID, Boolean fHasGoAway);
```

Creates a Macintosh window from the parameters in the argument list. `bounds` is a rectangle that describes the size and position of the window. `aProcID` is a `short` that describes that type of window. If `FHasGoAway` is `TRUE`, the window has a close box on the left side of the title bar.

This function lets the Window Manager allocate memory itself. If you want to create your own windows differently, override this function. For example, you might want to perform your own memory allocation or create color windows. For compatibility with the Desktop class, the convention of putting floating windows in front and non-floating windows in back should be observed.

Close

```
void Close();
```

Closes a window. This member function of a window object is called as a result of a click in the close box. The default function calls the `CloseWind` function of the window's supervisor (a director).

UserClose

```
void UserClose ();
```

Apple-event-aware version of `Close`.

Accessing

GetBounds

```
void GetBounds(Rect* theBounds);
```

Returns the bounds of the window's drawing area in global coordinates.

GetFrame

```
void GetFrame(LongRect *theFrame);
```

Gets the frame of a window. The frame of the window includes the 1-pixel border around the window. The upper-left corner of `theFrame` is always (-1, -1).

GetInterior

```
void GetInterior(LongRect *theInterior);
```

Gets the interior of a window. The interior of the window is the same as its `portRect` and does not include any of the border pixels. The upper left of `theInterior` is always (0, 0).

GetAperture

```
void GetAperture (LongRect *theAperture);
```

Returns the drawable area of the window. For windows, this is the entire window. The upper-left corner of `theAperture` is always (0, 0).

IsFloating

```
Boolean IsFloating ();
```

Returns `TRUE` if this is a floating window.

IsModal

```
Boolean IsModal ();
```

Returns `TRUE` if this is a modal window.

SetModal

```
void SetModal (Boolean fModal);
```

Specifies whether a window is modal. If `fModal` is `TRUE`, the window is modal. If `fModal` is `FALSE`, it is not modal. If a modal window is the frontmost window, mouse clicks anywhere except in the window and in the menu bar are ignored.

IsColor

```
Boolean IsColor ();
```

Returns `TRUE` if this is a color window.

SetTitle

```
void SetTitle (Str255 theTitle);
```

Sets the window's title.

GetTitle

```
void GetTitle (Str255 theTitle);
```

Gets the window's title.

SetActClick

```
void SetActClick (Boolean anActClick);
```

If `anActClick` is `TRUE`, the window processes the mouse click that activated it.

WantsActClick

```
Boolean WantsActClick ();
```

Returns `TRUE` if the window processes the mouse click that activates it.

Contains

```
Boolean Contains (Point thePoint);
```

Returns `TRUE` if `thePoint` is inside the window. `thePoint` is in global coordinates.

SetSizeRect

```
void SetSizeRect (Rect *aSizeRect);
```

Sets the minimum and maximum size for the window. `aSizeRect`'s top and left values are the minimum height and width. `aSizeRect`'s bottom and right values are the maximum height and width.

SetStdState

```
void SetStdState (Rect *aStdState);
```

Sets the standard state of a window. The standard state is the size and location of the window when it's zoomed out. Before you call this member function of a window object, make sure that the Macintosh window associated with it supports zooming. If it does, the `spareFlag` in the window record is `TRUE`.

SetHelpResID

```
void SetHelpResID (short aResID);
```

Sets the resource ID of the `'hrct'` Balloon Help resource associated with this window. For more information about using Balloon Help with the THINK Class Library, see "Using Balloon Help with views" in Chapter 121, "CView."

GetHelpResID

```
short GetHelpResID ();
```

Gets the resource ID of the `'hrct'` resource associated with this window. If you do not set one with `SetHelpResID`, `helpResID` is set to `kDefaultHelpResID` by default.

Appearance

Show

```
void Show ();
```

Shows a window. Also calls the `ShowWind` function of its enclosure (the desktop).

Hide

```
void Hide ();
```

Hides a window. Also calls the `HideWind` function of its enclosure (the desktop).

Activate

```
void Activate ();
```

Activates a window and all of its panes. Also calls the `ActivateWind` function of its supervisor (a director).

Deactivate

```
void Deactivate ();
```

Deactivates a window and all of its panes. Also calls the `DeactivateWind` function of its supervisor (a director).

Select

```
void Select ();
```

Selects a window by bringing it to the front and making it active. Calls the `SelectWind` function of its enclosure (the desktop).

ShowResume

```
void ShowResume ();
```

Makes a window visible when an application resumes. If you hide a window when your application is suspended, call this member function of your window object from its director's `Resume` function.

HideSuspend

```
void HideSuspend (void);
```

Hides a window when an application is suspended. To hide a window when the application is being suspended, call this member function of the window object from its director's `Suspend` function. `HideSuspend` doesn't actually hide the window. Instead, it moves it out of the visible range so the front-to-back ordering of the windows doesn't change.

ShowOrHide

```
void ShowOrHide (Boolean showFlag);
```

Makes a window visible or invisible without generating any update or activate events.

HideFloat

```
void HideFloat ();
```

Hides a floating window by moving it offscreen. Calling this function for a non-floating window does nothing.

ShowFloat

```
void ShowFloat ();
```

Shows a floating window by moving it onscreen. Calling this function for a non-floating window does nothing.

SetShowFloatLoc

```
void SetShowFloatLoc (Point loc);
```

Sets the location that the floating window will move to when you call `ShowFloat` next.

Size and location

Drag

```
void Drag (EventRecord *macEvent);
```

Drags a window. This function is invoked when the user clicks in a window's drag region. The default function calls the desktop's `DragWind` function.

UserDrag

```
void UserDrag (EventRecord *macEvent);
```

Apple-event-aware version of `Drag`.

Resize

```
void Resize (EventRecord *macEvent);
```

Resizes a window. This function is invoked when you click and drag the window's grow region. After resizing, `Resize` calls the window's `ChangeSize` function.

UserResize

```
void UserResize (EventRecord *macEvent);
```

Apple-event-aware version of `Resize`.

Zoom

```
void Zoom (short direction);
```

Zooms a window. This function is invoked when the user clicks in a window's zoom box. Direction is either `inZoomIn` or `inZoomOut`. After changing the size of the window, this function calls the `AdjustToEnclosure` for each of its panes.

UserZoom

```
void UserZoom (EventRecord * macEvent);
```

Apple-event-aware version of `Zoom`.

Move

```
void Move (short hGlobal, short vGlobal);
```

Moves the window to the specified location in global coordinates. Note that the new position refers to the upper-left corner of the window's content region. The title bar will be above the specified coordinates.

ChangeSize

```
void ChangeSize (short width, short height);
```

Changes the size of the window to the specified width and height. After changing the window size, this function calls `AdjustToEnclosure` for each of its panes. This function respects the maximum and minimum window sizes you set with `SetSizeRect`.

MoveOffScreen

```
void MoveOffScreen ();
```

Moves the window out of the visible area of the desktop.

Drawing

Update

```
void Update ();
```

Updates the window. This function is invoked when the Switchboard processes an update event. `Update` calls the window's `Prepare` function and then calls `Draw` for each of the window's panes. You should not override this function.

Prepare

```
void Prepare ();
```

Sets the Macintosh port to be this windows port and prepares for drawing.

Mouse

DispatchClick

```
void DispatchClick (EventRecord *macEvent);
```

Finds a pane to handle the mouse click. You should not use or override this function. If you want to handle a click in a particular window-derived class, override the `DoClick` function (inherited from CView) instead.

DispatchCursor

```
void DispatchCursor (Point where,
   RgnHandle mouseRgn);
```

Finds a pane that handles `AdjustCursor`. This function converts `where` from global to window coordinates, and then lets the `DispatchCursor` function in CView take care of displaying Balloon Help if it is enabled. You should not use or override this function.

Conversion

FrameToGlobalR

```
void FrameToGlobalR (LongRect *frameRect,
   Rect *globalRect);
```

Converts `frameRect` from frame to global coordinates. Places the result in `globalRect`.

Object I/O

PutTo

```
void PutTo (CStream& aStream);
```

Writes to the stream.

GetFrom

```
void GetFrom (CStream& aStream);
```

Reads from the stream.

Member Functions: Private

CWindowX

```
void CWindowX ();
```

Private function; performs common construction.

Global Variables 126

Introduction

This chapter describes the global variables in the THINK Class Library. These variables are defined in `GlobalVars.cp`.

Global Objects

These globals hold pointers to the only instances of some classes. Most of them are initialized in the application initialization routines. You should not need to reset any of these variables yourself, except for `gSleepTime`.

---

gApplication

```
CApplication *gApplication;
```

The global application object. You should set this variable to an instance of your application class in your main program. See "Writing the main program" in Chapter 18, "CApplication," for an example.

---

gDesktop

```
CDesktop *gDesktop;
```

The desktop. This variable holds the only instance of CDesktop. It's initialized in the application function `MakeDesktop`. The desktop is the enclosure for all the windows in your application and is the top of the visual hierarchy.

---

gBartender

```
CBartender *gBartender;
```

The bartender. This variable holds the only instance of CBartender. It's initialized in the application function `MakeBartender`. The bartender converts menu choices into command numbers and handles most menu operations.

gClipboard
```
CClipboard *gClipboard;
```
The Clipboard. This variable holds the only instance of CClipboard. It's initialized in the application function `MakeClipboard`. The Clipboard is where data is cut and copied to and pasted from.

gGopher
```
Cbureaucrat *gGopher;
```
The gopher is the first bureaucrat to get commands. If the gopher can't handle the command, it passes control to its supervisor. Initially, the gopher is set to the application (`gApplication`). When a document becomes active, the gopher points to the document. A document can set the gopher to point to one of its panes.

gError
```
CError *gError;
```
The global error handler. This variable is initialized in the constructor for CApplication.

gDecorator
```
CDecorator *gDecorator;
```
The window dresser. This variable holds the only instance of CDecorator. It's initialized in the application function `MakeDecorator`. The decorator takes care of arranging windows on the screen.

Mouse Click Globals

The THINK Class Library uses these globals to count mouse clicks. The only variable you need to use is `gClicks`.

gLastMouseDown
```
EventRecord gLastMouseDown;
```
Event record of the last mouse-down event.

gLastMouseUp
```
EventRecord gLastMouseUp;
```
Event record of the last mouse-up event.

gLastViewHit

```
CView *gLastViewHit;
```

The last view the mouse went down in.

gClicks

```
short gClicks;
```

Click counter. This variable counts multiple clicks. Its value is 1 for a single click, 2 for a double click, 3 for a triple click and so on. To be considered a multiple click, three conditions must be met: 1) The current mouse down must be in the same view as the last mouse down; 2) no more than the amount of time specified by `GetDblTime` may elapse since the last mouse down; and 3) the view function `HitSamePart` must return `TRUE`.

Cursors

These cursor handles are initialized in the constructor for CApplication. You can use them as arguments to `SetCursor`. Remember that these are handles, so you have to call `SetCursor` like this:

```
SetCursor(*gWatchCursor);
```

gIBeamCursor

```
CursHandle gIBeamCursor;
```

I-beam for text views.

gWatchCursor

```
CursHandle gWatchCursor;
```

Watch cursor for waiting.

System Globals

You can use these globals to get information about the environment that your application is running in and about the state of your application.

---

gSystem

```
tSystem gSystem;
```

This global is a record that contains fields that tell you about the capabilities of the Macintosh under which your program is running. It's initialized in the `InspectSystem` function of CApplication.

`tSystem` is declared in `Global.h` like this:

```
typedef struct
{
   Boolean  hasWNE          : 1;
   Boolean  hasColorQD      : 1;
   Boolean  hasGestalt      : 1;
   Boolean  hasAppleEvents  : 1;
   Boolean  hasAliasMgr     : 1;
   Boolean  hasEditionMgr   : 1;
   Boolean  hasHelpMgr      : 1;
   Boolean  hasScriptMgr    : 1;
   Boolean  hasFPU          : 1;
   Boolean  hasProcessMgr   : 1;
   short    scriptsInstalled;
   short    systemVersion;
} tSystem;
```

The field `scriptsInstalled` tells you how many scripts are in use. The field `systemVersion` gives you the version of the Macintosh System that's running. The version number is stored as 2-byte-long number. For example, if `systemVersion` is `0x0607`, the System version number is 6.0.7.

---

gSleepTime

```
long gSleepTime;
```

The switchboard uses this value to pass to `WaitNextEvent`. It is the maximum time (in ticks) between events. `Perform` functions in CChore-derived classes and `Dawdle` functions in CBureaucrat-derived classes can change this value indirectly to force an idle event.

gInBackground

```
Boolean gInBackground;
```

TRUE if the application is in the background.

Utility Globals

gInEnvironment

```
Boolean gInEnvironment;
```

TRUE if you're running in the Symantec C++ debugging environment. FALSE if you're running as a stand-alone application.

gSignature

```
OSType gSignature;
```

The signature of your application. You should initialize this variable in your SetUpFileParameters application function. Use this variable whenever your application needs to create a file.

gUtilRgn

```
RgnHandle gUtilRgn;
```

Utility region. This region is initialized with NewRgn in the constructor for CApplication. You can use it wherever you need a region, but you should not rely on it being the same across calls to different functions.

Warning

If you use gUtilRgn, make sure that you never delete it. If you've created a complex region and you want to release some memory, use SetEmptyRgn to make it as small as possible.

TCL Utilities 127

Introduction

The THINK Class Library uses some library routines to deal with the Macintosh Toolbox, long coordinates, and memory allocation.

For a detailed description of the exception handling mechanism, see Chapter 9, “Exception Handling and RTTI.”

For more information about memory management, see “Handling low-memory situations” in Chapter 18, “CApplication.”

Toolbox Utilities

These routines simplify working with the Macintosh Toolbox. They are defined in `TBUtilities.cp`. Make sure to include `TBUtilities.h` before you use these functions.

QuickDraw utilities

DrawSICN

```
void DrawSICN (short SICNid, short index,
   Point location);
```

Draws a small icon at the point `location`. `SICNid` is the resource ID of a `'SICN'` resource. `index` is the number of the small icon within the resource.

PinInRect

```
void PinInRect (const struct LongRect *theRect,
   struct LongPt *thePoint);
```

Pins `thePoint` within `theRect`. This routine is similar to the Toolbox routine `PinRect`, except that the point is changed in place, and this routine does not subtract 1 at the right and bottom edges.

Window manager utilities

BringBehind

```
void BringBehind (WindowPtr macWindow,
   WindowPtr behindWindow);
```

`BringBehind` moves `macWindow` so that it is behind `behindWindow`.

IsDialogWindow

```
Boolean IsDialogWindow (WindowPeek macWindow);
```

Returns `TRUE` if `macWindow` is a dialog window.

IsMyWindow

```
Boolean IsMyWindow (WindowPeek macWindow);
```

Returns `TRUE` if `macWindow` is an application window.

IsSystemWindow

```
Boolean IsSystemWindow (WindowPeek macWindow);
```

Returns `TRUE` if `macWindow` is a system (desk accessory) window.

Dialog manager utilities

FindDlogPosition

```
void FindDlogPosition (ResType theType,
   short theID, Point *corner);
```

Returns in `corner` the coordinates of the upper-left corner of the dialog or an alert if the dialog was centered in the upper-third of the screen. This routine is useful for the standard file dialogs that ask you to supply a corner point. Otherwise, you should use `PositionDialog`.

PositionDialog

```
void PositionDialog (ResType theType,
   short theID);
```

Centers the bounding box of a dialog or an alert in the upper-third of the screen. `theType` is either `'DLOG'` or `'ALRT'`, and `theID` is the resource ID of the dialog or the alert. `PositionDialog` does not display the dialog; it just changes its location.

Font manager utility

GetFontNumber

```
void GetFontNumber (ConstStr255Param fontName,
   short *fontNum);
```

Given a font name, this function returns its font number in `fontNum`. If the font is not found, it returns a negative number.

Keyboard utilities

KeyIsDown

```
Boolean KeyIsDown (short theKeyCode);
```

Returns `TRUE` if the specified key is being held down. Note that the key code depends on the kind of keyboard. This function does not tell you if a specific key character is being held down.

AbortInQueue

```
Boolean AbortInQueue ();
```

Returns `TRUE` if there is a Command-Period pending in the event queue. If there is, the event is removed from the queue.

IsCancelEvent

```
Boolean IsCancelEvent (EventRecord *anEvent);
```

Returns `TRUE` if `anEvent` is a cancel event. On U.S. keyboards, a cancel event is Command-Period. For more information about cancelling in international environments, see Tech Note 263.

String utilities

Also see “Pascal String Utilities” later in this chapter.

CopyPString

```
void CopyPString (ConstStr255Param srcString,
   Str255 destString);
```

Copies the `srcString` to the `destString`.

ConcatPStrings

```
void ConcatPStrings (Str255 first,
   ConstStr255Param second);
```

Concatenates the `second` string to the `first`.

System font utilities

The system font (in the Window Manager `GrafPort`) is used to draw menus.

TCLResetSystemFont

```
void TCLResetSystemFont ();
```

Resets the Window Manager port's font and size to 0. Some applications change these global values and neglect to reset them. Unless they are 0, `SetSystemFont` does not work correctly.

TCLSetSystemFont

```
void TCLSetSystemFont (short aFont, short aSize);
```

Sets the system font and font size stored in the low-memory globals `SysFontFam` and `SysFontSize`.

TCLGetSystemFont

```
void TCLGetSystemFont (short *font, short *size);
```

Returns the current system font number in `font` and its size in `size`.

TCLSetHiliteMode

```
void TCLSetHiliteMode ();
```

Sets the hilite mode (inverse in the user's color).

Operating System Utilities

These functions are defined in `OSChecks.cp`. Make sure to include `OSChecks.h` before you use these functions.

TrapAvailable

```
Boolean TrapAvailable (short trapNum);
```

Returns `TRUE` if the specified trap is available.

WNEIsImplemented

```
Boolean WNEIsImplemented ();
```

Returns `TRUE` if the `WaitNextEvent` is available. `WaitNextEvent` is available in System 6.0 and later.

TempMemCallsAvailable

```
Boolean TempMemCallsAvailable ();
```

Returns TRUE if the MultiFinder temporary memory calls are available.

ColorQDIsPresent

```
Boolean ColorQDIsPresent ();
```

Returns TRUE if Color QuickDraw is available.

FlushCache

```
void FlushCache ();
```

Flushes the CPU cache. This is required on the 68040 after modifying code.

Long Coordinate Utilities

These routines operate on long rectangles and long points. These routines do not map long points from frame coordinates to QuickDraw coordinates, or vice versa. To map frame coordinates to QuickDraw coordinates, use the transformation functions in CPane. See "Coordinate transformation" in Chapter 71, "CPane."

These functions are defined in LongCoordinates.cp. Make sure to include LongCoordinates.h before you use these functions.

Long point utilities

QDToLongPt

```
void QDToLongPt (Point srcPt, LongPt *destPt);
```

Converts srcPt from a QuickDraw point to a long point and places the result in destPt.

LongToQDPt

```
void LongToQDPt (LongPt *srcPt, Point *destPt);
```

Converts srcPt from a long point to a QuickDraw point and places the result in destPt. If srcPt is not in QuickDraw space, it is clipped to 16 bits. To convert a long point in frame coordinates to a QuickDraw point, use CPane's FrameToQD function.

SetLongPt

```
void SetLongPt (LongPt *pt, long h, long v);
```

Sets the horizontal and vertical elements of long point `pt` to `h` and `v`.

AddLongPt

```
void AddLongPt (LongPt *srcPt, LongPt *destPt);
```

Adds `srcPt` and `destPt` and returns the result in `destPt`.

SubLongPt

```
void SubLongPt (LongPt *srcPt, LongPt *destPt);
```

Subtracts `srcPt` from `destPt` and returns the result in `destPt`.

EqualLongPt

```
Boolean EqualLongPt (LongPt *pt1, LongPt *pt2);
```

Returns `TRUE` if `pt1` and `pt2` are the same point.

PtInQDSpace

```
Boolean PtInQDSpace (LongPt *pt);
```

Returns `TRUE` if `pt` is within 16-bit QuickDraw coordinate space.

Long rectangle utilities

QDToLongRect

```
void QDToLongRect (Rect *srcRect,
   LongRect *destRect);
```

Converts `srcRect` from a QuickDraw rectangle to a long rectangle and returns the result in `destRect`.

LongToQDRect

```
void LongToQDRect (LongRect *srcRect,
   Rect *destRect);
```

Converts `srcRect` from a long rectangle to a QuickDraw rectangle and returns the result in `destRect`.

SetLongRect

```
void SetLongRect (LongRect *r, long left,
   long top, long right, long bottom);
```

Sets the elements of the long rectangle `r`.

OffsetLongRect

```
void OffsetLongRect (LongRect *r, long dh,
   long dv);
```

Offsets `r` by `dh` and `dv`. Positive values are to the right and down.

InsetLongRect

```
void InsetLongRect (LongRect *r, long dh,
   long dv);
```

Insets the sides of `r` by `dh` and `dv`. Positive values move the sides in.

SectLongRect

```
Boolean SectLongRect (LongRect *src1,
   LongRect *src2, LongRect *destRect);
```

Calculates the intersection of `src1` and `src2` and places the result in `destRect`. `destRect` may be the same as `src1` or `src2`. If `destRect` is not empy, returns `TRUE`.

UnionLongRect

```
void UnionLongRect (LongRect *src1,
   LongRect *src2, LongRect *destRect);
```

Calculates the union of `src1` and `src2` and places the result in `destRect`. `destRect` may be the same as `src1` or `src2`.

PtInLongRect

```
Boolean PtInLongRect (LongPt *pt, LongRect *r);
```

Returns `TRUE` if the long point `pt` is in the long rectangle `r`.

UnionLongRect

```
void UnionLongRect (LongPt *pt1, LongPt *pt2,
   LongRect *r);
```

Calculates the smallest rectangle that encloses `pt1` and `pt2` and returns the result in the long rectangle `r`.

EqualLongRect

```
Boolean EqualLongRect (LongRect *r1,
   LongRect *r2);
```

Returns `TRUE` if the long rectangles `r1` and `r2` are identical.

EmptyLongRect

```
Boolean EmptyLongRect (LongRect *r);
```

Returns `TRUE` if the long rectangle `r` encloses no points.

RectInQDSpace

```
Boolean RectInQDSpace (LongRect *r);
```

Returns `TRUE` if the long rectangle `r` is entirely within QuickDraw space.

THINK Class Library Utilities

These routines are generally useful when working with the THINK Class Library. Some of them work with the exception handling mechanism (see Chapter 9, "Exception Handling and RTTI") and with the memory management routines (see "Handling low-memory situations" in Chapter 18, "CApplication").

The first two functions are defined in `TCLUtilities.cp`; the rest are defined in `DialogFunctions.cp`. Make sure to include the appropriate corresponding header file whenever you use these functions.

Error reporting utilities

ErrorAlert

```
void  ErrorAlert (short error, long message);
```

Displays an alert for the given `error` and `message` codes. If the low word of `message` is 0, `ErrorAlert` looks for an `'Estr'` resource that matches the error code in `error`. If it can't find one, it displays a generic error message and the error number.

If the low word of `message` is greater than 0, it is used as an index into a `'STR#'` resource. If the high word of `message` is 0, `ErrorAlert` uses the resource `'STR#' 131`. If the high word of `message` is not 0, its value is added to 1024 to get the resource ID of a `'STR#'` resource. You can use the exception handler's `SpecifyMsg` function to build the message.

For instance, if `message` is `655,363`—`0x000A0003` hex—the resource ID of the `'STR#'` resource is 1024+10, and the index into it is 3.

If the application is running, `ErrorAlert` uses `'ALRT'` `ALRT_Exception` (251). Otherwise, it uses `'ALRT'` `ALRT_ExceptionAbort` (252).

GetErrorMsgText

```
void GetErrorMsgText (short error, long msg,
   Str255 errStr);
```

Returns the specified error text in `errStr`.

ShowAlert

```
void ShowAlert (short anALRT, short aSTR,
   short aStrIndex);
```

Displays the alert whose resource ID is `anALRT`. `aStr` is a resource ID, and `aStrIndex` is the index of a string in that resource that will appear as additional text in the alert.

FalseAlert

```
Boolean FalseAlert (short anALRT, short aSTR,
   short AStrIndex);
```

Calls `ShowAlert` and returns `FALSE`. `aStr` is a resource ID, and `aStrIndex` is the index of a string in that resource that will appear as additional text in the alert.

TrueAlert

```
Boolean TrueAlert (short anALRT, short aSTR,
   short AStrIndex);
```

Calls `ShowAlert` and returns `TRUE`. `aStr` is a resource ID, and `aStrIndex` is the index of a string in that resource that will appear as additional text in the alert.

OSTypeToStudy

```
void OSTypeToStudy (OSType type, Str255 str);
```

Converts an `OSType` stored in `type` to a string `str`.

StringToOSType

```
void StringToOSType (Str255 str, OSType *type);
```

Converts the string `str` to an `OSType` pointed to by `type`. The tail of `type` is padded with blanks, if necessary.

Memory allocation utilities

NewHandleCanFail

```
Handle NewHandleCanFail (long size);
```

Allocates a handle without drawing on the memory reserve. If the allocation fails, returns `NULL`.

ResizeHandleCanFail

```
void ResizeHandleCanFail (Handle theHandle,
   long newSize);
```

Resizes a handle without drawing on the memory reserve.

GetResourceCanFail

```
Handle GetResourceCanFail (ResType theType,
   short resID);
```

Gets a resource without drawing on the memory reserve.

GetNamedResourceCanFail

```
Handle GetNamedResourceCanFail (ResType theType,
   Str255 theName);
```

Gets a resource by name without drawing on the memory reserve.

SetAllocation

```
Boolean SetAllocation (Boolean canFail);
```

Changes the parameters that `gApplication` uses when the grow zone function is invoked because the memory manager can't satisfy a request. If `canFail` is `TRUE` (or `kAllocCanFail`), memory requests are allowed to fail without drawing on memory reserves. If `canFail` is `FALSE` (or `kAllocCantFail`), the application tries to satisfy the memory request from the reserves. `SetAllocation` returns the previous setting. You should reset the allocation parameters after you make a memory request.

You might do it like this:

```
void AClass::AFunction()
{
    Boolean oldAlloc;

    oldAlloc = SetAllocation(kAllocCanFail);
        request memory here
    SetAllocation(oldAlloc);
        fail gracefully if it doesn't succeed
}
```

This function calls the application's `RequestMemory` function. See "Handling low-memory situations" in Chapter 18, "CApplication," for more information on working with memory in the THINK Class Library.

SetCriticalOperation

```
void SetCriticalOperation (Boolean aCriticalOp);
```

Changes the parameters that `gApplication` uses when the grow zone function is invoked because the memory manager can't satisfy a request. If `aCriticalOp` is `TRUE`, more of the memory reserve is available to satisfy a failing memory request.

This function calls the application's `SetCriticalOperation` function. See "Handling low-memory situations" in Chapter 18, "CApplication," for more information on working with memory in the THINK Class Library.

EqualMem

```
Pascal Boolean EqualMem (void *p1, void *p2,
    long n);
```

Compares two blocks of memory and returns `TRUE` if they are equal. `p1` and `p2` point to memory; `n` is the number of bytes to compare.

SetMinimumStack

```
void SetMinimumStack (long minSize);
```

Sets the stack to be at least `minSize` bytes. If you use this routine, it must appear only once, and it must be the first statement in your program.

Memory disposal utilities

These memory utilities release different kinds of memory and set the variable that references them to NULL. These routines are implemented as macros, which are defined in Global.h. The names of these functions have been changed; the names used in the previous release (without the preceding TCL) are still available.

TCLForgetHandle

```
void TCLForgetHandle (Handle h); [MACRO]
```

If h is not NULL, calls DisposHandle(h) and sets h to NULL.

TCLForgetPtr

```
void TCLForgetPtr (Ptr p); [MACRO]
```

If p is not NULL, calls DisposPtr(p) and sets p to NULL.

TCLForgetResource

```
void TCLForgetResource (Handle r); [MACRO]
```

If r is not NULL, calls ReleaseResource(r) and sets r to NULL.

TCLForgetObject

```
void TCLForgetObject (void *&obj); [MACRO]
```

If obj is not NULL, calls obj's Dispose function and sets obj to NULL.

Long Coordinate QuickDraw Utilities

The following routines are long-coordinate versions of some common QuickDraw routines. The arguments that these routines take are long points and long rectangles in frame coordinates. If you use these routines, make sure that you call them only in a pane's Draw function, as they expect that the pane has already been prepared.

Note

These routines transform frame coordinates to QuickDraw coordinates every time you call them. It's usually more convenient to do the transformation once, then do all the drawing in QuickDraw coordinates.

These functions are defined in `LongQD.cp`. Make sure to include `LongQD.h` before using these functions.

Long rectangle drawing routines

LEraseRect

```
void LEraseRect (LongRect *area);
```

Erases the rectangle specified in `area`.

LFrameRect

```
void LFrameRect (LongRect *area);
```

Frames the rectangle specified in `area`.

LInvertRect

```
void LInvertRect (LongRect *area);
```

Inverts the rectangle specified in `area`.

LPaintRect

```
void LPaintRect (LongRect *area);
```

Paints `area` in the current pattern and mode.

LFillRect

```
void LFillRect (LongRect *area, Pattern pat);
```

Fills `area` in the pattern `pat` in `patCopy` mode.

Long oval drawing routines

LEraseOval

```
void LEraseOval (LongRect *area);
```

Erases the oval specified in `area`.

LFrameOval

```
void LFrameOval (LongRect *area);
```

Frames the oval specified in `area`.

LInvertOval

```
void LInvertOval (LongRect *area);
```

Inverts the oval specified in `area`.

LPaintOval

```
void LPaintOval (LongRect *area);
```

Paints `area` in the current pattern and mode.

LFillOval

```
void LFillOval (LongRect *area, Pattern pat);
```

Fills `area` in the pattern `pat` in `patCopy` mode.

Long rounded rectangle drawing routines

LEraseRoundRect

```
void LEraseRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Erases the rounded rectangle specified in `area`.

LFrameRoundRect

```
void LFrameRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Frames the rounded rectangle specified in `area`.

LInvertRoundRect

```
void LInvertRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Inverts the rounded rectangle specified in `area`.

LPaintRoundRect

```
void LPaintRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Paints `area` in the current pattern and mode.

LFillRoundRect

```
void LFillRoundRect (LongRect *area,
   short ovWidth, short ovHeight, Pattern pat);
```

Fills `area` in the pattern `pat` in `patCopy` mode.

Long bit transfer routine

LCopyBits

```
void LCopyBits (BitMap *srcBits, BitMap *dstBits,
   LongRect *srcRect, LongRect *dstRect,
   short mode, RgnHandle mask);
```

Transfers a bit image from `srcBits` to `dstBits`. For a full description of this routine, see the description of `CopyBits` in *Inside Macintosh Volume I.*

Long point pen routines

LMoveTo

```
void LMoveTo (long hLoc, long vLoc);
```

Moves the pen to the point `hLoc`, `vLoc`.

LLineTo

```
void LLineTo (long hLoc, long vLoc);
```

Draws a line from the current pen location to `hLoc`, `vLoc`.

Long region utility routines

LRectRgn

```
void LRectRgn (RgnHandle rgn, LongRect *rect);
```

Creates a region whose shape is specified by `rect`. `rgn` must be a handle to an existing region.

LClipRect

```
void LClipRect (LongRect *r);
```

Changes the clipping region of the current `GrafPort` to the long rectangle `r`.

Pascal String Utilities

The `TCLpstring.cp` module contains Pascal string function analogs to the standard C string functions. To use these functions, include `TCLpstring.h`. None of the functions calls `BlockMove`, making them much faster for short strings. The functions copy strings rather than modify strings in place. `TCLpstring.cp` must be placed in a memory-resident segment.

TCLpstrcpy

```
void TCLpstrcpy (void *dst, const void *src);
```

Copies the `src` Pascal string to the `dst` Pascal string.

TCLpstrncpy

```
void TCLpstrncpy (void *dst, const void *src,
   short n);
```

Copies the `src` Pascal string to `dst` for up to `n` bytes.

TCLpstrcat

```
void TCLpstrcat (void *dst, const void *src);
```

Catenates the `src` Pascal string to the `dst` Pascal string.

TCLpstrncat

```
void TCLpstrncat (void *dst, const void *src,
   short n);
```

Concatenates the `src` Pascal string to the `dst` Pascal string for up to `n` bytes.

TCLctopstrcpy

```
void TCLctopstrcpy (Str255 dst, char *src);
```

Copies the `src` C string to the `dst` Pascal string.

TCLptocstrcpy

```
void TCLptocstrcpy (char *dst, Str255 src);
```

Copies the `src` Pascal string to the `dst` C string.

TCLpstrcatlong

```
void TCLpstrcatlong (void *dst, const void *src,
   long n);
```

Copies the `src` Pascal string to the `dst` Pascal string for up to `n` bytes.

TCLptocstrncpy

```
void TCLptocstrncpy (char *dst,
   const unsigned char *src, long n);
```

Copies the `src` Pascal string to the `dst` C string for up to `n` bytes.

TCLctopstrncpy

```
void TCLctopstrncpy (unsigned char *dst,
   const char *src, long n);
```

Copies the `src` C string to the `dst` Pascal string for up to `n` bytes.

View Resource Utilities

The `ViewUtilities.cp` module contains functions that read view resources written by Visual Architect. To use these functions, include `ViewUtilities.h`.

TCLGetWindow

```
CWindow *TCLGetWindow (short resID,
   CBureaucrat *aSupervisor);
```

Reads a `'CVue'` resource with resource ID `resID` into memory and converts it to a window supervised by `aSupervisor`. Positions the window on the desktop as specified in the resource. Returns a pointer to the window.

TCLGetNamedWindow

```
CWindow *TCLGetNamedWindow (Str255 name,
   CBureaucrat *aSupervisor);
```

Reads a `'CVue'` resource with the specified name into memory and converts it to a window supervised by `aSupervisor`. Positions the window on the desktop as specified in the resource. Returns a pointer to the window.

TCLGetSubview

```
CPane *TCLGetSubview;(short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Reads a `'CVue'` resource with resource ID `resID` into memory and converts it to a pane enclosed by `anEnclosure` and supervised by `aSupervisor`. Positions the pane in its enclosure as specified in the resource. Returns a pointer to the pane.

TCLGetNamedSubview

```
CPane *TCLGetNamedSubview (Str255 name,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Reads a `'CVue'` resource with the specified name into memory and converts it to a pane enclosed by `anEnclosure` and supervised by `aSupervisor`. Positions the pane in its enclosure as specified in the resource. Returns a pointer to the pane.

TCLGetItemPointer

```
CView *TCLGetItemPointer (CView *aView,
   short itemNo);
```

Locates a subview by item number. Searches for a panorama downward through the visual hierarchy starting from `aView`. If `itemNo` is 0, this function returns the first panorama found; otherwise, it returns the panorama's subview at index position `itemNo`.

Index

Entries in **boldface** are menu commands or dialog boxes. Entries in `typewriter face` are functions, member functions, variables, keywords, or files.

For THINK Class Library member functions, macros, and utility functions, see the Function Index that follows.

C

D

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Function Index

This index contains member functions, macros and utility functions in the THINK Class Library.

A

This document was created with FrameMaker 4.0.4

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

Z

Function Index

This index contains member functions, macros and utility functions in the THINK Class Library.

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

Z