User's Guide

SYMANTEC.®

C++

DEVELOPMENT SYSTEM FOR
POWER MACINTOSH

VERSION 8

POWER MACINTOSH/
MACINTOSH

Symantec C++ for Power Macintosh ◆

User's Guide and Reference

Credits

Documentation Elizabeth Collins, John Minniti, Jeanne Munson, Stephen Raphel, Susan Rona, and Cambridge Publications

Development David Bustin, Thomas Cardozo, Thomas Emerson, Bob Foster, Udi Kalekin, Paul Kaplan, Doug Knowles, Jim Laskey, John Micco, Pat Nelson, Mark Romano, Phil Shapiro, and Rob Vaterlaus

Quality Assurance Celso Barriga, Colen Garoutte-Carson, Constantine Hantzopoulos, Kevin Irlen, Yuen Li, and Christopher Prinos

Technical Support Glenn Austin, Mark Baldwin, Craig Conner, Colen Garoutte-Carson, Rick Hartmann, Michael Hopkins, Steve Howard, Scott Morison, and Kevin Quah

Project Management Constantine Hantzopoulos, Doug Knowles, and David Neal

Product Management David Allcott

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

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.

SYMANTEC'S LICENSOR(S) MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. SYMANTEC'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

IN NO EVENT WILL SYMANTEC'S LICENSOR(S), AND THEIR DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY SYMANTEC'S LICENSOR) BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF SYMANTEC'S LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

SYMANTEC'S Licensor's liability to you for actual damages from any cause whatsoever, and regardless of the form of the action (whether in contract, tort (including negligence), product liability or otherwise), will be limited to $50.

Contents ◆

Part One: Welcome to Symantec C++ for Power Macintosh

Part Two: Creating an Application

Part Three: Learning by Example (Tutorials)

Part Four: Symantec Project Manager Reference

Part Five: TCL and VA Reference

Part Six: Appendixes

Symantec C++

Welcome to Symantec C++ for Power Macintosh

Part One

Overview ◆ 1

Symantec C++ is a complete development environment for creating both C and C++ applications on Power Macintosh computers. This chapter describes the features of Symantec's C++ version 8.0. It covers the requirements for using Symantec C++, provides suggestions about using this book, and describes the typographic conventions used throughout the text.

Product Highlights

This section highlights the product's existing features and its more recent enhancements over the last major release.

PowerPC Compilers

Symantec's C++ version 8.0 includes PowerPC compilers for both C and C++, which are 100% native on the PowerPC. These compilers support the same kind of language features as the previous 68000-based C and C++ compilers.

In version 8.0, the build error messages are persistent, that is, the Build Errors window remains on the screen when a new compile or search is started. Error messages for a project entry remain in the window until deleted by the user, or until the project entry is recompiled.

Project Management

The Symantec C++ for Power Macintosh provides a host of advanced project management features. The highlights are provided below.

Project models

Project models are templates from which new projects are created. They define the libraries, resources, and source code files that the project contains and the project's option settings. In addition to letting you create your own project models, Symantec C++ provides several predefined project models. Once created, any project file can be edited by adding or removing files. Unlike previous versions, project models can contain arbitrarily nested folders.

Precompiled headers
Symantec C++ lets you precompile header (`#include`) files and track their dependencies. If files that contribute to the precompiled header are modified, the precompiled header file is marked as requiring a rebuild. Precompiled header files load significantly faster than text header files.

Subgroups
Version 8.0 of the Symantec Project Manager now allows project models to contain arbitrarily nested groups. This feature helps you better structure your projects.

Multiple open projects
Version 8.0 supports multiple open projects. Each open project can be run or compiled.

Option handling
The Project Manager provides access to all project- and translator-specific options through a single menu command. The project options include:

- Project. These options control general project behavior, and build and run settings.

- Project type. These options control project type (application, shared library, static library) and specify the target name.

- Linker. These options control which linker to use when building, and the various linker options.

- Compiler. These options control specific behaviors of the C and C++ compilers.

Named sets of project and translator options
The Symantec Project Manager allows you to define multiple options sets for each project. These sets can be used to group together various option configurations. For example, you can define options sets for the different stages of your project's development.

Project window

The Project window displays information about the project's organization, status, information, and so on. Now there is more flexible control of this display. For example, file display can be organized hierarchically in groups. Other highlights are described below.

Drag and drop

The Symantec Project Manager fully supports drag and drop. Files may be added to and removed from a project by dragging files and folders to and from the Finder and other projects. For example, you can add an entire hierarchy of sources to a project, while preserving the Finder's organization of the files.

New display options

These options control the display in the Project window. This display includes debugging and make status, immediate group owner of source file, source translator, type of each entry (for example, source file, precompiled header source, group), full source path, last known modification date, and code and data size.

Class Browser

Symantec C++ 8.0 provides a Class Browser and Editor designed for object-oriented program development. These tools facilitate the design and maintenance of C++ projects by allowing you to directly view the project's class hierarchy and data and function members. The ability to browse in and edit pre-existing class hierarchies is especially useful when attempting to understand the structure of unfamiliar source code. The Class Browser has the following features:

- Alphabetic or hierarchical display of classes
- Display of data members and member functions
- Display of source code for class and function definitions
- Full integration with the Symantec C++ for Power Macintosh Editor

Debugger

Symantec C++ 8.0 includes a source-level debugger that can be used to debug applications or shared libraries that are built using the Symantec Project Manager. The debugger uses browser-style windows to display information about the process currently being debugged. One new feature is a Stack Crawl pane that contains a list of all stack frames for the program counter location in your code.

New editing features

The Editor includes the following enhanced capabilities:

- Split window: Editor windows can be split to view different parts of a source file in separate panes.

- Pop-up procedure list: This easily accessible listing of user-defined symbols and markers from C/C++ sources, including functions, class definitions, `#pragma` marks, `enum` declarations, and `typedef` definitions. Selecting an item in the pop-up menu takes you to the marker or declaration of the symbol.

- Syntax highlighting: The Editor provides the ability to configure color and style highlighting of language elements. It supports syntax highlighting in C, C++, AppleScript, MPW Shell Script, and Pascal.

- C/C++ mode: Automatic indenting is based on user preferences.

Delimiter matching

The Editor supports delimiter matching. If you double-click on any delimiter (parentheses, brackets, or braces), it finds a matching delimiter. Similarly, if you double-click on a string constant, or comment delimiter, the Editor finds the next instance of that delimiter.

New search options

The Symantec C++ Editor provides powerful multi-file search support, including the ability to:

- Search the front window or all open windows
- Search selected or all files in the current project
- Search within a current selection
- Batch and incremental search
- Search headers included in the precompiled header

Scripting support

The Symantec Project Manager is fully scriptable and recordable under Apple's Open Scripting Architecture (OSA). Scripts can be specified on either an application-wide or project-specific level and can be executed by selecting a single item from the **Scripts** menu.

Worksheet window

The Symantec Project Manager provides a generalized Worksheet window that permits communication with ToolServer and SourceServer. This communication allows you to use Macintosh Programmer's Workshop (MPW) tools and Projector services.

Visual Architect (VA)

Visual Architect, an improved version of Symantec's visual interface development tool, runs native on the PowerPC.

Online documentation

THINK Reference provides on-line hypertext reference for Toolbox routines, language usage, and the THINK Class Library (TCL). You can look up specific information in *THINK Reference* from within the Symantec Project Manager.

Symantec C++ 8.0 includes the software for Symantec C++ 7.0.5. The documentation for Symantec C++ 7.0 is included on the CD-ROM.

Prerequisites for Using Symantec C++

This book assumes that you know, or are learning how to program in, C or C++. You should also be familiar with the Macintosh and its operating system. If you are planning to write Macintosh applications, you should be familiar with the Macintosh Toolbox as described in *Inside Macintosh*.

About This Manual

This manual contains six parts: Welcome to Symantec C++ for Power Macintosh, Creating an Application, Learning by Example (Tutorials), Symantec Project Manager Reference, TCL and VA Reference, and Appendixes. A chapter-by-chapter summary is provided below.

Conventions in the User's Guide and Reference

This book uses the following typographic conventions:

- Names of menus and commands are in **boldface**.
- Names of files, code fragments, resource names, function names, variables, and information you type appear in `typewriter` face. Metanames appear in *italic*.
- All numbers are decimal numbers. Hexadecimal numbers are written in C notation, that is, `0x3EFA`, instead of in Pascal notation (`$3EFA`).
- Keys you press at the same time are shown as follows: Shift+F2, Option+F, or Ctrl+F3. Please note that even though the letter keys are listed in uppercase, do not hold down the Shift key when executing these key combinations unless the Shift key is listed as part of the combination.

Parts One through Three

Parts One through Three describe the basic process of programming in the Symantec C++ environment.

Part One: Welcome to Symantec C++ for Power Macintosh
Part One provides an introduction to the book and product.

Chapter	Description
1. Overview	This chapter.
2. Introducing Symantec C++ 8.0	Describes the basic concepts for programming in the Symantec C++ environment. It describes the role each tool plays in the process of writing an application and gives a preview of the steps involved in creating a typical THINK Class Library (TCL) application using Visual Architect (VA).

Part Two: Creating an Application in Symantec C++
Part Two takes the user through the basic process of creating an application with Symantec C++ and provides an overview of the THINK Class Library and Visual Architect.

Chapter	Description
3. Starting a Project	Explains the fundamentals of working with the Symantec Project Manager. It describes the steps involved in creating projects and performing basic manipulations on project entries.
4. Editing a Project's Code	Covers the details of editing and compiling application code.
5. Viewing and Editing Classes	Describes a tool for viewing and editing a project's class hierarchy, data members, and function members.
6. Using the Debugger	Describes the process of compiling, linking, and debugging an application using the Symantec Project Manager and the symbolic debugger.

Chapter	Description
7. Creating a User Interface with VA	Describes the basic steps in creating an application using Visual Architect.
8. Advanced Topics	Describes other development tools and options including options sets, precompiled headers, scripting, source code control (SourceServer), and ToolServer.

Part Three: Learning by Example (Tutorials)
Part Three provides a set of tutorials which, when completed, demonstrate the steps necessary to produce a single application.

Chapter	Description
9. Tutorial Introduction	Describes learning opportunities in the tutorials.
10. Tutorial: Hello World	Shows how to build a simple application in Symantec C++ that uses both the ANSI C and IOStreams libraries.
12. Tutorial: MiniEdit	Shows how to use more of the advanced features of the Symantec Project Manager.
13. Tutorial: Object Bullseye	Shows how to use the Symantec Project Manager debugger.
14. Tutorial: Vector	Shows how to use templates with Symantec C++.
15. Tutorial: Beeper	Presents a basic tutorial for Visual Architect.
16. Tutorial: Process Monitor	Presents a more elaborate tutorial demonstrating many of the features and techniques involved in programming with Visual Architect and the Symantec Project Manager.

Parts Four through Six

Parts Four through Six contain reference information for Symantec C++.

Part Four: Symantec Project Manager Reference

Part Four provides a complete reference for the Symantec Project Manager, including windows, menus, commands and options.

Chapter	Description
16. The File Menu	Describes the commands that let you create new projects and open existing ones.
17. The Project Window	Describes the information that can be displayed in the project window, including project organization and status information.
18. The Project Menu	Describes all the commands and options associated with a project.
19. The Editor Window	Describes ways to customize editor preferences, including general settings, syntax formatting, and function pop-up.
20. The Edit Menu	Describes the commands for editing project code.
21. The Search Menu	Describes the commands that let you find information in your project files.
22. The Class Browser Window	Describes how to view and edit class information.
23. The Build Menu	Describes all the commands used to create an application, including compiling, linking, syntax checking, and creating executable code.

Chapter	Description
24. The Debugger Windows	Describes the windows for displaying debugging information, including the Main debugging window, Debug Browser window, Data window, Control palette, Source window, and Log window.
25. The Debugger Menus	Describes the commands associated with the debugger **File**, **Edit**, **Debug**, **Source**, **Data**, and **Windows** menus.
26. Windows Menu	Describes the commands for configuring Symantec Project Manager windows, including size and position, and commands to display the Build Errors, Search Results, Class Browser, and Worksheet windows.

Part Five: TCL and VA Reference

Part Five provides a full reference to Think Class Library (TCL) and Visual Architect (VA). Basic concepts for TCL/VA are covered, as is programming with TCL and using Object I/O. A full menu-by-menu reference for VA is also included. This part does not include the TCL class library description, global variables, or library routines. These can be found in the online *THINK Reference.*

Chapter	Description
27. TCL and VA: Basic Concepts	Describes the basic concepts of Visual Architect and the THINK Class Library that are needed to create an application.
28. Programming with TCL	Describes some of the basics of how the THINK Class Library works and how to use it to build an application.
29. Visual Architect File Menu	Describes the commands for manipulating Visual Architect resource files.

Chapter	Description
30. Visual Architect Edit Menu	Describes the standard Macintosh editing commands, as well as commands for manipulating resource objects.
31. Visual Architect View Menu	Describes the commands to manipulate the views created in Visual Architect.
32. Visual Architect Pane Menu	Describes the commands to change characteristics of the panes in your views.
33. Visual Architect Options Menu	Describes the commands for setting the behavior of the View Editor in Visual Architect.
34. Visual Architect Tools Menu	Describes the commands to add panes to your views.
35. VA Symantec Project Manager Menu	Describes the commands to control the generation of code by Visual Architect for inclusion in your project.

Part Six: Appendixes
This part contains appendixes to the User's Guide and Reference.

Appendixes	Description
Appendix A: Linker Error Messages	Describes the error messages generated by the Symantec Linker.
Appendix B: Debugger Error Messages	Describes the error messages generated by the Symantec Debugger.

Electronic supplemental information (ESI)

In addition to the material in this User's Guide, several topics are covered online. This presentation includes supplemental information on such topics as porting applications to Symantec C++ 8.0, some background information about programming for the Power Macintosh, and information about other features.

Installing Symantec C++

This section describes the procedure for default installation. For information about custom installation refer to the online electronic supplemental information.

Read the license agreement

Before installing Symantec C++, you should read and become familiar with the terms of the license agreement.

Send in the registration card

Remember to fill out and send in your registration card. To receive technical support, information about upgrades or news about special promotions, you must be a registered user.

Read the ReadMe files

Please make sure to read any files named `ReadMe`. These files contain information that was not available at the time the manuals were printed.

Installing all of Symantec C++

To load the default installation of Symantec C++ drag the folder named `Symantec C++ for Power Mac` from the CD-ROM onto your hard disk. For a custom installation of Symantec C++, consult the online electronic supplemental information.

For information on installing Symantec C++ 7.0.5, see the online documentation for this product on CD-ROM.

What To Do Next

This section provides a guide to using this book according to your level of knowledge.

Steps for the user new to Power Macintosh development

You should read "About Programming in C++ for the Power Macintosh" in the online electronic supplemental information. You should also read the chapters in Part Two.

Steps for the user new to Symantec C++

You should read the chapters in Part Two, and work through the tutorial chapters in Part Three to become proficient with Symantec C++.

Steps for the user new to the Macintosh

You should read "Macintosh Conventions" in the online electronic supplemental information and all of the Symantec C++ 8.0 documentation.

Introducing Symantec C++ 8.0 ◆ 2

This chapter introduces the Symantec C++ programming environment. It provides an overview of the THINK Class Library (TCL) and Visual Architect (VA), as well as of the processes involved in creating an application.

Programming with Symantec C++

The Symantec Project Manager is the crux of the Symantec C++ development environment. Unlike traditional, command-line development environments, Symantec C++ provides integrated components for accomplishing your development tasks, such as editing, compiling, linking, class browsing, and debugging. The Symantec Project Manager is the central location from which you access the integrated tools that make up Symantec C++.

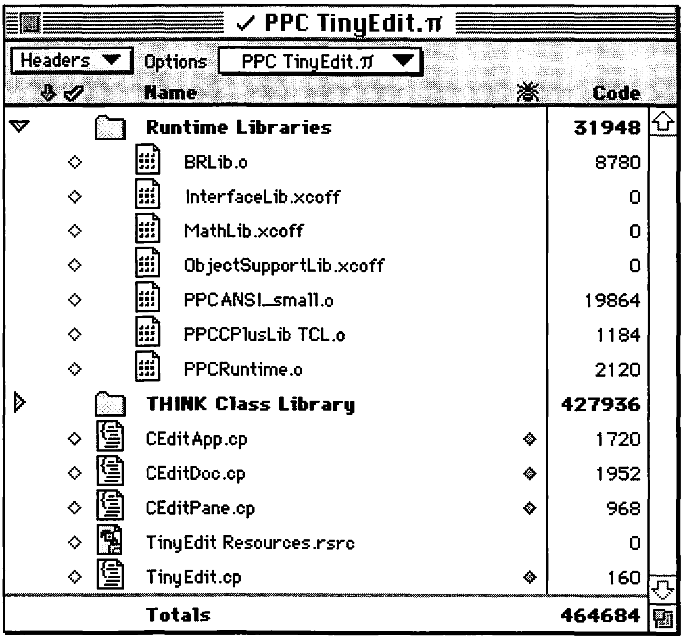
One particularly important component of Symantec C++ is Visual Architect. Because Visual Architect is integrated into the THINK Class Library, you can develop an application's user interface along with its underlying code in a graphical, interactive environment. The Symantec Project Manager can compile files generated by Visual Architect, even when Visual Architect is open.

The following sections outline the steps involved in using Symantec C++ to create a typical application based on the THINK Class Library. You may want to create something other than a standard Macintosh application—for example, an IOStreams-based application or a library. By reviewing the steps outlined here, you will become familiar with the fundamental tasks you can perform with Symantec C++.

Starting a project

The first step in application development with Symantec C++ is creating a project using the Symantec Project Manager. A project is a collection of files that, when built, creates a target application (or library). A typical project contains source, header, resource, and documentation files, as well as binary libraries and other projects.

The management information for a project is contained in the Project file, which customarily has a `.π` suffix. The information in the Project file helps the Project Manager determine how to process each of the project's component files. Whenever a Project file is opened in the Project Manager, a Project window showing the project components is displayed (Figure 2-1).



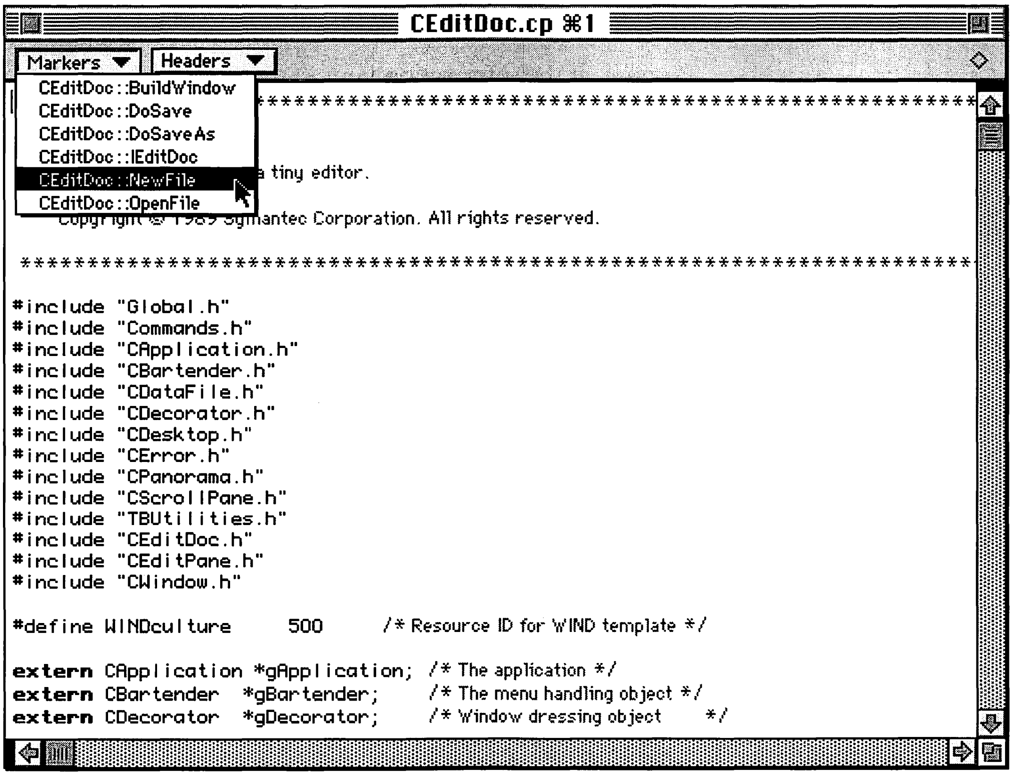
	Name		Code
▽	**Runtime Libraries**		**31948**
◇	BRLib.o		8780
◇	InterfaceLib.xcoff		0
◇	MathLib.xcoff		0
◇	ObjectSupportLib.xcoff		0
◇	PPCANSI_small.o		19864
◇	PPCCPlusLib TCL.o		1184
◇	PPCRuntime.o		2120
▷	**THINK Class Library**		**427936**
◇	CEditApp.cp	◆	1720
◇	CEditDoc.cp	◆	1952
◇	CEditPane.cp	◆	968
◇	TinyEdit Resources.rsrc		0
◇	TinyEdit.cp	◆	160
	Totals		**464684**

Figure 2-1 Project window

When you create a project, you have the option of basing it on a pre-existing project model. Each model predefines a generic set of capabilities for an application. Perhaps the most versatile of these models is VA Application, which includes TCL resources as well as Visual Architect. Other project models allow you to create IOStreams- and ANSI-based applications.

Editing application code

The Symantec Project Manager contains an integrated text editor that is customized for working with C++ and C source code. Editor windows are opened by double-clicking source files in the Project window. To facilitate working with source files, you can customize the Editor to provide a list of functions in the current file, as shown in Figure 2-2.



```
*****************************************************

a tiny editor.

   antec Corporation. All rights reserved.

 ****************************************************************************

#include "Global.h"
#include "Commands.h"
#include "CApplication.h"
#include "CBartender.h"
#include "CDataFile.h"
#include "CDecorator.h"
#include "CDesktop.h"
#include "CError.h"
#include "CPanorama.h"
#include "CScrollPane.h"
#include "TBUtilities.h"
#include "CEditDoc.h"
#include "CEditPane.h"
#include "CWindow.h"

#define WINDculture      500      /* Resource ID for WIND template */

extern CApplication *gApplication; /* The application */
extern CBartender   *gBartender;   /* The menu handling object */
extern CDecorator   *gDecorator;   /* Window dressing object    */
```

Figure 2-2 Editor window, showing the pop-up list of functions in the current file

The Editor is tightly integrated with the Project Manager. For example, when errors are encountered during compilation of your source code, you can automatically choose to display an Editor window with the offending line of code highlighted.

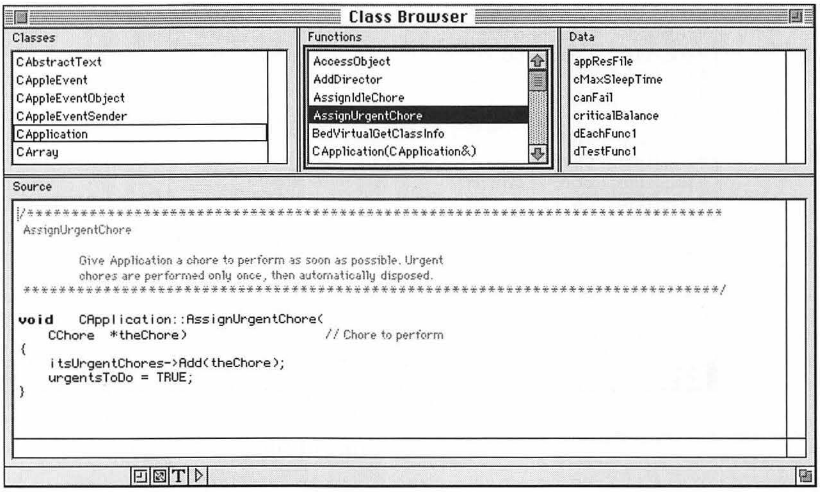
Building an application

While building an application, the Symantec Project Manager calls on a battery of translators to process the various project components. For example, the PowerPC C++ compiler processes the C++ source files and Symantec Rez converts resource description files to resource files. After all the components have been translated successfully, the Symantec Project Manager calls on the appropriate linker to produce the application file, which can then be run from the Project Manager or from the Finder.

Because of the information kept in the Project file, the Project Manager automatically is able to keep track of the files that have changed and of any files that refer to changed files. As a result, the Project Manager processes only those files that need updating.

Viewing classes

Once the source code files have been compiled, you can use the Symantec Class Browser to examine an application's class hierarchy and to edit the contents of classes (Figure 2-3).



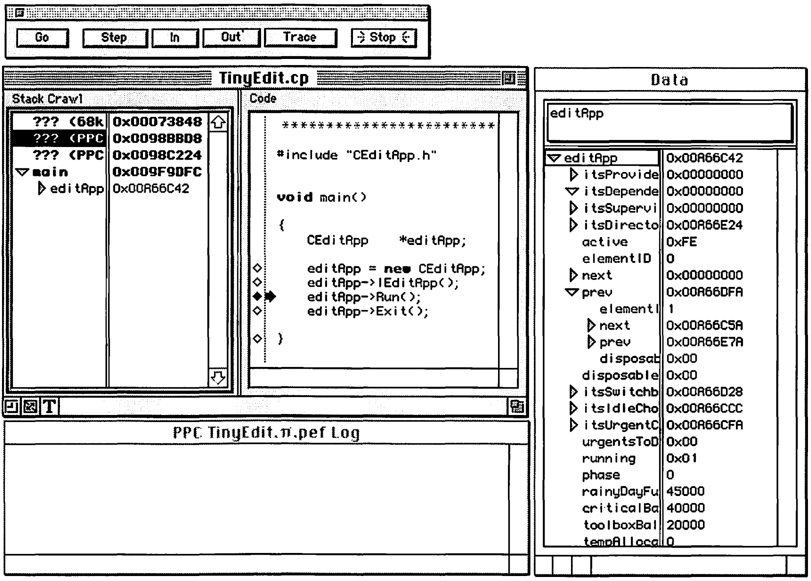
```
/******************************************************************************
 AssignUrgentChore

        Give Application a chore to perform as soon as possible. Urgent
        chores are performed only once, then automatically disposed.
 ******************************************************************************/

void    CApplication::AssignUrgentChore(
    CChore   *theChore)                    // Chore to perform
{
    itsUrgentChores->Add(theChore);
    urgentsToDo = TRUE;
}
```

Figure 2-3 Class Browser window

Work performed in the Class Browser is complementary to the class viewing and editing performed with the text editor.

Testing an application

Symantec C++ provides powerful source-level debugging capabilities to help you test and debug applications. The Symantec Debugger, shown in Figure 2-4, allows you to control execution of code as well as to display and modify data values.



```
??? (68k      0x00073848
??? (PPC      0x0098BBD8
??? (PPC      0x0098C224
▽ main        0x009F9DFC
  ▷ editApp   0x00A66C42
```

```
*************************

#include "CEditApp.h"


void main()

{
    CEditApp     *editApp;

    editApp = new CEditApp;
    editApp->IEditApp();
    editApp->Run();
    editApp->Exit();

}
```

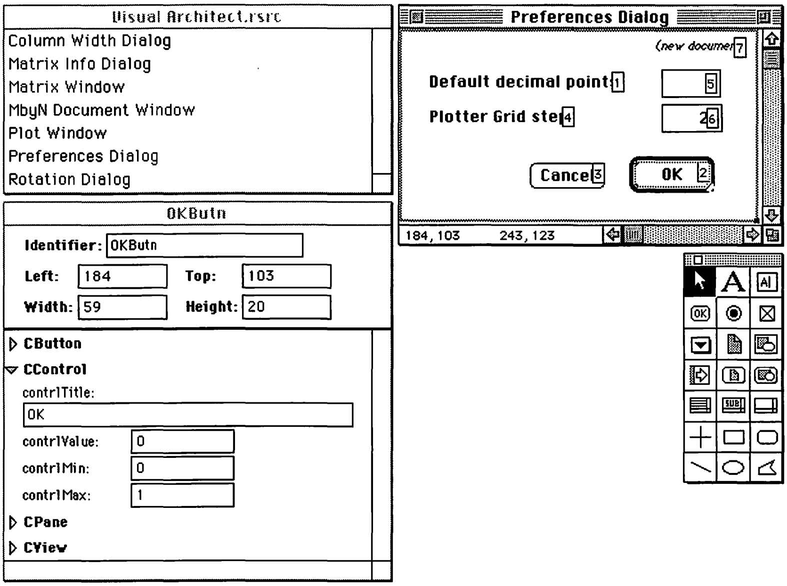
```
editApp

▽ editApp          0x00A66C42
    ▷ itsProvide   0x00000000
    ▽ itsDepende   0x00000000
    ▷ itsSupervi   0x00000000
    ▷ itsDirecto   0x00A66E24
      active       0xFE
      elementID    0
    ▷ next         0x00000000
    ▽ prev         0x00A66DFA
        elementI   1
      ▷ next       0x00A66C5A
      ▷ prev       0x00A66E7A
        disposab   0x00
      disposable   0x00
    ▷ itsSwitchb   0x00A66D28
    ▷ itsIdleCho   0x00A66CCC
    ▷ itsUrgentC   0x00A66CFA
      urgentsToD   0x00
      running      0x01
      phase        0
      rainyDayFu   45000
      criticalBa   40000
      toolboxBal   20000
      tempAlloca   0
```

Figure 2-4 Symantec Debugger

The Symantec Debugger is tightly coupled with the Project Manager, making it easy to cycle between debugging and editing of source files.

Constructing a user interface with Visual Architect

Visual Architect provides a graphical, interactive environment for simplifying the creation of resources and the use of THINK Class Library classes. Visual Architect is illustrated in Figure 2-5.



Visual Architect.rsrc

Column Width Dialog
Matrix Info Dialog
Matrix Window
MbyN Document Window
Plot Window
Preferences Dialog
Rotation Dialog

OKButn

Identifier: OKButn

Left: 184 Top: 103

Width: 59 Height: 20

▷ CButton

▽ CControl

contrlTitle: OK

contrlValue: 0

contrlMin: 0

contrlMax: 1

▷ CPane

▷ CView

Preferences Dialog

(new docume 7

Default decimal point 1 — 5

Plotter Grid ste 4 — 2 6

184, 103 243, 123

Figure 2-5 Visual Architect

Visual Architect generates both resource and source files that are automatically incorporated into a project. Because it is designed to be a component in iterative development, you can use Visual Architect at any stage in an application's development. The following section, "The THINK Class Library and Visual Architect," provides a brief introduction to Visual Architect, which you use to create the application framework.

Using additional tools

Several additional tools are included with the Symantec Project Manager, for use in creating an application.

AppleScript capability

The Symantec Project Manager lets you use AppleScript to record and control complex operations.

SourceServer
Apple's SourceServer application assists in version tracking with large programming projects.

ToolServer
Apple's ToolServer lets you access Macintosh Programmer's Workshop (MPW) tools from the Symantec Project Manager.

The THINK Class Library and Visual Architect

This section provides a conceptual overview of what is involved in creating a typical THINK Class Library application using Visual Architect. It discusses how the THINK Class Library and Visual Architect interact and how Visual Architect streamlines the application development process.

Overview of the THINK Class Library

The THINK Class Library is a collection of C++ classes designed to implement standard Macintosh applications. These classes handle Macintosh functions such as menu commands, window updates, event dispatching, operating system calls, memory management, Clipboard maintenance, and printing. To create an application, you can use existing classes in the THINK Class Library in which these lower-level interfaces have already been established, rather than developing code from scratch using Macintosh Toolbox and Operating System calls.

The THINK Class Library is organized into three distinct, interacting structures: the class hierarchy, the visual hierarchy, and the chain of command.

Class hierarchy
The class hierarchy is the set of all the classes that make up the THINK Class Library. The class hierarchy is organized around the concept of inheritance. It contains a set of base classes from which other classes inherit their behavior (member functions) and attributes (data members).

Visual hierarchy

The visual hierarchy describes the organization of all visible entities, such as windows and buttons, in a given application. The visual hierarchy describes all the views that the THINK Class Library contains. A TCL view is an object descended from a class that is used for implementing objects with visual representations. Views respond to commands involving the mouse and can also be links in the chain of command.

The visual hierarchy is based on the concept of enclosure. Everything you see on the screen belongs to—is enclosed by—another visual entity. This is in contrast to the class hierarchy, which is based on derived class relationships.

Chain of command

The chain of command specifies both the objects in an application that handle specific commands (such as menu item choices) and the order in which those commands are handled. Because you are responsible for choosing this assignment of objects, you need to decide the level of abstraction at which you want to handle a specific command. For example, the **Save** and **Save As** commands are handled at the document level, whereas the **New**, **Open**, and **Quit** commands are handled at the application level.

Interaction of the structures

The THINK Class Library converts Macintosh events into calls to member functions defined by the classes in the class hierarchy. Some member functions handle events that affect the visual hierarchy, including mouse clicks, keyboard events, Activate events, and Update events. Other member functions handle requests that affect the chain of command, such as for an object to open a file. The latter type of request typically is the result of a menu choice.

Creating a THINK Class Library application

To create an application that uses the THINK Class Library, you derive new classes from existing library classes. These new classes should implement only the unique parts of an application, because generic application behavior is already handled by the base classes in the library.

Typically, you derive a number of classes. You need a class for your unique application object; a "document" class to manage windows and files; and one or more classes for managing communication between or among windows, panes, files, and menu commands. These last classes, in other words, deal with the user interface.

In addition, you need to define both a menu structure to contain the commands that the application implements and the linkage between menu commands and actions. Actions are performed in response to the user choosing a particular menu command.

Finally, virtually all Macintosh applications make use of resources. TCL-based applications are no exception; they require standard resources to function. These resources are found in the file `TCL Resources`. You must add these resources to your project, either by copying them into the `.rsrc` file that contains your own additional application-specific resources, or by starting from a copy of one of the TCL demos, or from one of the VA Application models. For more details on the THINK Class Library, see Chapter 28, "Programming with the THINK Class Library." For a discussion of VA, see Chapter 27, "TCL and VA: Basic Concepts."

Overview of Visual Architect

Visual Architect is a powerful development tool that allows you to rapidly create applications using the THINK Class Library. Visual Architect streamlines the process of creating, editing, and connecting the classes, menus, commands, and other resources needed by an application. This section introduces this development tool.

The role of Visual Architect

Visual Architect automatically generates source code files and TCL resources. The source code files contain definitions and declarations for the classes created for an application. The resources contain information needed to initialize window and pane classes according to your specification, as well as the menus and their associated commands. The source code files are standard C++ `.cp` and `.h` files and, as such, can be opened and edited using the Symantec Project Manager. The resources can be created and edited through Visual Architect itself.

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

Visual Architect is designed to work directly with the Symantec Project Manager. When you use the Symantec Project Manager to start a new VA project, it automatically creates a file named `Visual Architect.rsrc`.

Note

> A VA project also contains another resource file, `Project Resources.rsrc`. This file initially contains resources needed by the THINK Class Library, but not by Visual Architect. You can add more resources to either file using Symantec Rez or ResEdit.

Creating and modifying classes

One of Visual Architect's most powerful aspects is how it facilitates the implementation of views and the classes constructed from these views with the help of an interactive graphical environment. Visual Architect automatically derives classes from views using the THINK Class Library, as well as defining the classes' data members and member functions. Visual Architect also generates source code files that contain these data and function members; thus, you do not need to determine the class definitions required for a specific application. Visual Architect adds these source files to your Symantec C++ project.

In general, the classes that you construct fall into one of two categories:

- VA views (director or document classes)
- Panes

You can add, delete, and modify derived classes in either of these categories.

Visual Architect allows you to change the attributes of views and panes through dialog boxes, thus enabling you to avoid time-consuming hand coding.

Working with Visual Architect

This section briefly describes VA views, as well as how to add commands and Balloon Help to views, trying out a view, and modifying VA code.

Visual Architect provides several predefined views for implementing common graphical representations, such as document windows, dialog boxes, floating windows, splash screens, subviews, floating tool palettes, and basic windows. The implementation of a specific VA view is based on a particular THINK Class Library class.

The most important view defined by Visual Architect is the Main Window view. A Main Window view typically is displayed in an application when the user chooses **New** or **Open** from the **File** menu. It displays the contents of an associated file and serves as the focus of attention for a user.

Defining commands associated with views and menus

To process certain user actions (for example, mouse clicks or keyboard events), the THINK Class Library predefines many frequent actions, such as closing a window, saving a file, quitting an application, and changing text attributes. Having these actions predefined means you can focus on creating the commands that are unique to your application. As a result, coding time is reduced.

To implement either a predefined action or a new action using Visual Architect, you need to indicate the class or classes that will handle the action. If the action is to open a view, you specify that view and Visual Architect generates the necessary code. Otherwise, you indicate those classes that need to respond to the action and Visual Architect then generates an empty member function. Later, you can insert code in the member function that handles the action.

Adding Balloon Help
Visual Architect provides a convenient way to add Balloon Help to the visual elements of an application. Balloon Help can be added to panes by opening the Balloon Help window and typing the text you want in the balloon. Because user interface elements can exist in up to four different states—such as enabled and disabled—when the application is running, you can define up to four different balloons for each element.

Trying out a view
With Visual Architect, you can examine how an interface works before you generate source code, compile it, and run the application. When you try out a view, it appears exactly as it would in the running application. Furthermore, you can interact with the view to a limited extent. For example, you can scroll, resize, and reposition the view. All the view's elements—such as pop-up and tear-off menus, dialog text fields with type constraints, scrolling edit text, custom buttons, and button groupings—are active as well.

Trying out views lets you see 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 never accomplished in one step. Most often, you design some of the user interface elements in Visual Architect, hand code in the Symantec Project Manager, compile, run, and inspect the project. At that point, you would return to Visual Architect to make changes and start the cycle again.

Due to the interactive nature of programming, Visual Architect does not force you to live with the code it generates "as is." Most of the code it generates is well-commented C++ skeleton code. Any changes you make to this code by hand are not overwritten in subsequent code-generation steps.

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 types: a lower-level class, reserved for Visual Architect, and an upper-level class, reserved for custom programming. The first time Visual Architect generates source code for a graphical element, it generates the code for both classes in separate files.

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 is displayed here. You should not modify this code.

The upper-level class is derived from the lower-level class. To customize the skeleton code, you add member functions, additional data members, and so forth to this class. Member functions that you add manually to the upper-level class often override or expand on the corresponding lower-level class member functions. Visual Architect writes to the upper-level class 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.

To summarize, if you plan to create an application that uses standard Macintosh interface elements such as windows, menus, and so on, you should use the Visual Architect/THINK Class Library (VA/TCL) development environment. For further discussion of Visual Architect and the THINK Class Library, see Chapter 27, "TCL and VA: Basic Concepts" and Chapter 28, "Programming with the THINK Class Library."

Symantec C++

Creating an Application

Part Two

Starting a Project ◆ 3

Creating an application or library in Symantec C++ is a multistep process involving three primary tools: the Symantec Project Manager, the Symantec Debugger, and Visual Architect. This part of the guide explains the fundamentals of working with the Symantec Project Manager.

Starting a new application or library with the Symantec Project Manager begins with the creation of a project. This chapter describes the steps involved in creating projects and performing some basic manipulations on project entries.

What Is a Project?

A project is a set of files that, when assembled by the Symantec Project Manager, produce an application or library. In a typical project, these project entries consist of C and/or C++ source files and their associated header files, object libraries, resources, other projects, and documentation files. A single project generally is used to create a single target.

Note

> The application or library that the Symantec Project Manager builds from a project is referred to as the project's target.

The central element of a project is the Project file. By convention, the Project file has the suffix `.π` (period, Option-P). The Project file contains all information necessary for management of the project, such as locations of the project entries, and additional information such as compiler options and browser tables. The Project file also contains the object code for the target and the Debugger symbol tables.

In general, most project entries specific to a project, together with the Project file, are kept in a folder referred to as the project folder. However, a project's entries do not all have to reside in the project folder. For example, only one copy of general-purpose libraries, which all projects need, is kept in the folder. In addition, a project may include project entries that are located in other project folders, thus permitting sharing of code and resources.

By organizing a target's files as a project in this manner, the Symantec Project Manager can assume full management responsibility. In contrast to traditional "make" systems, this strategy frees you from the bookkeeping involved in accessing project entries and building the target. Because the Project Manager keeps track of all project entries in the project, the features of the Symantec Project Manager are smoothly integrated. For example, if an error occurs during compilation, you can open a window containing the source code with the questionable line of code highlighted in a single step.

Also, the Symantec Project Manager automatically determines those project entries that need to be rebuilt following changes to any project entry(ies). After a header file is changed, for example, the Symantec Project Manager knows to rebuild all source code files that include the header file.

Project contents

The different types of entries that can be included in a project are described in this section. Each type is handled differently by the Symantec Project Manager when the target is built and when the project entry is accessed from within the Project Manager. The Project Manager uses filename extensions to identify the type of project entry. You can change this mapping of extensions to entry types in the **Project Options** dialog box, as described in Chapter 18, "The Project Menu."

Source files

The Symantec Project Manager can process C (`.c`), C++ (`.cp` or `.cpp`), and resource directive or Rez (`.r`) source files. A single project can contain both C and C++ source files. These files are all text files.

Precompiled header files

The Symantec Project Manager allows you to create precompiled headers and include them in your source files. Precompiled headers are "processed" versions of header files and are in a format that the Project Manager can load significantly faster than text header files.

Resource files

For your target to access resources, resource (`.rsrc`) files, such as those created by ResEdit and Resorcerer, can be included in your project. The resources from these files are added to the target when it is built.

Libraries

The Symantec Project Manager allows you to include binary libraries in your project. Some examples of libraries provided with the Symantec Project Manager are C and C++ libraries (such as `PPCAnsi`, `PPCIOStreams`, and `PPCUnix`) and Macintosh libraries (such as `InterfaceLib.xcoff`, `QuickTimeLib.xcoff`, and `AppleScriptLib.xcoff`). The Project Manager allows you to create your own libraries, which can in turn be included in other projects.

Projects

The Symantec Project Manager also allows you to include other projects in a project. Including projects lets you group together sets of related project entries and access all included projects' entries within the Project Manager. Included projects are built to completion when the project containing them is built. Thus, you can develop a suite of applications by having one project for each application and one additional project that includes all the individual projects; the entire suite of applications can be built with one command. Further, if the target of an included project is a library, the library is linked into your project.

Documentation files

You can add any documentation files to the project to make them readily accessible during development. These files will neither, however, be included in the final target nor be involved in any way with building. By including these files directly in the project, you make them always available for reference and modification. You can use any application to create these files.

Groups

To better organize project entries within your project, the Symantec Project Manager allows you to create groups. Groups are similar in concept to Finder folders. By placing your project entries into groups, you make it easier to locate individual project entries, especially with a large project. Like folders, groups can be nested. The placement of project entries into groups has no effect on the final target. Further, the location of a project entry in a group has no bearing on the file's location on the disk.

Organizing files and folders

When the Symantec environment is installed, a specific plan for folder organization is followed. This folder plan is set up to allow the Symantec Project Manager to quickly and unambiguously locate your project's entries. Specifically, the Project Manager looks for your project's entries in one of two locations, the system tree or the project tree.

The system tree

The Symantec Project Manager folder, along with all the subfolders within it, is called the system tree. The Symantec Project Manager folder is the folder that contains the Symantec Project Manager (usually the `Symantec C++ for Power Mac` folder). The Symantec Project Manager treats all files in all folders within the Symantec Project Manager folder as if they were in the same flat folder.

The project tree

The project folder, along with all subfolders it contains, is called the project tree. The project folder is the folder that contains the Project file. The Symantec Project Manager treats all files in all folders within a project folder as if they were in the same flat folder.

Typically, a Project file resides in a project folder along with all project entries specific to the particular project. The folder may also contain other folders so as to group together related resource files and header files. Setting up your project entries in this way helps reduce the time it takes the Symantec Project Manager to search for files, and reduces the likelihood of confusion due to duplicate file names. You can expect to have many project folders.

Note

The project folders themselves must be placed outside the system tree. Otherwise, file search times are increased.

When you first add a file to a project, the Symantec Project Manager notes the tree to which the file belongs. Thus, you can move files in and out of folders and create and rename folders without having to tell the Symantec Project Manager exactly where the files are located. If you move files later on, the Symantec Project Manager first looks in this tree.

To hide the contents of any folders within the system and project trees from the Symantec Project Manager, you can enclose the name of the folder in parentheses. The only exception occurs when the name of the folder, excluding the parentheses, matches the project name; in this case, the contents of the folder are visible. For example, a folder named `(Hidden.π)` in the system tree would be hidden from all projects except one named `Hidden.π`.

Models and Projects

This section describes the use of templates, or project models, in creating a project. It discusses the different project models and how to create projects based upon them.

This section assumes you've already correctly installed the Symantec Project Manager. Before working with the Symantec Project Manager, you should create a common folder to contain your specific project folders. You can name this common folder anything you like, such as `My Projects`, and you can place it anywhere you like as long as it is outside of the system tree (see the previous section for more information).

Choosing the project model

When creating a new project, you must determine a project model for it. Project models are templates that determine those project entries that are to be initially added to a project and those configuration options that are to be initially provided. Using project models reduces the amount of overhead involved with creating projects. You can also create your own project models, as described in Chapter 16, "The File Menu."

The project models supplied with Symantec C++ for Power Mac include VA Application, VA App w/Shared TCL, ANSI C++ (IOStreams), and C++ Mac Application, ANSI C, C Mac Application, and Native MPW Tool. These project models are only briefly described here; see Chapter 16, "The File Menu," for more details.

VA Application and VA App w/Shared TCL project models
These are the project models you choose if you are planning to use Visual Architect and the THINK Class Library in the design and construction of an application's user interface.

Note

> For projects created using Visual Architect and the THINK Class Library, typically you use all of the Symantec C++ for Power Macintosh development tools. This discussion concentrates on projects created with the VA Application project models.

ANSI C++ (IOStreams) project model
Use the ANSI C++ (IOStreams) project model if you are creating a C++ application that uses the standard IOStreams environment for input/output. You can code in this project model without having to create an interface to the Macintosh Toolbox or a user interface.

C++ Mac Application project model
Use the C++ Mac Application project model if you are creating a standard Macintosh C++ application without using the IOStreams library or the THINK Class Library. You will have to manually create all the Macintosh user interface elements, such as windows, menus, and printing.

ANSI C project model
Use the ANSI C project model if you are creating a C application that uses the standard ANSI environment for input/output. You can code in this project without having to create an interface to the Macintosh Toolbox or a user interface.

C Mac Application project model
Use the C Mac Application project model if you are creating a standard Macintosh C application that does not use the ANSI library. You will have to manually create all the Macintosh user interface elements, such as windows, menus, and printing.

Native MPW Tools

Use the Native MPW Tool project model if you are creating a tool to use with the Macintosh Programmer's Workshop. MPW tools can also be used from the Symantec Project Manager with ToolServer.

Empty Project model

Use the Empty Project model when you are providing all the necessary source code and project, library, and resource files for an application. You will have to create all the Macintosh interface elements.

Creating a new project

To create a new project:

1. Launch the Symantec Project Manager from the Finder by double-clicking its icon or by selecting the icon and choosing **Open** from the Finder's **File** menu.

   The **File Open** dialog box opens (Figure 3-1).

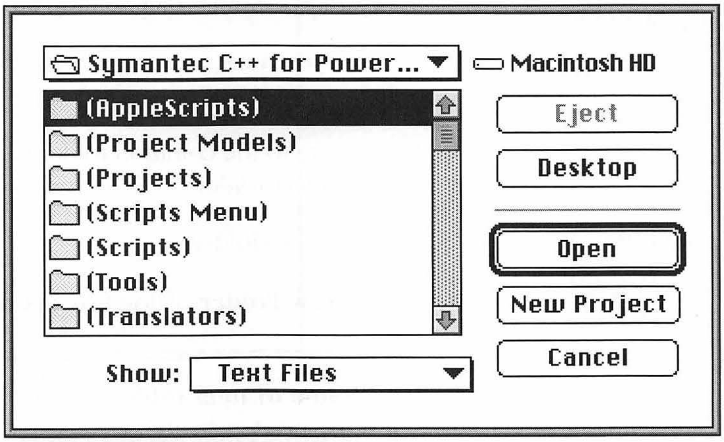


Figure 3-1 File Open dialog box

2. Click New Project.

The **New Project** dialog box opens (Figure 3-2).

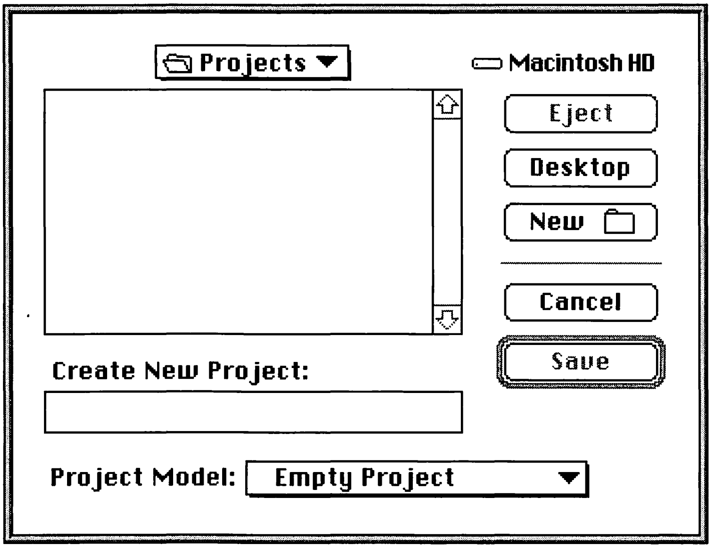


Figure 3-2 New Project dialog box

Next, you must create a project folder for your Project file.

3. Navigate to the common folder in which you want to put the project folder.

4. Click New (folder).

The **New Folder** dialog box opens (Figure 3-3).

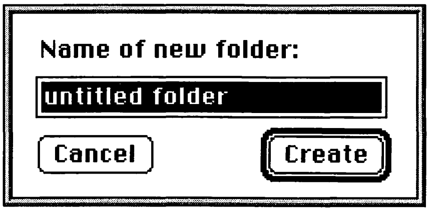


Figure 3-3 New Folder dialog box

5. Enter the name of the project folder and click Create.

A new folder is created. Its title is shown in the pop-up menu at the top of the **New Project** dialog box.

6. Choose the project model from the **Project Model** pop-up menu at the bottom of the **New Project** dialog box.

Note

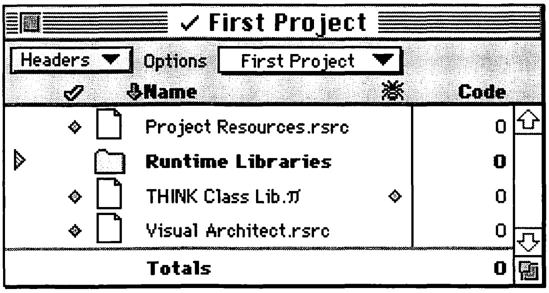
Use one of the VA Application project models if you want to be able to follow all of the remaining chapters in this part of the book.

You can now name and create the actual Project file:

7. Type the name of the project in the Create New Project textbox. By convention, the file should end in `.π` (period, Option-P).

8. Click Save in the **New Project** dialog box.

   The Symantec Project Manager creates the Project file. Depending on the project model, other files may be also created in the project folder. In addition, the appropriate project entries for that model are added to the Project file. When the process is completed, the Project window opens (Figure 3-4).



✓	Name		Code
◇	Project Resources.rsrc		0
	Runtime Libraries		**0**
◇	THINK Class Lib.π	◇	0
◇	Visual Architect.rsrc		0
	Totals		**0**

Figure 3-4 Project window

The Project window is the central element in managing a project. It displays the status of all individual project entries included in your project. The Name column lists the names of all project entries and groups, and the Code column provides their compiled size in bytes.

Note

You can customize the format of the Project window using the Project Window page of the **Project Options** dialog box; see Chapter 18, "The Project Menu," for details.

Adding and removing project entries

The Project file contains various project entries that are added by default when the project is created. The specific project model that you choose determines the default entries that are added to the Project file. To add new project entries to a Project file:

1. Choose **Add Files** from the **Project** menu.

   The **Add Files** dialog box opens (Figure 3-5).

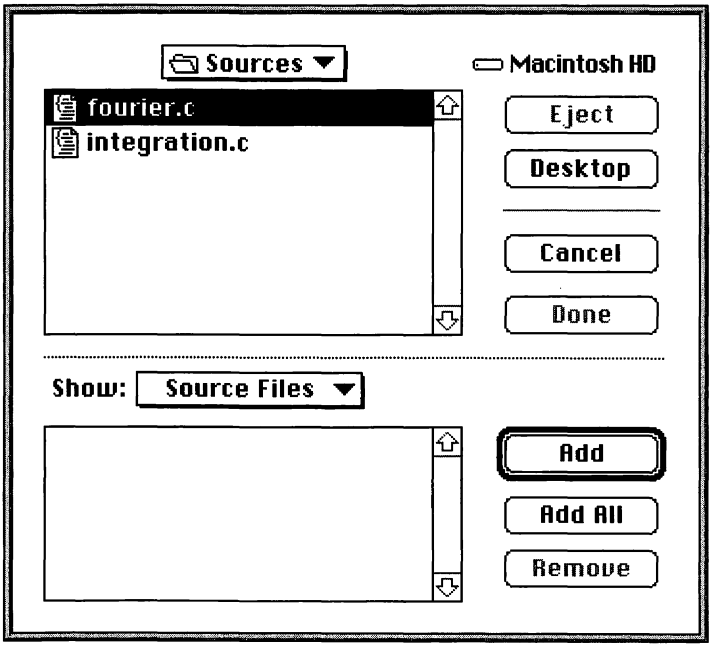


Figure 3-5 Add Files dialog box

You use the **Add Files** dialog box to add any types of files to your project. Note that only those files that have not previously been added to the project are listed in the upper scrolling list.

2. Choose either **Source Files** or **All Files** from the **Show** pop-up menu.

   If **Source Files** is chosen, only files that have an established translator in the project (plus shared libraries and Project files) are listed in the upper scrolling list. If **All Files** is chosen, all files are listed.

Note

For information about specifying translators for different files, see the Extensions Mapping page of the **Project Options** dialog box in Chapter 18, "The Project Menu."

3. Navigate to the appropriate folder and either double-click the name of the file in the upper scrolling list or select the name and click Add.

   The name of the file is added to the lower scrolling list.

4. Repeat step 3 for each additional file you want to add to the project.

5. When you have specified all the files you want to add to the project, click Done.

   The files are added to the project and their names are displayed in the Project window.

To remove a project entry:

1. Select the project entry name in the Project window.

2. Choose **Remove "*filename*"** from the **Project** menu, where *filename* is the name of the selected project entry. You can select multiple project entries in the Project window, in which case the menu item is titled **Remove Selected Items**.

   The project entries are removed from the project and their names are removed from the Project window.

You can also add project entries to a project using drag and drop. To add only files for which a translator has been established (plus shared libraries and Project files) to the project, drag the file icons from the Finder to the Project window. You can also drag a project entry from another Project window.

This is also what happens if you choose **Source Files** from the **Show** pop-up menu in the **Add Files** dialog box. To eliminate this filtering and allow you to add files to the project (as when you choose **All Files** from the **Show** pop-up menu in the **Add Files** dialog box), drag the file icon from the Finder to the Project window while pressing the Command key.

To learn how to drag folders containing files to your project, see the next section.

To remove project entries using drag and drop, drag the entries from the Project window to the Trash.

Creating groups

A group has the same relation to the Symantec Project Manager as a folder has to the Finder: it embraces a collection of related files. Like folders, groups can be nested. For example, the project created using the VA Application project model has several groups. Most of these groups contain related elements of the THINK Class Library (for example, Apple Event classes and control classes) and are located in the THINK Class Library group. In addition, the group Source contains source files generated by Visual Architect. As you add project entries to a project, you should create new groups to help keep these entries organized and help you locate individual entries.

To create a new group:

1. Choose **Add Group** from the **Project** menu to open the **Add Group** dialog box (Figure 3-6).

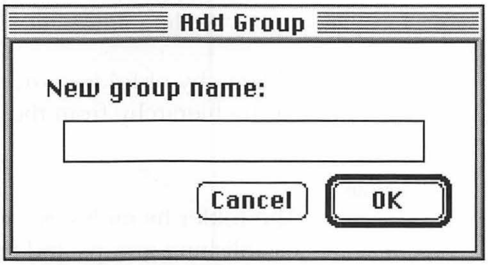


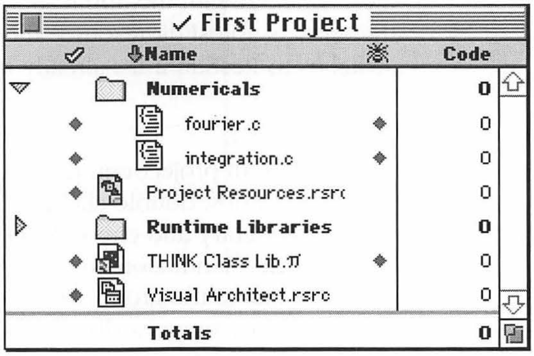
Figure 3-6 Add Group dialog box

2. Enter the name of the new group and click OK.

   The new group is displayed in the Project window.

If you are adding a file or set of files to the group using the **Add Files** command in the **Project** menu, be sure the group is first highlighted in the Project window.

To display the contents of a group, click the small triangle located to the left of the group name (Figure 3-7).


✓ First Project

	✓	Name		Code
▽		**Numericals**		**0**
	◆	fourier.c	◆	0
	◆	integration.c	◆	0
	◆	Project Resources.rsr(		0
▷		**Runtime Libraries**		**0**
	◆	THINK Class Lib.π	◆	0
	◆	Visual Architect.rsrc		0
		Totals		**0**

Figure 3-7 Project window with contents of the group Numericals displayed

To hide the contents of a group, click the small triangle again.

It is also possible to create groups by dragging folders to the Project window. Entire group hierarchies can be added in this process. All the files within a folder are added to the group corresponding to the folder in which they are located. Note that the same rules of filtering apply as when dragging and dropping individual files, as discussed in the section "Adding and removing project entries" in this chapter.

To add a hierarchy of folders and files, drag the folder icon containing the hierarchy from the Finder to the Project window.

Note

The folder hierarchy is used only as a template for establishing groups and the locations of project entries within these groups. Project entry files can be located anywhere in the project tree, regardless of the group in which the entry resides.

Opening project entries

To open a project entry and access its contents, double-click the entry in the Project window.

If the project entry file is a text file, it is displayed in an Editor window. If it is a Project file, the project is opened by the Symantec Project Manager and its Project window is displayed. Otherwise, the file is opened with the application used to create it. For example, if the entry is a ResEdit resource file, opening it launches ResEdit (or switches to ResEdit if it's already running) and opens that file.

Note

For all project entries except groups and shared libraries, double-clicking is a shortcut for selecting the entry and choosing **Open "*filename*"** from the **File** menu. Double-clicking on a group lets you rename the group. Double-clicking on a shared library opens a dialog box that lets you set options for it.

You can also open a project using the **Switch Main Project** command; see the next section for details.

Working with multiple projects

The Symantec Project Manager allows you to have multiple projects open at the same time. One of the currently open projects is designated as the main project. When only one project is open, it is automatically designated as the main project. You can specify the project that should be the main project using the **Switch Main Project** submenu in the **Project** menu. This submenu lists all the recently and currently open projects, as well as any project aliases in the `(Projects)` folder in the `Symantec C++ for Power Mac` folder. The main project's name is listed with a checkmark. To designate a project as the main project, choose its name from the **Switch Main Project** submenu. If the project you choose is not open, it is opened automatically.

When working with several projects at the same time, it is important to know the project that will be affected by project-related commands you might choose. The Symantec Project Manager applies the following rules to determine the project that will be affected:

1. If the frontmost window is a Project window, the command affects the project to which the window belongs.

2. If the frontmost window is not a Project window, the command affects the main project.

The following section lists the project-related commands that can be chosen while a window other than the project window is frontmost.

Project-specific commands

The following project-specific commands are also available when the frontmost window is not a Project window:

- **Find**, **Find Next**, and **Find in Next File** in the **Search** menu (when performing multifile searches). See Chapter 21, "The Search Menu," for details on this menu.

- **Options**, **Add Files**, **Add Window**, **Add Group**, and **Run** in the **Project** menu. See Chapter 18, "The Project Menu," for details on this menu.

- All commands except **Get Library Info** in the **Build** menu. See Chapter 23, "The Build Menu," for details on this menu.

- **Build Errors**, **Search Results**, and **Class Browser** in the **Windows** menu. See Chapter 26, "The Windows Menu," for details on this menu.

Closing a project

To close a project, do either of the following:

- Choose **Close** from the **File** menu with the Project window frontmost.

- Click the close box in the Project window title bar.

If any Build Error, Search Results, or Class Browser windows are open for that project, they are closed along with the Project window.

Editing a Project's Code ◆ 4

This chapter describes how to use the Editor. It outlines the procedures for such tasks as viewing and moving around in text files, splitting an Editor window into subpanes to view different parts of a text file, and jumping to a specific place in your code. It also describes auto-indenting and syntax highlighting, two of the special features of the Editor. For more details, see Chapter 19, "The Editor Window," and Chapter 20, "The Edit Menu."

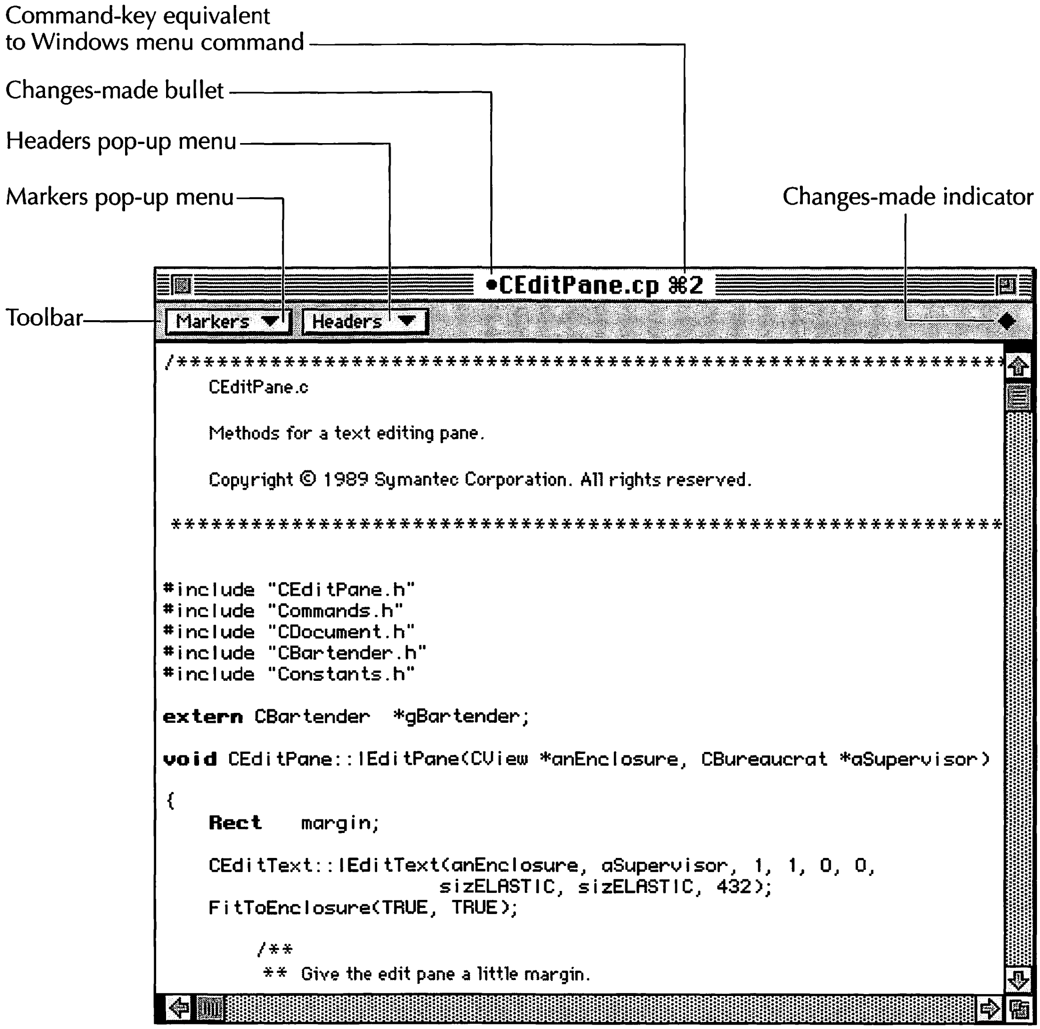
Opening Files and Viewing Application Code

The Editor works similarly to other standard Macintosh editors, with the addition of several special features. This section describes the procedures for opening text files, as well as for viewing and moving around in them. You can open any text file, including source files that contain C, C++, and Symantec Rez code for a project.

To open a text file for editing:

1. Select the name of the file.
2. Choose **Open** from the **File** menu (Command-O).
3. Navigate to the file in the **File Open** dialog box and click Open.

The contents of the file are displayed in an Editor window. An example of an Editor window for a source file is shown in Figure 4-1.



```
/***************************************************************
    CEditPane.c

    Methods for a text editing pane.

    Copyright © 1989 Symantec Corporation. All rights reserved.

 ***************************************************************


#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)
{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                        sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little margin.
```

Figure 4-1 Editor window for a source file

Viewing headers or function definitions

You can also use the Editor to view classes, including a class's own definition and logically related class definitions and member functions. Having this global view of related code simplifies the process of examining unfamiliar source code.

To view a set of class or member function definitions:

1. Click the **Headers** pop-up menu in the Project or Editor window's toolbar and select one of the header files.

   The header file is shown in an Editor window.

Note

This works only if the file has been compiled.

2. Click the **Markers** pop-up menu in the Editor window's toolbar and select a class or function.

   The Editor window displays the definition of the class or function.

For more on the **Markers** pop-up menu, see "Moving to a specific function" and "Using markers" later in this chapter. For more on the **Headers** pop-up menu, see Chapter 19, "The Editor Window," and Chapter 20, "The Edit Menu."

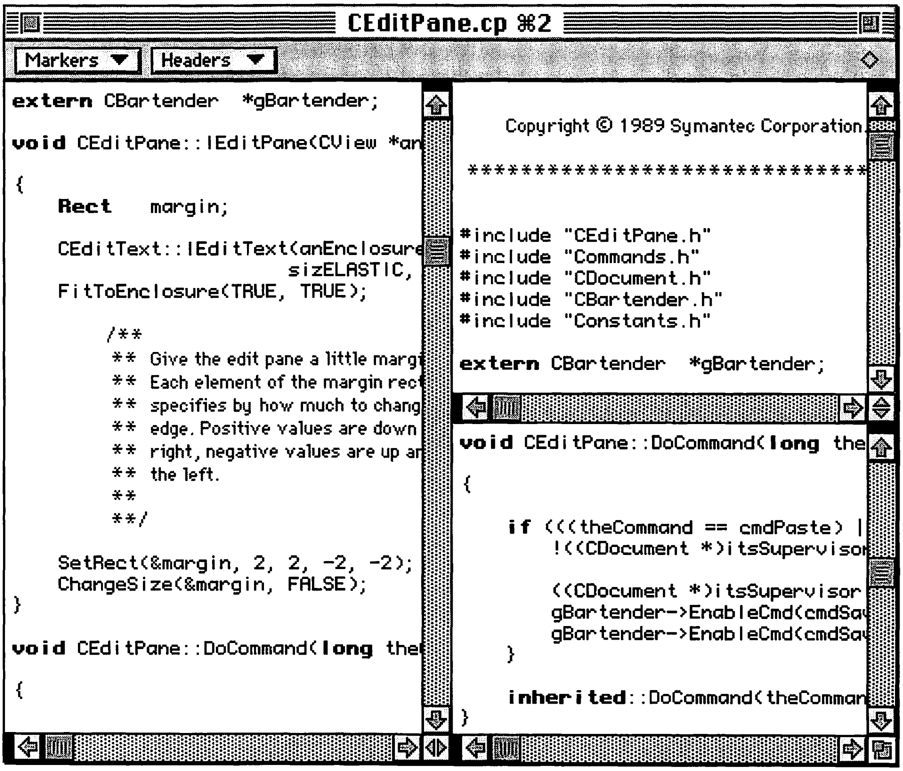
You can also access class or function definitions using the **Search** menu. See "Searching and Replacing Text," later in this chapter. If you do not know a class name, you can still view its definition using the Class Browser. To learn about that tool, see Chapter 5, "Viewing and Editing Classes," and Chapter 22, "The Class Browser Window."

Navigating in a text file

You can use standard Macintosh features to move around in a text file you have opened. In addition, the Editor provides a number of advanced features, such as splitting windows, moving to a function, jumping to a marker or a specific line, and finding a selection.

Splitting windows and resizing panes

You can split windows to form subpanes for viewing different parts of a source file, as shown in Figure 4-2.



```
extern CBartender   *gBartender;

void CEditPane::IEditPane(CView *an

{
    Rect    margin;

    CEditText::IEditText(anEnclosur
                        sizELASTIC,
    FitToEnclosure(TRUE, TRUE);

        /**
        ** Give the edit pane a little margi
        ** Each element of the margin rec
        ** specifies by how much to chang
        ** edge. Positive values are down
        ** right, negative values are up ar
        ** the left.
        **
        **/

    SetRect(&margin, 2, 2, -2, -2);
    ChangeSize(&margin, FALSE);
}

void CEditPane::DoCommand(long the

{
```

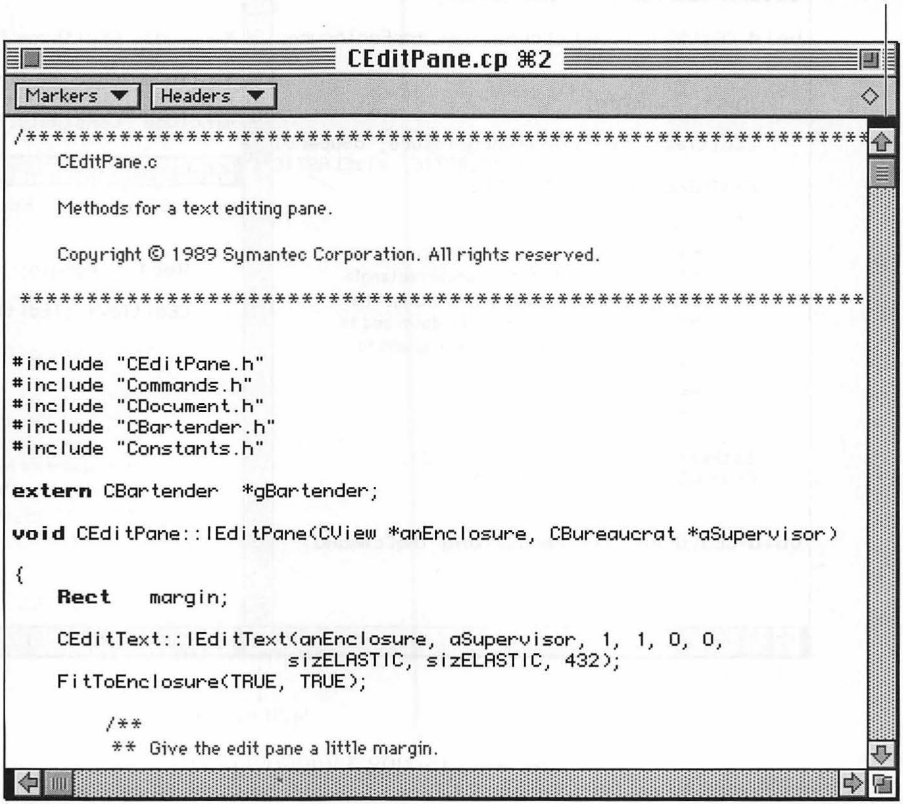
```
      Copyright © 1989 Symantec Corporation

 ****************************

#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender   *gBartender;
```

```
void CEditPane::DoCommand(long the
{

    if (((theCommand == cmdPaste) |
        !((CDocument *)itsSuperviso

        ((CDocument *)itsSupervisor
        gBartender->EnableCmd(cmdSa
        gBartender->EnableCmd(cmdSa
    }

    inherited::DoCommand(theComman
}
```

Figure 4-2 Subpanes of an Editor window

To split a window, you use the split bar, which is the black rectangle next to the scroll bar (see Figure 4-3). Double-click or click and drag the split bar. To remove a split bar, double-click its split mover, which is represented by the double triangles next to the split bar. Alternatively, click and drag the split mover to the edge of the window.

Double-click here to split window horizontally

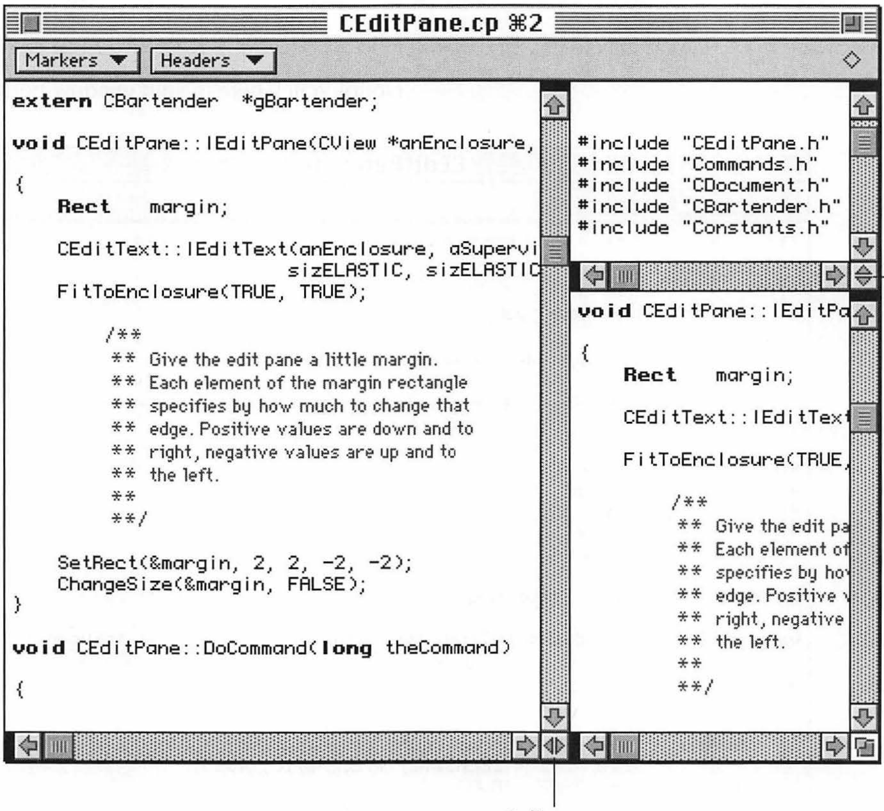


```
/*****************************************************************
	CEditPane.c

	Methods for a text editing pane.

	Copyright © 1989 Symantec Corporation. All rights reserved.

 *****************************************************************


#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)

{
	Rect	margin;

	CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
						sizELASTIC, sizELASTIC, 432);
	FitToEnclosure(TRUE, TRUE);

		/**
		 ** Give the edit pane a little margin.
```

Double-click here to split window vertically

Figure 4-3 Splitting a window

Once the window is split, you can resize the window pane using the split mover (see Figure 4-4). Click and drag the split mover as desired. Note that you can scroll independently in each of the subpanes you have created.



```
extern CBartender   *gBartender;

void CEditPane::IEditPane(CView *anEnclosure,

{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervi
                         sizELASTIC, sizELASTIC
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little margin.
         ** Each element of the margin rectangle
         ** specifies by how much to change that
         ** edge. Positive values are down and to
         ** right, negative values are up and to
         ** the left.
         **
         **/

    SetRect(&margin, 2, 2, -2, -2);
    ChangeSize(&margin, FALSE);
}

void CEditPane::DoCommand(long theCommand)

{
```

```
#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"
```

```
void CEditPane::IEditPa

{
    Rect    margin;

    CEditText::IEditTex

    FitToEnclosure(TRUE,

        /**
         ** Give the edit pa
         ** Each element of
         ** specifies by ho
         ** edge. Positive v
         ** right, negative
         ** the left.
         **
         **/
```

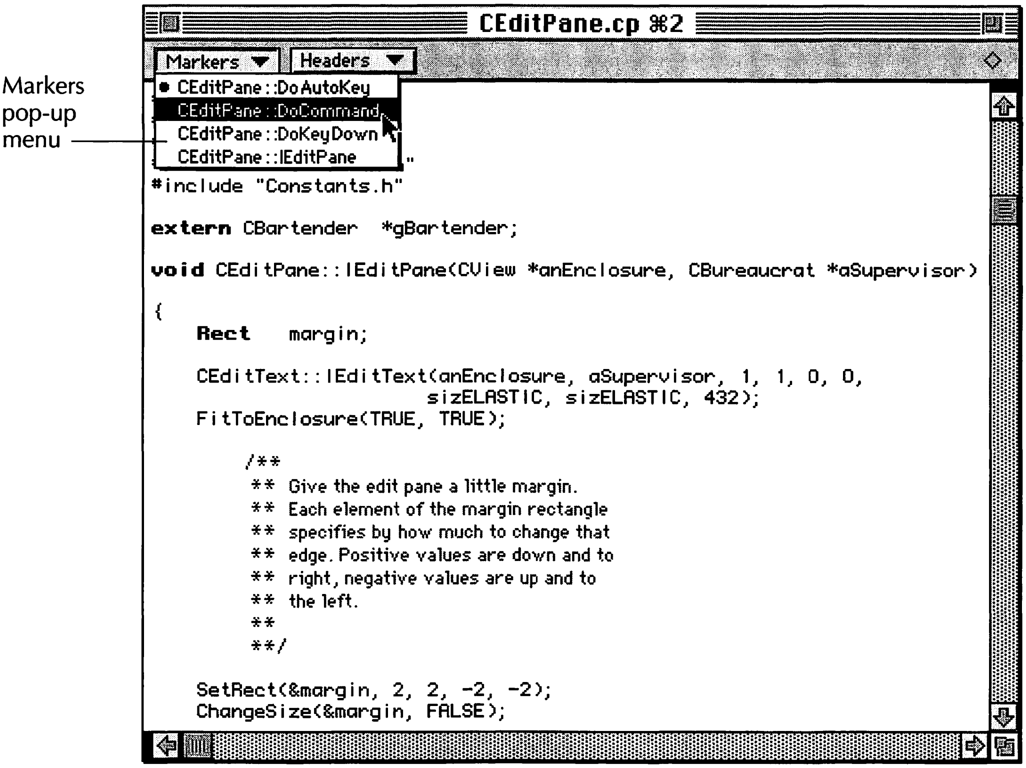
Split mover

Split mover

Figure 4-4 Resizing a window pane

Moving to a specific function

To move to a specific function in a file, select the function from the **Markers** pop-up menu in the window's toolbar, as shown in Figure 4-5.



```
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)

{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                         sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little margin.
         ** Each element of the margin rectangle
         ** specifies by how much to change that
         ** edge. Positive values are down and to
         ** right, negative values are up and to
         ** the left.
         **
         **/

    SetRect(&margin, 2, 2, -2, -2);
    ChangeSize(&margin, FALSE);
```

Figure 4-5 Moving to a specific function

The **Markers** menu lists both user-defined and automatically generated markers, including classes, `enums`, `typedefs`, `pragma` marks, and functions.

You also can access this list by Command-clicking the title bar (Command-Option-Comma). Command-clicking the title itself brings up the **File Path** pop-up menu, which shows the path for the file. See Chapter 19, "The Editor Window," for details.

Using markers

You can place a marker at any line in a file, as follows:

1. Choose **Add Marker** from the **Search** menu (Command-M).
2. Type the name of the marker in the **Add Marker** dialog box.
3. Click Add.

   Your marker is now displayed in the **Markers** pop-up menu.

To remove one or more markers:

1. Choose **Remove Markers** from the **Search** menu.
2. In the **Remove Markers** dialog box, click those markers that you want to remove.
3. Click Remove.

   Those markers are no longer shown on the **Markers** pop-up menu.

To move to a specific marker in your file, select the marker from the **Markers** pop-up menu.

Jumping to a specific line in a file

To move to a specific line in a file:

1. Choose the **Go To Line** command from the **Search** menu (Command-Comma).
2. Type the desired line number in the Line field of the **Go To Line** dialog box.
3. Click Go To.

Returning to the selected text

To go back to a highlighted portion of text (the current selection) after scrolling to another part of the file, press Enter.

If you have scrolled so the current selection is not in view, pressing the Enter key brings it back into view. If the entire selection cannot be displayed, pressing the Enter key again toggles between viewing the beginning and the end of the selection.

Entering and Editing Text

Some of the Editor's features (such as syntax highlighting, delimiter matching, and auto-indenting) simplify the task of writing legible code and reading such code.

Adding and deleting text

You can add and delete text in the usual ways provided by Macintosh text editors. Double-clicking a word selects the entire word; triple-clicking a line selects the entire line.

Scrolling and automatic indenting

Because the Editor is a source code editor, it lacks the word-wrap feature contained in other editors. If you type past the right edge of the window, the window automatically scrolls horizontally so you can still see the insertion point.

The Editor has an automatic indenting feature. When you press Return to start a new line, the Editor indents the new line with the same number of leading tabs and spaces as the previous line.

The Editor also has a block auto-indent feature. When you press Return at the end of a line that starts a block of code (for example, a line that ends with the left brace { character for C, C++, and Rez files) the Editor adds an additional level of indentation. The Editor also automatically outdents the ending line of the block as you type the matching right brace } character.

To change the indentation for a particular line, backspace over the indent with the Delete key.

To prevent the Editor from auto-indenting or outdenting a line, hold down the Option key and press Return.

You may want to turn off these features in the Editor General Settings page of the **Project Manager Preferences** dialog box, if they conflict with your coding conventions. The block auto-indent feature works only when the regular auto-indent feature is also turned on.

Syntax highlighting

The Editor helps ensure the legibility of code by highlighting different kinds of words in different colors and styles. For example, by default, preprocessor directives are in blue, language keywords such as `do` and `while` are in bold, string literals are in red, and comments are in gray.

The Editor supports syntax highlighting in the following languages: C, C++, AppleScript, MPW Shell Script, Pascal, and Symantec Rez. Untitled windows default to the C++ configuration.

To learn how to custom configure colors and styles, see Chapter 20, "The Edit Menu."

Delimiter matching

The delimiter matching feature of the Editor lets you check for matching pairs of parentheses, square brackets, and braces. Parentheses and other delimiters generally appear in matched pairs, which may be nested within each other to any depth. The Editor warns you with a beep when you type a closing delimiter that has no matching opener.

If you double-click on a bracketing delimiter (that is, a parenthesis `()`, square bracket `[]`, or brace `{}`), the Editor selects the text between that delimiter and its matching delimiter.

This feature also works with the slash `/`, reverse slash `\`, double-quote `"`, and single-quote `'` characters. The Editor selects forward to the next occurrence of the delimiter.

In addition, delimiter matching works dynamically. If you double-click and drag within a pair of delimiters, the Editor selects the delimiters and all text between them. Holding down the Option key omits the delimiters from the selection.

After you type the closing element of a delimiter pair (that is, a right parenthesis `)`, square bracket `]`, or brace `}`), the Editor briefly highlights the matching delimiter.

The Editor beeps if the delimiters are improperly matched.

To verify that all functions are properly balanced:

1. Place the insertion point at the beginning of a file.

2. Use the **Find** command on the **Search** menu (Command-F) to search for the first opening code-block delimiter (a left brace { in C or C++).

3. Use the **Balance** command on the **Edit** menu (Command-B) and the **Find Next** command on the **Search** menu (Command-G) repeatedly until you reach the end of the file.

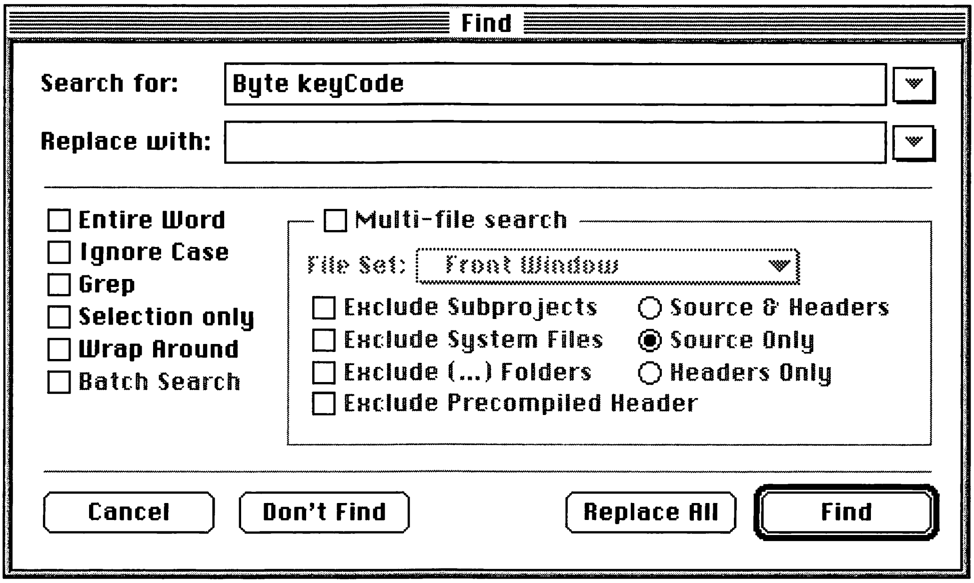
Searching and Replacing Text

A wide range of search and replace capabilities are available as outlined in this section. For information on more advanced features, see Chapter 21, "The Search Menu."

Finding and replacing strings

To find the first instance of a string:

1. Choose the **Find** command from the **Search** menu (Command-F). The **Find** dialog box opens (Figure 4-6).



Search for: Byte keyCode

Replace with:

□ Entire Word
□ Ignore Case
□ Grep
□ Selection only
□ Wrap Around
□ Batch Search

□ Multi-file search

File Set: Front Window

□ Exclude Subprojects
□ Exclude System Files
□ Exclude (...) Folders
□ Exclude Precompiled Header

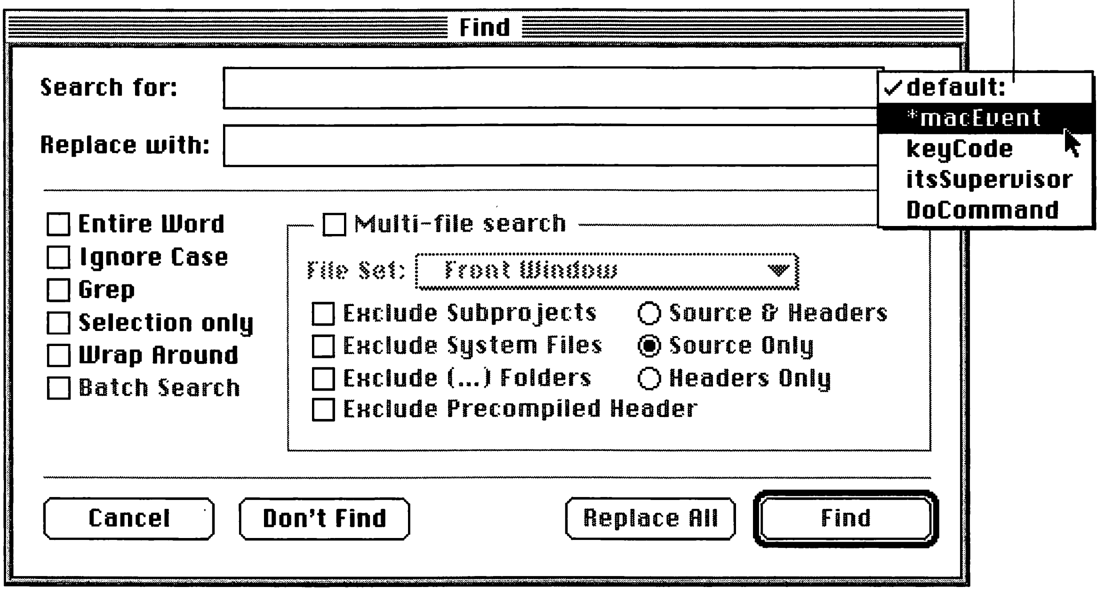
○ Source & Headers
● Source Only
○ Headers Only

Figure 4-6 Find dialog box

2. In the Search for and Replace with textboxes, enter the search and replace strings.

   Either type the string or click on the arrow by the textbox to open a pop-up menu (Figure 4-7). Each pop-up menu contains the five strings most recently entered in the field.

Search for pop-up menu



Search for:

Replace with:

□ Entire Word
□ Ignore Case
□ Grep
□ Selection only
□ Wrap Around
□ Batch Search

□ Multi-file search

File Set: Front Window

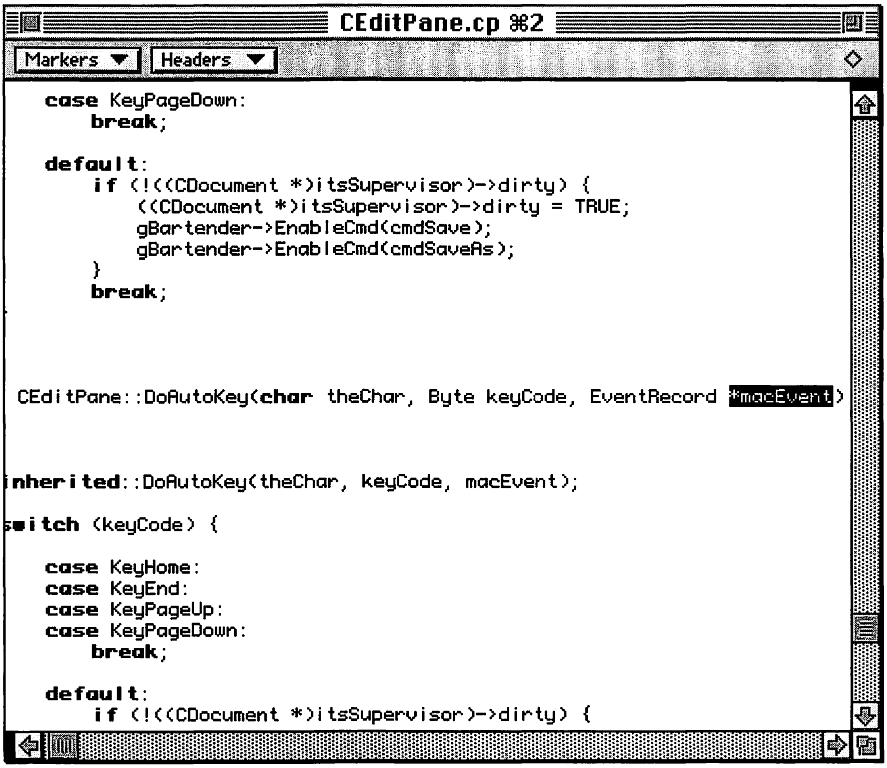
□ Exclude Subprojects
□ Exclude System Files
□ Exclude (...) Folders
□ Exclude Precompiled Header

○ Source & Headers
● Source Only
○ Headers Only

Figure 4-7 Search for pop-up menu in the Find dialog box

3. Click Find.

   If the Editor cannot find the string, you hear a beep. If it does find the string, it scrolls to and highlights the string, as shown in Figure 4-8.



```
   case KeyPageDown:
       break;

   default:
       if (!((CDocument *)itsSupervisor)->dirty) {
           ((CDocument *)itsSupervisor)->dirty = TRUE;
           gBartender->EnableCmd(cmdSave);
           gBartender->EnableCmd(cmdSaveAs);
       }
       break;



 CEditPane::DoAutoKey(char theChar, Byte keyCode, EventRecord *macEvent)


inherited::DoAutoKey(theChar, keyCode, macEvent);

switch (keyCode) {

   case KeyHome:
   case KeyEnd:
   case KeyPageUp:
   case KeyPageDown:
       break;

   default:
       if (!((CDocument *)itsSupervisor)->dirty) {
```

Figure 4-8 Highlighted string found by Editor

You have four choices for finding and replacing further instances of the search string. Click Cancel at any time to end the search.

- To go to the next instance of the search string without replacing the current instance, choose **Find Next** from the **Search** menu (Command-G). To reverse the direction of search, press Shift as you open the **Search** menu and choose **Find Previous** (Shift-Command-G).

- To replace the current instance of the search string with a replacement string, choose **Replace** from the **Search** menu (Command-Equals sign).

- To replace the current instance of the search string and proceed to the next one, press Option-Command-Equals sign. To reverse the direction of the search, press Shift as you open the menu and choose **Replace & Find Previous** (Shift-Option-Command-Equals sign).

- Replace every instance of the search string in the searched file with the replacement string, click Replace All in the **Find** dialog box or choose **Replace All** from the **Search** menu.

Warning

When you click Replace All, a dialog box warns you that Replace All is not reversible. Be sure that you want to replace all instances of the search string before you click Continue. Also, before choosing a **Replace** command, make sure you have entered the replacement string you want in the **Find** dialog box. Otherwise, the search string is replaced with nothing, thus deleting every instance of the string in the file.

Searching through multiple files

You can do both batch and nonbatch searches on one or more files.

Performing a nonbatch search

You can look for a string in more than one file. To find and replace a string in all open windows or in all files in the current project:

1. Choose **Find** from the **Search** menu (Command-F).

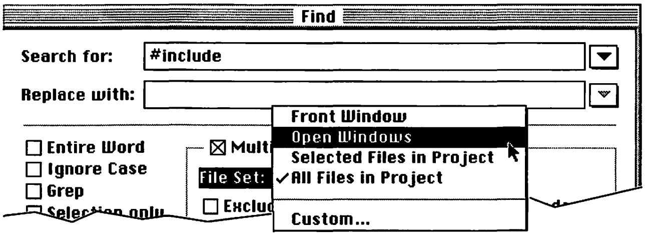
2. Enter the string to find in the Search for field.

3. Enter the replacement string in the Replace with field.

4. Set the Multi-File Search option on in the **Find** dialog box.

5. Click one of the Source Only, Headers Only, or Source & Headers radio buttons.

6. In the **File Set** pop-up menu, choose **All Files in Project** to search all files in the current project. Alternatively, choose **Open Windows** to search all open windows (Figure 4-9).



Find

Search for: #include

Replace with:

☐ Entire Word
☐ Ignore Case
☐ Grep
☐ Selection only

☒ Multi
File Set:
☐ Exclu

Figure 4-9 Multi-file search of all open windows

7. Make sure the Batch Search option is set off.

8. Click Find.

9. Replace the string as noted in the previous section.

10. Choose **Find in Next File** from the **Search** menu (Command-T), after the Editor reaches the end of the file.

    When all the selected files have been searched, the multi-file search turns off.

If the Wrap Around option is set on, the Editor continues the search at the top of the same file, rather than beeping and stopping at the end of the file. Choose **Find in Next File** from the **Search** menu (Command-T), so that the search moves on to the next file.

Performing a batch search

To find all instances of a search string in one or more files:

1. Choose **Find** from the **Search** menu (Command-F).

2. Enter the search string in the Search for field.

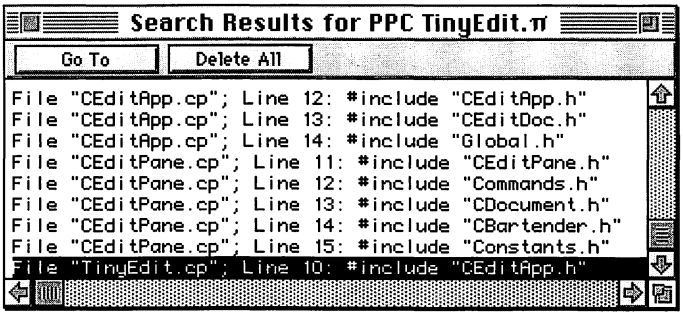
3. Set on both the Batch Search option and the Multi-File Search option in the **Find** dialog box.

4. Click one of the Source Only, Headers Only, or Source & Headers radio buttons.

5. Choose the files to search in the **File Set** pop-up menu.

6. Click Find.

Instead of scrolling to the next instance of the search string when you click Find, the Editor brings up a Search Results window that lists all instances of the search string in the searched files, as shown in Figure 4-10.



```
File "CEditApp.cp"; Line 12: #include "CEditApp.h"
File "CEditApp.cp"; Line 13: #include "CEditDoc.h"
File "CEditApp.cp"; Line 14: #include "Global.h"
File "CEditPane.cp"; Line 11: #include "CEditPane.h"
File "CEditPane.cp"; Line 12: #include "Commands.h"
File "CEditPane.cp"; Line 13: #include "CDocument.h"
File "CEditPane.cp"; Line 14: #include "CBartender.h"
File "CEditPane.cp"; Line 15: #include "Constants.h"
File "TinyEdit.cp"; Line 10: #include "CEditApp.h"
```

Figure 4-10 Batch Search Results window

To go to a selected instance of the search string from the Batch Search Results window, do one of the following:

- Click the Go To button in the window's toolbar.
- Double-click the entry in the window.

To learn about other options available for searching multiple files, see Chapter 21, "The Search Menu."

Using Grep to search for patterns

You can use Grep to search for strings that match a general pattern, rather than for a specific string. To use Grep, set the Grep option on the **Find** dialog box. The Editor accepts all standard Grep-style patterns. If you are not familiar with Grep, see Chapter 21, "The Search Menu," for details on how to specify patterns.

Saving Changes

To save a file without closing it, choose **Save** from the **File** menu (Command-S).

The first time you save the file, its Editor window is untitled. A **Save** dialog box opens that prompts you to name the file. Type the filename and click Save.

To change the name of a file, choose **Save As** from the **File** menu.

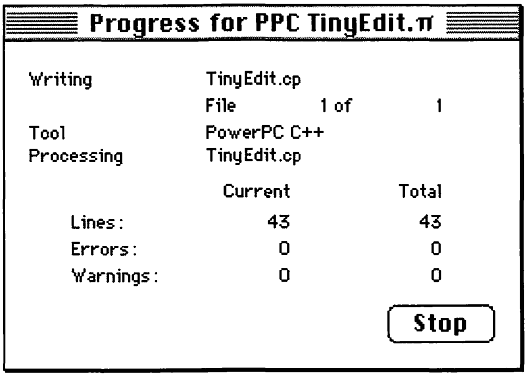
The **Save As** command saves the contents of the Editor window in a new file that you name in a **Save As** dialog box. If the file is part of a project, the name of that file also changes in the Project window.

To save the contents of the current Editor window under a new name while editing the original file under its original name, choose **Save A Copy As** from the **File** menu.

The Project window remains unchanged. This feature allows you to make backup copies without editing the backup by mistake.

Compiling a File

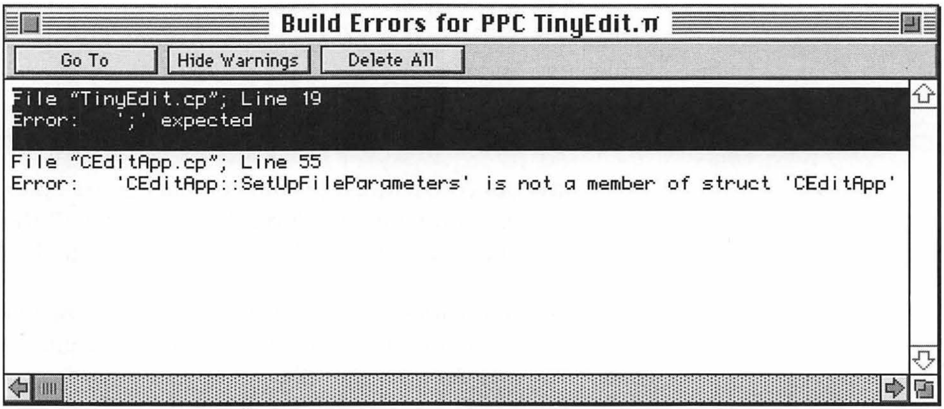
After you finish editing a file, you can immediately compile it as part of the currently open project. To do so, choose **Compile** from the **Build** menu. The Symantec Project Manager opens the **Progress** dialog box (Figure 4-11), which displays the progress of the current compilation.



```
Progress for PPC TinyEdit.π

Writing             TinyEdit.cp
                    File        1 of        1
Tool                PowerPC C++
Processing          TinyEdit.cp
                       Current         Total
     Lines:               43             43
     Errors:               0              0
     Warnings:             0              0
```

Figure 4-11 Progress dialog box

If the compilation is successful, the compiled file automatically is added to the project (unless the file is in the project already). If instead errors are generated, the file is not added to the project and the Build Errors window opens (Figure 4-12).



```
File "TinyEdit.cp"; Line 19
Error:    ';' expected

File "CEditApp.cp"; Line 55
Error:    'CEditApp::SetUpFileParameters' is not a member of struct 'CEditApp'
```

Figure 4-12 Build Errors window

The Build Errors window lists the errors found in a source file. How many are listed depends on the setting of the Error Reporting option on the Debugging subpage of the Power PC C or Power PC C++ page of the **Project Options** dialog. For every error, the Build Errors window shows the following:

- The project entry containing the statement that generated the error
- The location of the statement
- A brief message explaining the error

With Symantec C++, it is easy to correct the errors in your source file. To go to the statement in the source file that generated the error, select the error message and click Go To. Alternatively, double-click the error message. An Editor window opens, with the statement line in question selected.

After you fix all the errors in your code, compile the file again by choosing **Compile** from the **Build** menu.

Viewing and Editing Classes ◆ 5

Once you have created and compiled a project, you can analyze the structure of the source code by viewing the classes and functions it contains. Symantec C++ provides a Class Browser designed specifically for this purpose. With this tool, you can examine a project's class hierarchy as well as the classes' logically related data and function members. The ability to browse and edit pre-existing class hierarchies is especially useful when you are exploring the structure of unfamiliar source code.

This chapter describes how to work with C++ classes in the Class Browser. The first sections cover opening the Class Browser, navigating in the window, and viewing classes. With the Class Browser, you can also edit classes and their data members. The final section in the chapter outlines how to edit a class definition, a member function, and a data member.

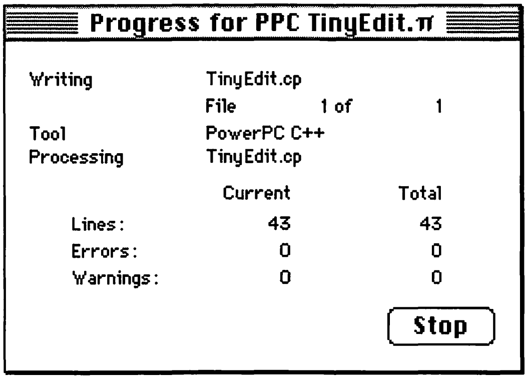
For a more detailed discussion of editing operations, see Chapter 4, "Editing a Project's Code." For a complete reference, see Chapter 22, "The Class Browser Window," Chapter 19, "The Editor Window," and Chapter 20, "The Edit Menu."

Before Browsing

Before you can browse the classes of a project, you must either build the whole project or compile a portion of it by selecting some of the project's files and compiling them, as described in the previous chapter.

To build the whole project, select **Bring Up To Date** from the **Build** menu. The Symantec Project Manager now checks the dependency tables and file date information, then compiles and links the files that have changed since the last time the project was built.

The **Progress** dialog box opens (Figure 5-1) to let you track the build of your project.



Progress for PPC TinyEdit.π

```
Writing          TinyEdit.cp
                 File        1 of      1
Tool             PowerPC C++
Processing       TinyEdit.cp
                   Current          Total
    Lines:            43              43
    Errors:            0               0
    Warnings:          0               0
```

Figure 5-1 Progress dialog box

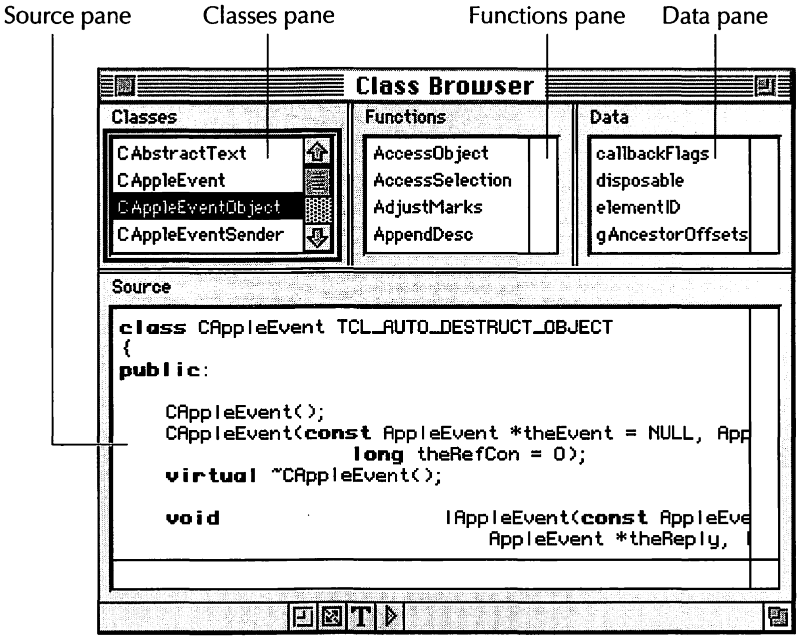
If the build generated errors, the Build Errors window opens to let you identify and fix the problems in your code. Chapter 4, "Editing a Project's Code," describes the process of fixing errors in your source files.

When a project is built, you can use the Class Browser to view and edit the classes.

Opening the Class Browser

The Class Browser displays a list-based or hierarchical view of a class hierarchy. To open a Class Browser window (Figure 5-2), choose **Class Browser** from the **Windows** menu (Command-J).

Once you have opened a Class Browser window for a project, you cannot display another project's classes in it. Instead, you must open a second Class Browser window for the other project. To open a Class Browser with a different project as the active project, hold down the Option key and choose **Class Browser** from the **Windows** menu.



CAbstractText
CAppleEvent
CAppleEventObject
CAppleEventSender

AccessObject
AccessSelection
AdjustMarks
AppendDesc

callbackFlags
disposable
elementID
gAncestorOffsets

```
class CAppleEvent TCL_AUTO_DESTRUCT_OBJECT
{
public:

    CAppleEvent();
    CAppleEvent(const AppleEvent *theEvent = NULL, App
                long theRefCon = 0);
    virtual ~CAppleEvent();

    void                IAppleEvent(const AppleEve
                            AppleEvent *theReply,
```

Figure 5-2 Class Browser window

The Class Browser window is divided into four panes:

- The Classes pane, which lists the classes.
- The Functions pane, which lists the member functions of the current (highlighted) class.
- The Data pane, which lists the data members of the current (highlighted) class.
- The Source pane, which displays function or data member source code. You can edit the source code in this pane or use the text editor (see Chapter 19, "The Editor Window").

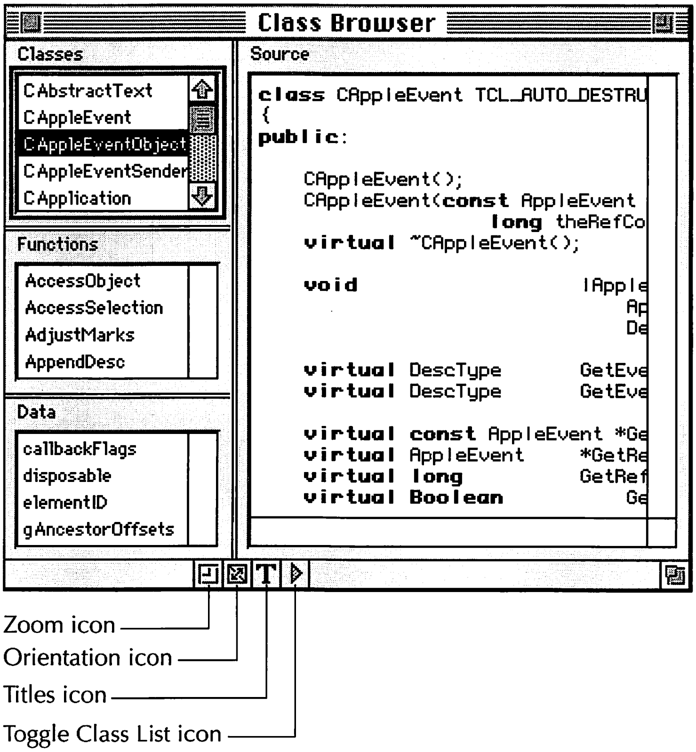
Navigating in the Class Browser Window

Only one of the four panes in the Class Browser window can be active at a time (Figure 5-3). The active pane, which has a black border, receives menu commands and all keystrokes.

To make a pane active, do one of the following:

- Click on it.
- Press Command-Tab to cycle through the panes (to cycle backward, use Command-Shift-Tab).

Depending on your working style, you may need to alter the layout of the Class Browser window. One possible rearrangement is shown in Figure 5-3.



Classes

CAbstractText
CAppleEvent
CAppleEventObject
CAppleEventSender
CApplication

Functions

AccessObject
AccessSelection
AdjustMarks
AppendDesc

Data

callbackFlags
disposable
elementID
gAncestorOffsets

Source

```
class CAppleEvent TCL_AUTO_DESTRU
{
public:

    CAppleEvent();
    CAppleEvent(const AppleEvent
                     long theRefCo
    virtual ~CAppleEvent();

    void                       IApple
                                   Ap
                                   De

    virtual DescType           GetEve
    virtual DescType           GetEve

    virtual const AppleEvent *Ge
    virtual AppleEvent        *GetRe
    virtual long               GetRef
    virtual Boolean                Ge
```

Figure 5-3 Class Browser window in a vertical orientation

You can use any of the four icons at the bottom of the window to alter the layout:

- Zoom: Expands the active pane to fill the entire window
- Orientation: Changes the orientation of the panes from horizontal to vertical, and back again
- Titles: Toggles to display or hide pane titles
- Toggle Class List: Toggles the list of classes in the Classes pane between an alphabetic class listing or a hierarchical ordering

In addition, size bars are available for changing the relative sizes of the panes. Adjust a pane's size by clicking on and dragging the split bars between the panes.

Note

> Any customization made to the Class Browser window is not saved when the window is closed.

Viewing the Class Hierarchy

You can view the class structure displayed in the Class Browser window's Classes pane either as an alphabetic list or a structured hierarchy. Both options are described here.

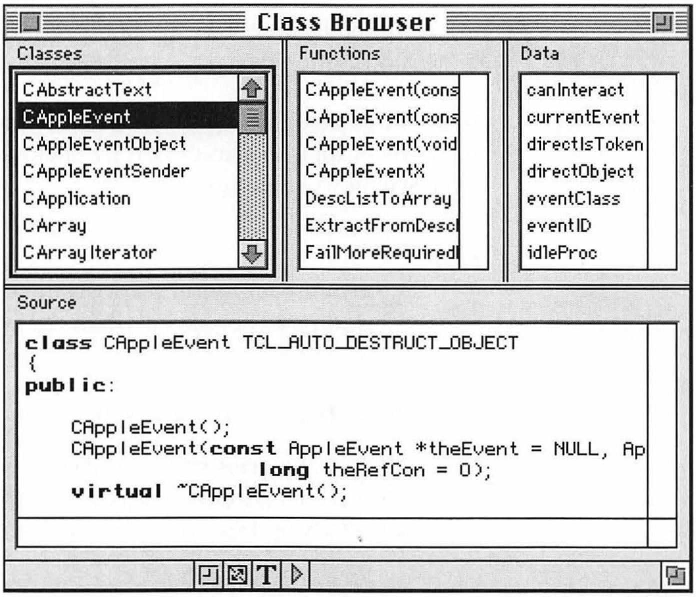
Note

> You could use the hierarchical view to check the logical structure of a project and the list-based alphabetic view to verify that nothing is missing.

List-based viewing

To display a class definition (which includes the class's functions and data members) in the Source pane, double-click on one of the classes in the Classes pane. Alternatively, select a class in the Classes pane by clicking on it and press Enter.

For example, the information displayed in the Source pane of Figure 5-4 is displayed as the result of double-clicking the CAppleEvent class in the Classes pane in Figure 5-4.

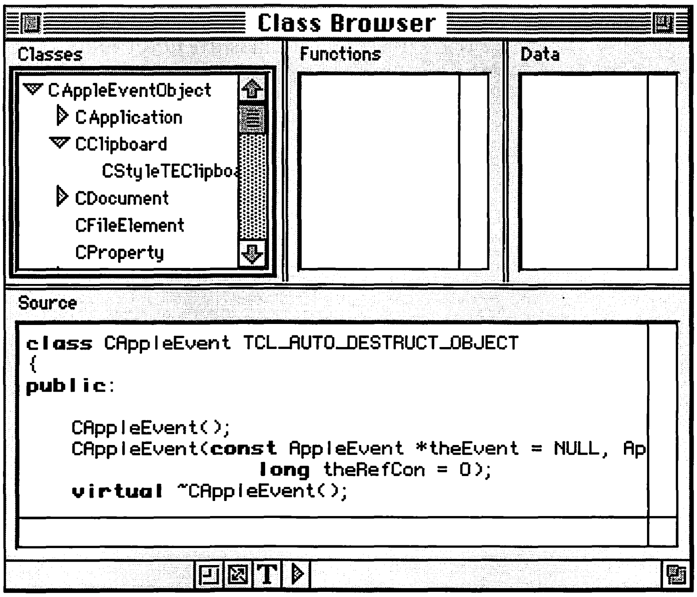


```
class CAppleEvent TCL_AUTO_DESTRUCT_OBJECT
{
public:

    CAppleEvent();
    CAppleEvent(const AppleEvent *theEvent = NULL, Ap
                long theRefCon = 0);
    virtual ~CAppleEvent();
```

Figure 5-4 List-based view of a class hierarchy in the Classes pane

Hierarchical viewing

The list of classes in the Classes pane can also be displayed hierarchically. A hierarchical class view shows a class's substructure (its subclasses, if it contains any). For example, Figure 5-5 shows the subclass structure of one of the classes in the Classes pane.



```
class CAppleEvent TCL_AUTO_DESTRUCT_OBJECT
{
public:

    CAppleEvent();
    CAppleEvent(const AppleEvent *theEvent = NULL, Ap
                    long theRefCon = 0);
    virtual ~CAppleEvent();
```

Figure 5-5 Classes pane displayed in hierarchical order

Note

The Class Browser does not fully support viewing of functions and data members for classes contained in subprojects. These classes cannot be expanded to view these elements. Instead, these classes appear as italicized entries in the Class Browser. To view the elements of a class that is in a subproject, you open up a second Class Browser for that subproject.

Editing Class Information

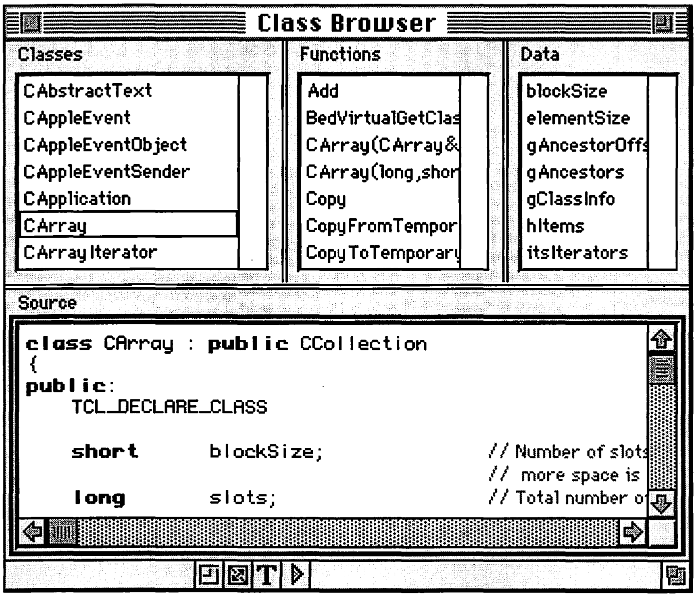
This section describes how to edit class definitions, member functions, and data members. Such editing occurs in the Source pane, which displays the text of the class, member function, or data member that was double-clicked or entered in another pane of the Class Browser. The Source pane limits the view to the class or member being browsed, rather than displaying an entire file. All editing operations available in the text editor are also available in the Source pane of the browser.

Editing a class definition

You can select a class and edit its definition in the Source pane. To do so:

1. Double-click the class's name in the Classes pane.

   The file containing the class's source code opens for editing in a Source pane (see Figure 5-6).



```
class CArray : public CCollection
{
public:
    TCL_DECLARE_CLASS

    short       blockSize;                  // Number of slots
                                            //  more space is
    long        slots;                      // Total number o
```

Figure 5-6 Editing a class definition within a Class Browser window

2. Make any changes to the code directly in the Source pane.

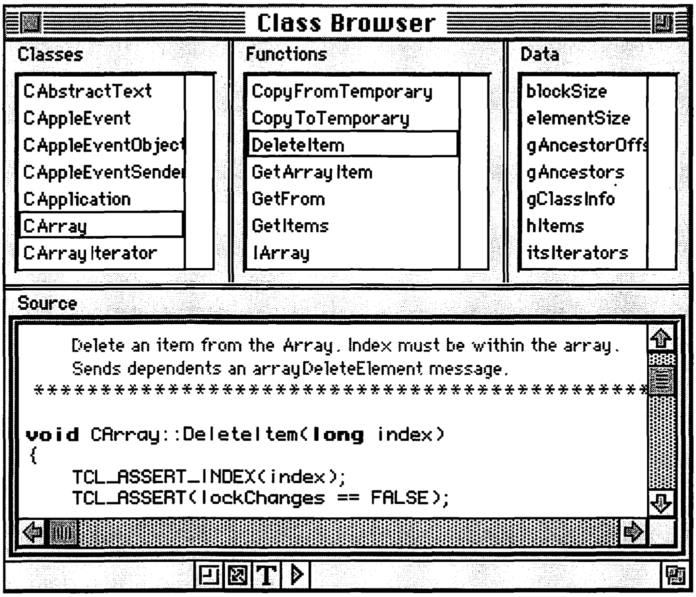
3. Save these changes by choosing the **Save** command from the **File** menu.

Editing a member function

The Functions pane lists all the member functions of the class selected in the Classes pane. To view and edit one of these member functions:

1. Double-click the name of the member function in the Functions pane.

   The file containing the member function's source code opens for editing in a Source pane (see Figure 5-7).



```
    Delete an item from the Array. Index must be within the array.
    Sends dependents an arrayDeleteElement message.
 ********************************************************

void CArray::DeleteItem(long index)
{
    TCL_ASSERT_INDEX(index);
    TCL_ASSERT(lockChanges == FALSE);
```

Figure 5-7 Editing a member function

2. Edit the function as desired.

   However, if you change function argument types or return types, you have to manually modify or add the function declarations in the header file as well.

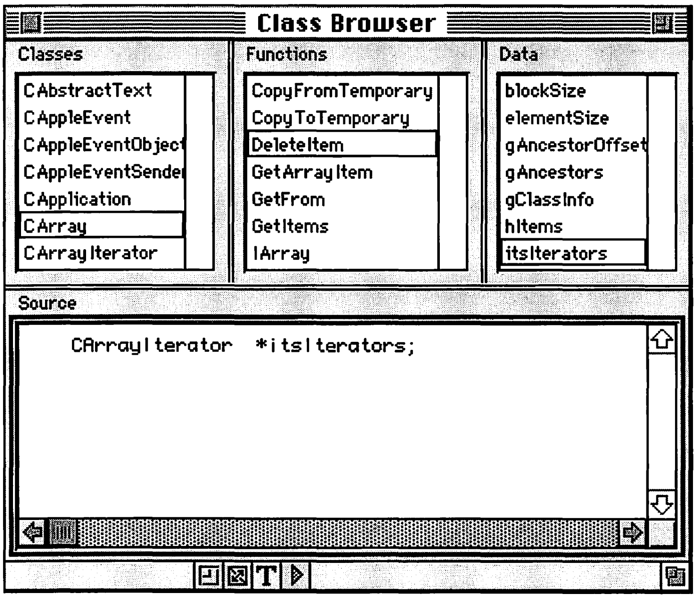
3. Save your changes by choosing **Save** from the **File** menu.

Editing a data member

The Data pane lists the data members defined for the class selected in the Classes pane. To view or edit a data member listed in the Data pane:

1. Double-click the data member name in the Data pane.

   The file containing the data member source code opens for editing in a Source window (Figure 5-8).



```
    CArrayIterator  *itsIterators;
```

Figure 5-8 Editing a data member

2. Edit the source as desired.

3. Save your changes by choosing **Save** from the **File** menu.

Using the Debugger ◆ 6

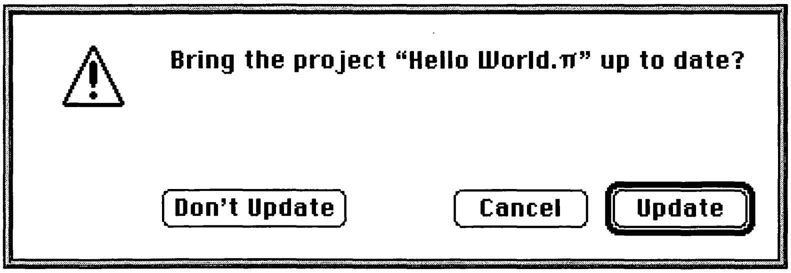
The Symantec Debugger is a powerful tool for testing your application. The Debugger lets you step through your code line by line as it runs. It also lets you set breakpoints at specific lines at which you might want to examine the state of the code's execution or to examine or change the values of variables.

This chapter outlines the basic steps involved in building an application as well as testing application code with the source-level Debugger. First, the Main and Debug Browser windows are described with their various panes. Then, procedures for stepping through code, setting breakpoints, examining the call chain, and formatting data are outlined. The final sections cover methods of analyzing variables, changing values, and evaluating expressions.

For a description of more advanced features, including lower-level debuggers that are available in the Symantec C++ environment, see Chapter 24, "The Debugger Windows," and Chapter 25, "The Debugger Menus."

Updating the Project

Before you can debug the project, you need to compile and link it. When you attempt to run a program with the Debugger (as outlined in the next section), the project may need to be updated due to changes in the source code. If so, the Symantec Project Manager prompts you to bring the project up-to-date (Figure 6-1).



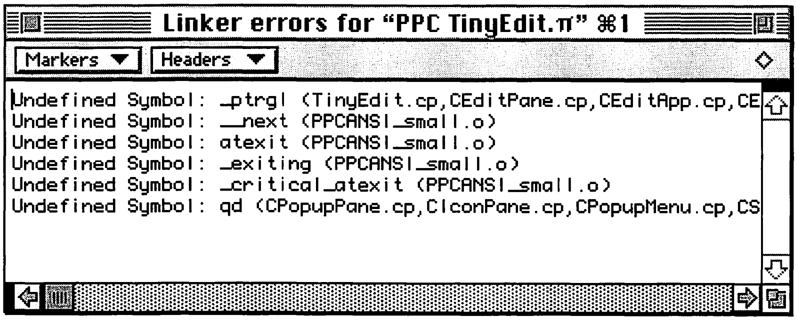
Bring the project "Hello World.π" up to date?

Figure 6-1 Update Project dialog box

Click Update to bring your project up-to-date.

The Symantec Project Manager then compiles and links the application. If errors occur during compilation, the Build Errors window opens. For information on dealing with compilation errors, refer to the section "Compiling a File," in Chapter 4, "Editing a Project's Code."

If errors are encountered during the linking process, a Linker Errors window opens (Figure 6-2).



```
Undefined Symbol: _ptrgl (TinyEdit.cp,CEditPane.cp,CEditApp.cp,CE
Undefined Symbol: __next (PPCANSI_small.o)
Undefined Symbol: atexit (PPCANSI_small.o)
Undefined Symbol: _exiting (PPCANSI_small.o)
Undefined Symbol: _critical_atexit (PPCANSI_small.o)
Undefined Symbol: qd (CPopupPane.cp,CIconPane.cp,CPopupMenu.cp,CS
```

Figure 6-2 Linker Errors window

The Linker Errors window shows the link errors and the source files that generated the errors. To rectify link errors, you need to examine the source files as well as which files are included in the project. For example, two common problems resulting in Undefined Symbol link errors are the failure to include a library or subproject and the failure to define variables or procedures in source files. See Chapter 23, "The Build Menu," for more details on handling link errors.

Starting a Debugging Session

You can launch the Debugger either from the Project Manager or from the Finder. As a general rule, launching the Debugger from the Project Manager is a good idea. If you have not changed any source files or recompiled anything since the last time you built the application, you can launch the Debugger from the Finder. Your project must be open in the Project Manager, so the Debugger can have access to symbolic debugger information.

To start a debugging session from the Project Manager:

1. Open the project that you want to debug.
2. Choose **Run with Debugger** from the **Project** menu (Command-R).

   If you have changed your source files since last running your project, the **Update Project** dialog box prompts you to update your project.
3. Click Update.

To start a debugging session from the Finder, drag a built application onto the debugger.

Trouble-shooting

If you have problems launching the Debugger, be sure that the project is built with the Incremental Linker and that the following options are turned on:

- Enable Symbolic Debugging (on the Compiler Options Debugging page of the **Project Options** dialog box)
- Run with Debugger (on the Project Options page of the **Project Options** dialog box)

See Chapter 18, "The Project Menu," to learn how to select a linker and turn options on.

Also, be sure that the Debugging flag in the bug column of the Project window is set on for all files that you want to view in the Debugger. Be sure that the two-machine Debugger Nub is set off, if you have that installed.

Note

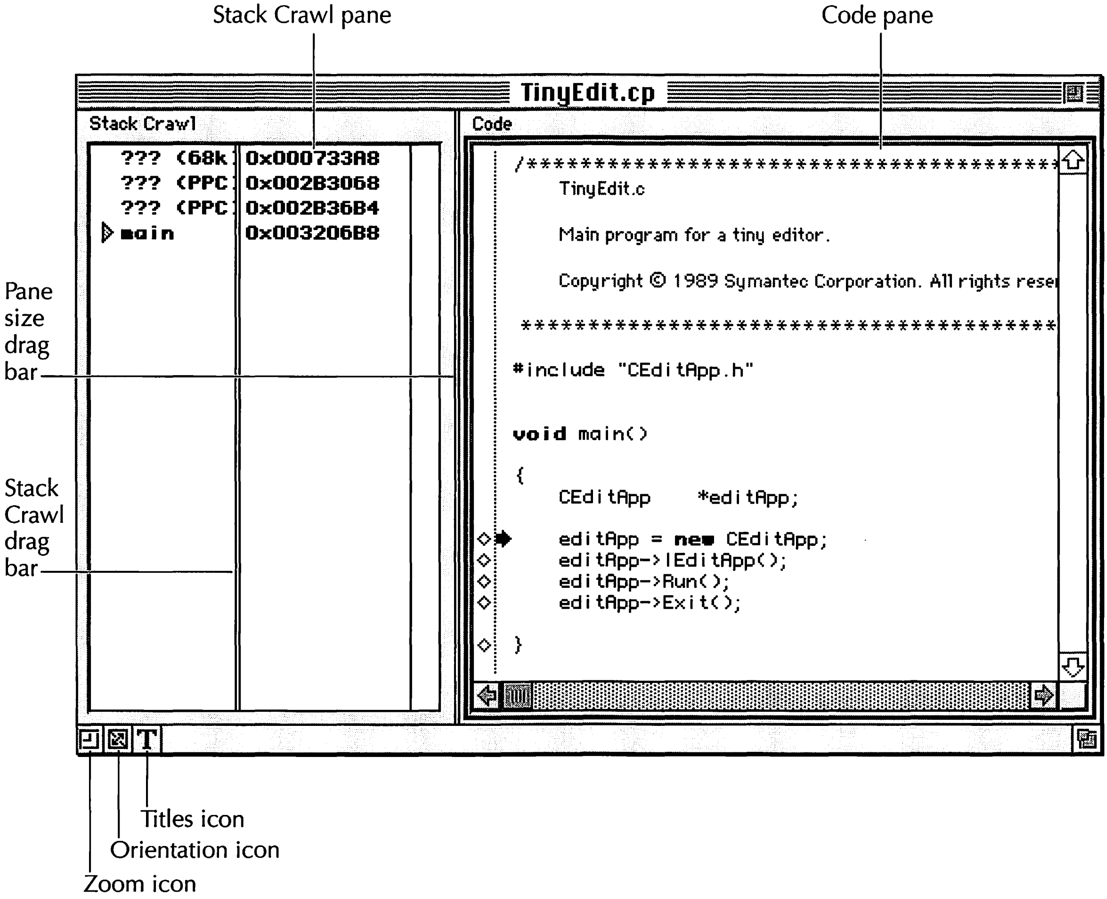
Debugging optimized code may not give the results you expect.

The Debugger Windows

The Debugger provides several kinds of windows for performing various debugging tasks. Two of those windows are discussed in this section: the Main debugger window and Debug Browser windows. The Control palette is discussed in the section "Stepping Through Code." For more information on these and the other Debugger windows, see Chapter 24, "The Debugger Windows."

The Main debugger window

Starting a debugging session opens the Main debugger window, shown in Figure 6-3.



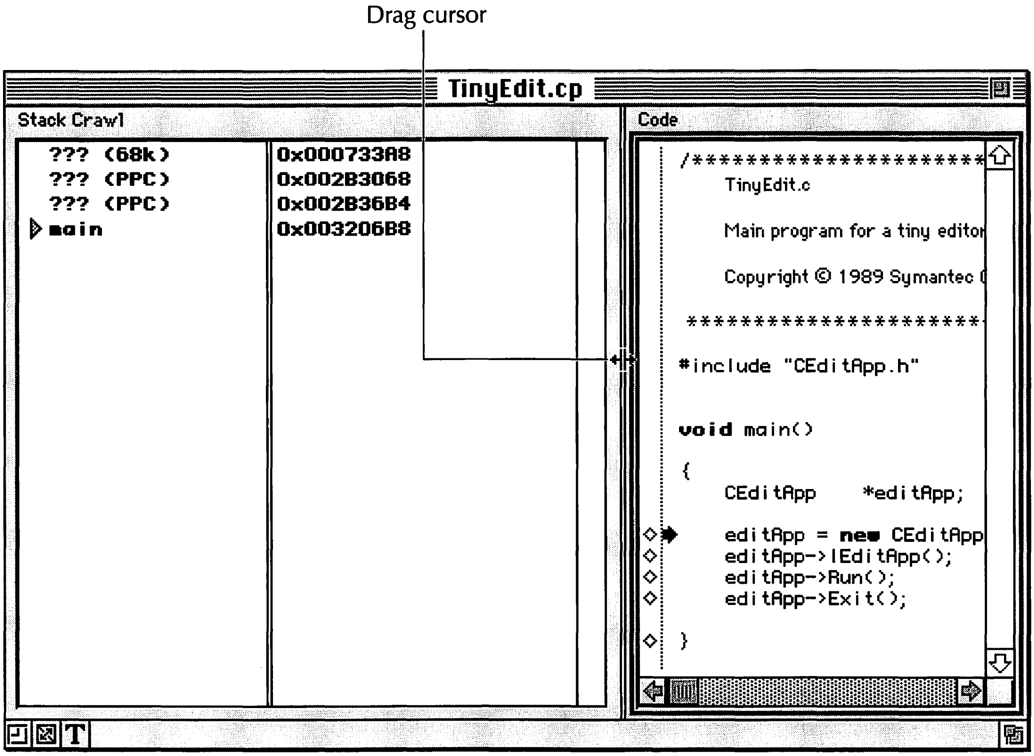
```
  ???  (68k  0x000733A8
  ???  (PPC  0x002B3068
  ???  (PPC  0x002B36B4
▷ main       0x003206B8
```

```
/**************************************
	TinyEdit.c

	Main program for a tiny editor.

	Copyright © 1989 Symantec Corporation. All rights rese

 **************************************

#include "CEditApp.h"


void main()

{
	CEditApp	*editApp;

	editApp = new CEditApp;
	editApp->IEditApp();
	editApp->Run();
	editApp->Exit();

}
```

Figure 6-3 Main debugger window

Looking at the Main window panes

The Main debugger window has two scrollable panes: a Code pane to examine the code and a Stack Crawl pane to examine the call chain for the current program counter. The two panes can be scrolled independently. Their relative sizes can be changed by dragging the double bar that separates them, as shown in Figure 6-4.



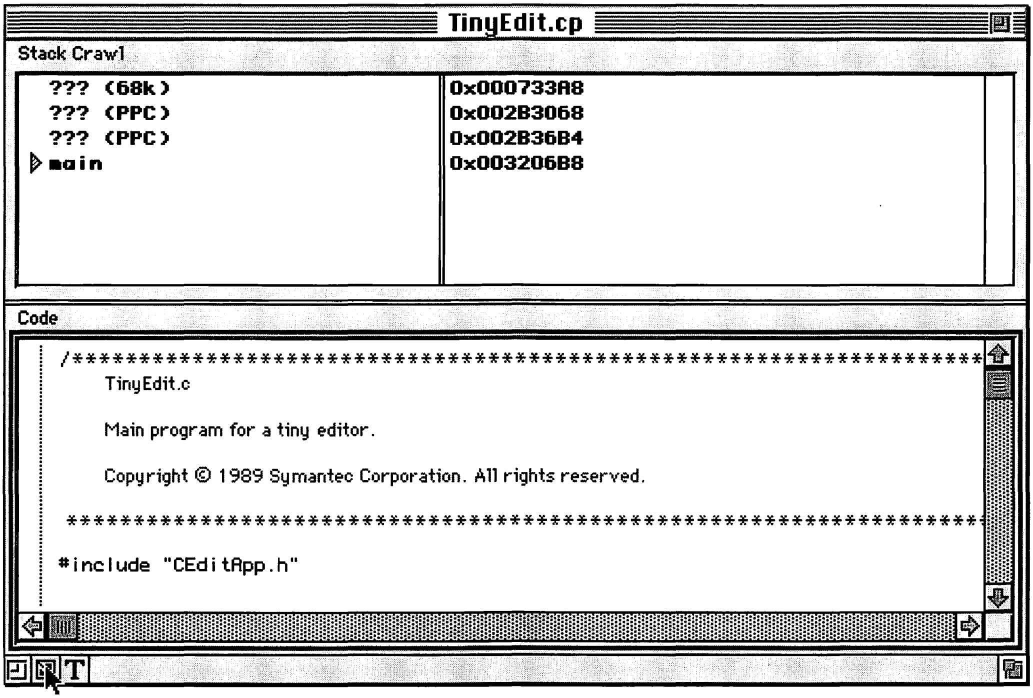
Drag cursor

```
??? (68k)    0x000733A8
??? (PPC)    0x002B3068
??? (PPC)    0x002B36B4
main         0x003206B8
```

```
/*********************
    TinyEdit.c

    Main program for a tiny edito

    Copyright © 1989 Symantec (

 *********************

#include "CEditApp.h"

void main()

{
    CEditApp     *editApp;

    editApp = new CEditApp
    editApp->IEditApp();
    editApp->Run();
    editApp->Exit();

}
```

Figure 6-4 Pane size drag bar

The panes' relative orientation can be controlled using the Orientation icon at the lower left of the window, as shown in Figure 6-5.

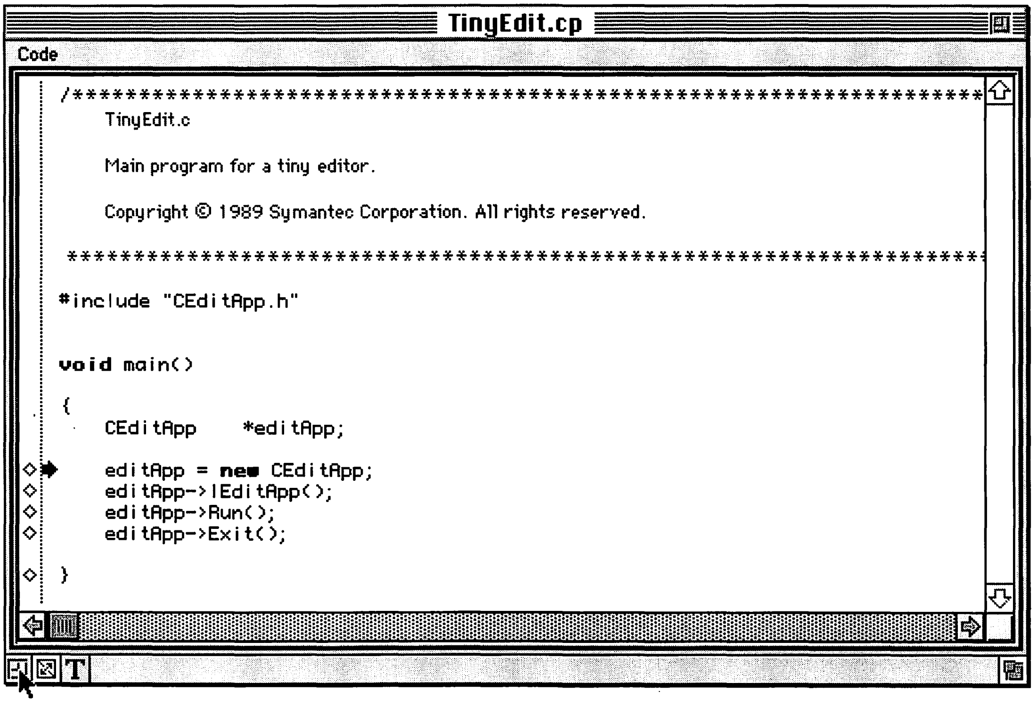


```
  ??? (68k)        0x000733A8
  ??? (PPC)        0x002B3068
  ??? (PPC)        0x002B36B4
▷ main             0x003206B8
```

```
/**********************************************************************
    TinyEdit.c

    Main program for a tiny editor.

    Copyright © 1989 Symantec Corporation. All rights reserved.

 **********************************************************************

#include "CEditApp.h"
```

Figure 6-5 Effect of clicking the Orientation icon

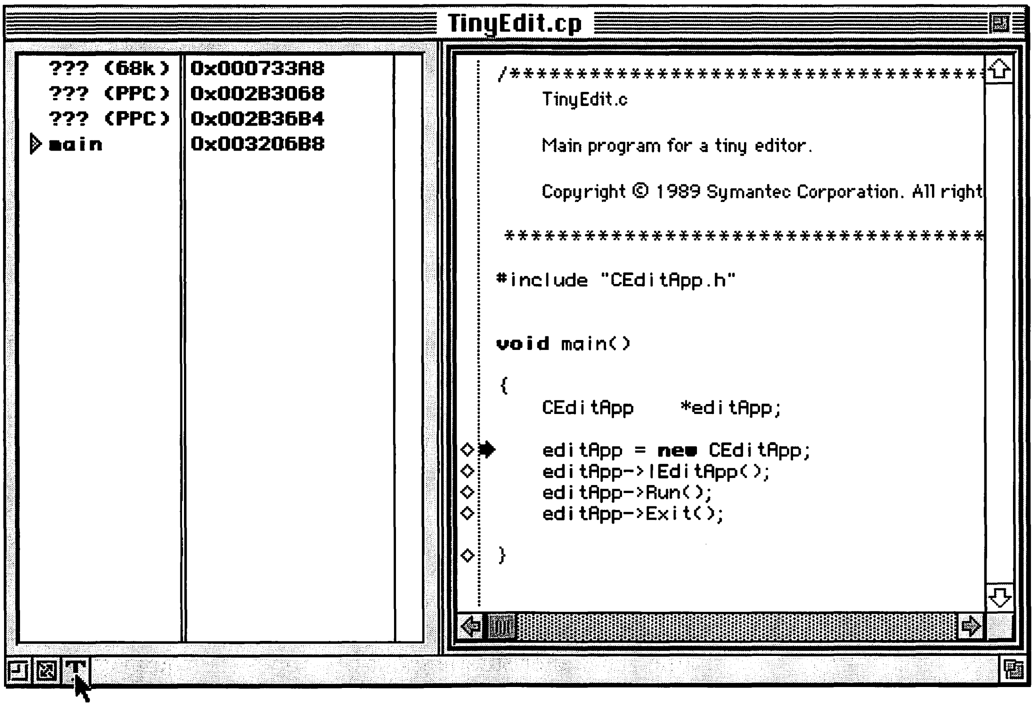
You can hide either of the panes by using the Zoom icon, as shown in Figure 6-6.



```
/*******************************************************************
    TinyEdit.c

    Main program for a tiny editor.

    Copyright © 1989 Symantec Corporation. All rights reserved.

 ********************************************************************

#include "CEditApp.h"


void main()
{
    CEditApp    *editApp;

    editApp = new CEditApp;
    editApp->IEditApp();
    editApp->Run();
    editApp->Exit();

}
```

Figure 6-6 Effect of clicking the Zoom icon

You can hide the titles of the panes by using the Titles icon, as shown in Figure 6-7.



```
??? (68k)   0x000733A8
??? (PPC)   0x002B3068
??? (PPC)   0x002B36B4
main        0x003206B8
```

```
/*************************************
    TinyEdit.c

    Main program for a tiny editor.

    Copyright © 1989 Symantec Corporation. All right

 ************************************

#include "CEditApp.h"


void main()

{
    CEditApp    *editApp;

    editApp = new CEditApp;
    editApp->IEditApp();
    editApp->Run();
    editApp->Exit();

}
```

Figure 6-7 Effect of clicking the Titles icon

Scrolling in the Main window
To scroll to a specific line in your code:

1. Choose **Go To Line** from the Debugger's **Source** menu (Command-Comma).

2. When the dialog box is displayed, type the number of the line to which you want to go.

3. Click Go To.

To scroll to a marker in your code:

1. Choose **Go To Marker** from the Debugger's **Source** menu (Command-Option-Comma).

2. Click the marker you want when the dialog box appears.

3. Click Go To.

   You can also choose a marker from the **Markers** pop-up menu by Command-clicking the title bar.

   To learn more about markers, see Chapter 19, "The Editor Window."

To print the file displayed in the Code pane, choose **Print** from the Debugger's **File** menu. To edit the file in the Code pane, choose **Edit** from the **Source** menu (Command-E). The Project Manager comes to the foreground and opens an Editor window for the file.

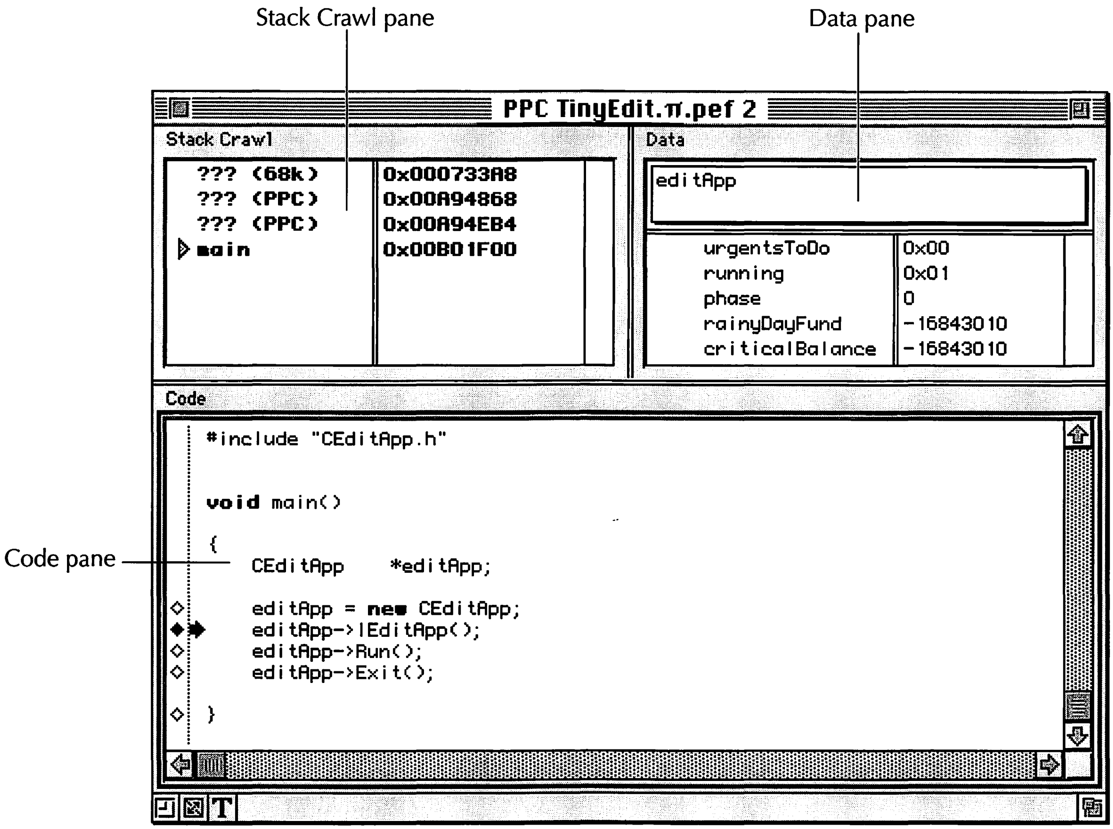
To open a different source file in the Code pane, click that source file in the Project Manager, then choose **Debug File** from the Project Manager's **Project** menu (Command-I).

The current statement arrow in the Code pane always points at the statement the Debugger is about to execute. Initially it points at the first executable statement in the code. The current statement arrow is hollow when there are instructions left to execute in a statement. A single line in the source code may be compiled into several assembly instructions. You may see hollow arrows when the statement is making an assignment, in a `for` statement, or cleaning up the stack after stepping out of a function. You may also see a hollow right angle arrow next to a line that corresponds to an active stack frame.

The Main debugger window contains no close box and remains open throughout a debugging session.

Debug Browser windows

You can also open auxiliary Debug Browser windows using the **New Browser** command in the Debugger's **File** menu (Command-N). An auxiliary Debug Browser window can contain up to three panes: a Code pane to examine the code, a Stack Crawl pane to examine the call chain of a function, and a Data pane to examine the values of expressions, as shown in Figure 6-8.



PPC TinyEdit.π.pef 2

```
??? (68k)     0x000733A8
??? (PPC)     0x00A94868
??? (PPC)     0x00A94EB4
▷ main        0x00B01F00
```

```
editApp

urgentsToDo      0x00
running          0x01
phase            0
rainyDayFund     -16843010
criticalBalance  -16843010
```

```
  #include "CEditApp.h"

  void main()

  {
      CEditApp    *editApp;

◇     editApp = new CEditApp;
◆➧    editApp->IEditApp();
◇     editApp->Run();
◇     editApp->Exit();

◇ }
```

Figure 6-8 An auxiliary Debug Browser window with expressions

Except for the Data pane and the close box, an auxiliary Debug Browser window is identical to the Main debugger window. The purpose of having auxiliary windows is to give the user as much flexibility as possible. Expressions can be examined only in an auxiliary window.

For further details concerning the Debug Browser window, see Chapter 24, "The Debugger Windows."

Stepping Through Code

The Debugger uses six commands to control execution. To make it easier to debug applications, you can invoke these commands in any of three different ways: using the buttons in the Debugger's Control palette, choosing commands from the **Debug** menu, or using Command-key equivalents.

The buttons in the Control palette also serve as status indicators. When a program is running, the **Go** button is lit. When the application is stopped, the **Stop** button is lit. Remember that an application can still be running even if the Main or Debug Browser window is frontmost.

To step through code, do any of the following:

- Use the buttons on the Debugger's Control palette, shown in Figure 6-9.

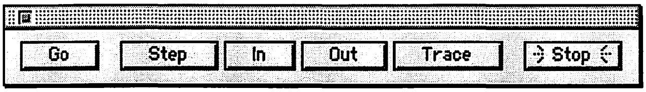


Figure 6-9 Debugger's Control palette with the Stop button lit

- Use the commands in the Debugger's **Debug** menu.
- Use the equivalent Command-keys listed in Table 6-1.

Table 6-1 Key combinations for Control palette buttons

Control palette button	Command-key equivalent
Go	Command-G
Step	Command-S
In	Command-I
Out	Command-O
Trace	Command-T
Stop	Command-.

To have the Debugger run code, click **Go** in the Control palette. Your code runs until it reaches a breakpoint, until it hits an exception (such as an illegal instruction), or until you stop it. If your application is already running, the **Go** command brings it to the front. To stop execution of code, click **Stop** in the Control palette.

To execute code line by line, click **Trace** in the Control palette. In all but one case, the execution continues to the next statement, even if the next statement is in another function. The exception occurs when the program counter steps into some code for which the Debugger does not have the source text. This typically happens when the routine steps into a function that is implemented in a library.

For a brief period, the execution is not really in the application but is in the library instead. You will not see the current statement arrow, but the name of the current function (if the Debugger can determine it) will still be visible in the call chain in the Stack Crawl pane.

To have the Debugger remain in the current function after executing the current statement, click **Step** in the Control palette. If you are at the end of a function, **Step** returns to the calling function. Use **Step** when you want to execute statements within a function without falling into the function being called.

Note

> **Step** will fall into the function being called if it contains a breakpoint.

To have the Debugger execute **Trace** commands until the execution falls into a function, click **In** in the Control palette. This command is useful when you want to skip over a set of assignments to fall into the next function call. If **Step In** reaches the last statement of the current function without falling into another function, it will stop immediately after the function returns.

To have the Debugger execute **Step** commands until the execution falls out of the current routine, click **Out** in the Control palette. This operation can be slow if there is much to be done, but it is a sure way of leaving the current routine.

To move through a block of code quickly:

1. Select a line.
2. Choose **Go Until Here** from the **Debug** menu (Command-H).

The Debugger goes as soon as the command is selected. This command has the same effect as setting a temporary breakpoint at the selected line, that is, it starts execution of your code and stops at the selected line.

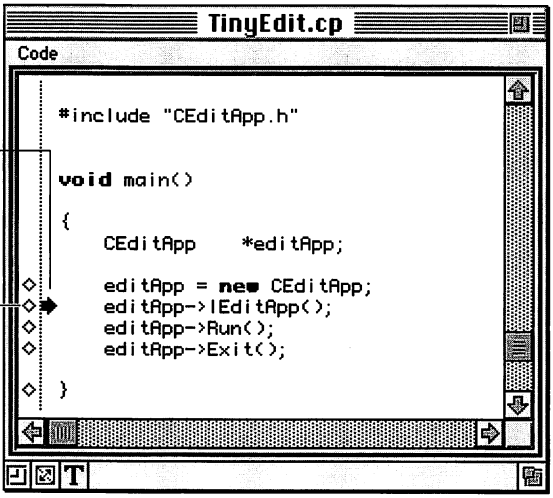
To jump ahead to a selected line without executing any intervening code, choose **Skip To Here** from the **Debug** menu. This allows you to skip over code that you know contains bugs but that is not crucial to the rest of the code's operation.

Note

Use this feature with caution, especially when debugging optimized code.

Setting Breakpoints

You set breakpoints in the Code pane. The empty diamonds that appear to the left of each executable statement indicate the places in the code at which you are permitted to set breakpoints, as shown in Figure 6-10.



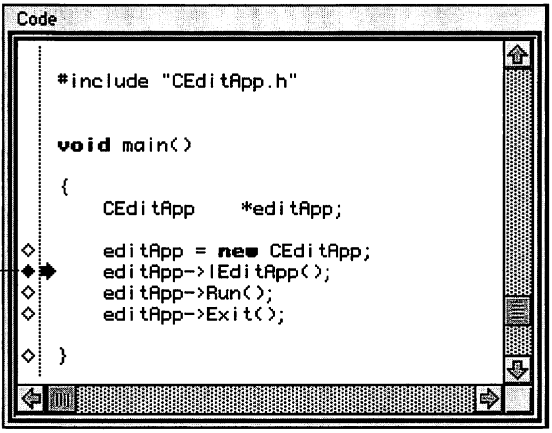
```
#include "CEditApp.h"

void main()
{
    CEditApp    *editApp;

    editApp = new CEditApp;
    editApp->IEditApp();
    editApp->Run();
    editApp->Exit();
}
```

Current statement indicator

A breakpoint diamond

Figure 6-10 Initial state of the Debugger Code pane

When code is running, the Debugger stops just before executing the first statement where a breakpoint has been set. A set breakpoint is indicated by a filled diamond to the left of the statement. The current statement arrow points at that statement, as shown in Figure 6-11.



```
#include "CEditApp.h"

void main()
{
    CEditApp    *editApp;

    editApp = new CEditApp;
    editApp->IEditApp();
    editApp->Run();
    editApp->Exit();
}
```

Filled diamond showing a breakpoint has been set

Figure 6-11 Debugger Code pane stopped at a breakpoint

The Symantec Debugger lets you set two kinds of breakpoints: simple breakpoints and temporary breakpoints.

The Debugger always stops execution at a simple breakpoint and a temporary breakpoint. At a temporary breakpoint, it also clears the breakpoint so execution will not stop there the next time.

Simple breakpoints

To set a simple breakpoint, do one of the following:

- Click a statement marker diamond.
- Click in the line to select it, then choose **Set Breakpoint** from the **Source** menu (Command-B).

  The diamond changes from hollow to filled to indicate that a breakpoint has been set.

The breakpoint remains set until you clear it.

Temporary breakpoints

To set a temporary breakpoint, do one of the following:

- Hold down the Option key as you set the breakpoint.

- Choose **Go Until Here** from the **Debug** menu (Command-H).

  The Debugger starts running the code and continues execution until it hits a breakpoint. Temporary breakpoints are cleared as soon as they are hit.

To clear a breakpoint, do one of the following:

- Click the filled diamond.
- Select the line and choose **Clear Breakpoint** from the **Debug** menu (Command-B).

  Clear Breakpoint is available for a selected line if a breakpoint has been set. If one has not been set for a selected line, this command is displayed as **Set Breakpoint**.

  The hollow diamond indicates that no breakpoint is set.

To clear every breakpoint in a project, choose **Clear All Breakpoints** from the **Source** menu.

Going until the next breakpoint
To advance the execution of your code to the next breakpoint, click the **Go** button in the Debugger's Control palette. The Debugger stops executing the code just before the next statement that is marked with a breakpoint. If the breakpoint is a simple breakpoint, it will remain set. If it is a temporary breakpoint, it will be cleared.

Examining the Call Chain

The call chain is the sequence of functions that were called to get to the current function. You can access the functions in the call chain through the Stack Crawl pane.

To examine the call chain of the current function, select the Main debugger window or a Debug Browser window. To examine the variables in a function in the call chain, click the triangle to the left of the variable name, as shown in Figure 6-12.

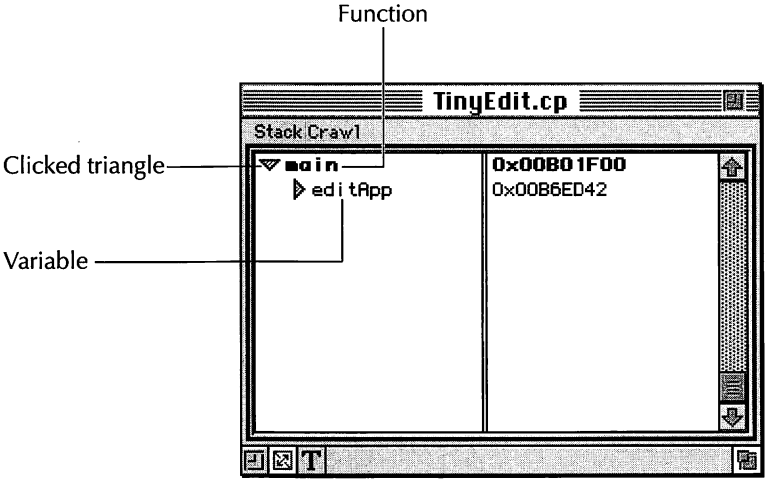
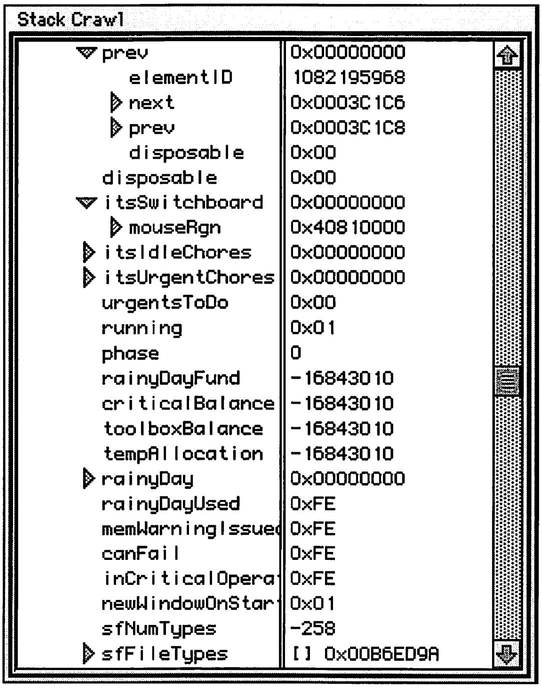


Figure 6-12 Displaying the variables in a function in the call chain

To examine the fields in a structure, class, or array in a function in the call chain, click the triangle to the left of the structure, class, or array name, as shown in Figure 6-13.



Stack Crawl

```
▽ prev                  0x00000000
     elementID          1082195968
   ▷ next               0x0003C1C6
   ▷ prev               0x0003C1C8
     disposable         0x00
  disposable            0x00
▽ itsSwitchboard        0x00000000
   ▷ mouseRgn           0x40810000
▷ itsIdleChores         0x00000000
▷ itsUrgentChores       0x00000000
  urgentsToDo           0x00
  running               0x01
  phase                 0
  rainyDayFund          -16843010
  criticalBalance       -16843010
  toolboxBalance        -16843010
  tempAllocation        -16843010
▷ rainyDay              0x00000000
  rainyDayUsed          0xFE
  memWarningIssue       0xFE
  canFail               0xFE
  inCriticalOpera       0xFE
  newWindowOnStar       0x01
  sfNumTypes            -258
▷ sfFileTypes           [] 0x00B6ED9A
```

Figure 6-13 Displaying structure or array fields in a function in the call chain

To hide variables or fields, click the same triangle again.

To copy a selected structure, class, array, or field to the Data pane, choose **Copy to Data** from the **Data** menu (Command-D).

To change the value of a selected expression, type the new value in the entry field of the Data pane, and press Return or Enter.

Formatting

To change the format of a selected expression, choose a new format from the **Data** menu.

The available formats are shown in Table 6-2.

Table 6-2 Display formats available

Type	**Formats available** **(Default formats in italics)**
integer	*Signed Decimal*, Unsigned Decimal, Hexadecimal, Character
unsigned	*Unsigned Decimal*, Signed decimal, Hexadecimal, Character
pointer	*Pointer*, Address, Hexadecimal, C String, Pascal String
array	*Address*, C String, Pascal String
`struct`	Address
union	Address
function	Address
float	Floating Point
fixed	Fixed Point

Command-keys and samples of these formats are shown in Table 6-3.

Table 6-3 Display format examples

Format	Command-key	Example
Signed decimal	Command--	`4523345 or -1`
Unsigned decimal	Command-U	`4523345 or 65535`
Hexadecimal	Command-\	`0xA09E1487`
Character	Command-R	`'C' or 'TEXT'`
Pointer	Command-P	`0x007A7000`

Address	Command-A	`[]0x0009FE44`
		`or struct`
		`0x0008FC14`

C string	Command-`	`"abcdef\nghi\33"`
Pascal String	Command-'	`"\pabcdef\nghi\33"`
Floating Point	Command-F	`1961.0102`
Fixed	Command-;	`1961.0102`

The C string and Pascal string formats display nonprinting characters in backslash form. Whenever it can, the Debugger uses the built-in escape characters (`\n, \r, \b`); otherwise, it uses `\nn`, where `nn` is an octal value.

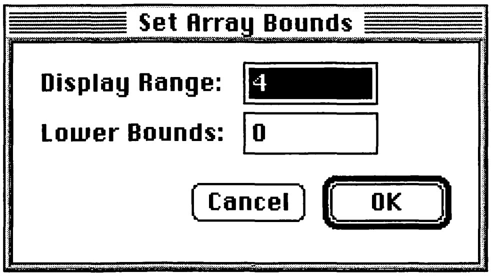
You can use typecasting to use formats that are not normally available. For example, to see the integer `i` as a C string, type the expression: `(char *) i`.

To see any pointer as an array, change its format to Address.

To set the bounds of an array:

1. Click the array and choose **Set Array Bounds** on the **Data** menu or double-click the array.

The **Set Array Bounds** dialog box in Figure 6-14 opens.



Set Array Bounds

Display Range: 4

Lower Bounds: 0

Figure 6-14 Set Array Bounds dialog box

2. Enter the number of items in the array and the base index of the array.

3. Click OK.

For more details on setting array bounds, see Chapter 25, "The Debugger Menus."

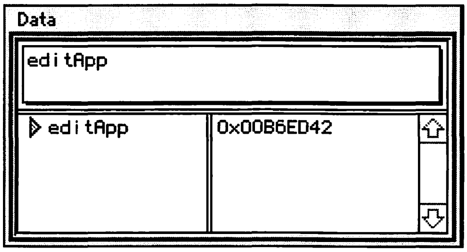
Analyzing Variables

You can examine and modify the values of variables in the Data pane. Expressions themselves are displayed in the left column of the pane and their values are displayed in the right column. You can enter any legal expression for the compiler you are using. Statements with side effects are locked by default.

For the Debugger to compile an expression that you want it to evaluate, it must know the context in which the expression is to be evaluated. The context of an expression is the block of code surrounding the expression when it is evaluated during execution.

To enter an expression in the Data pane, select the expression in the Code pane, then choose **Copy to Data** in the Debugger's **Edit** menu (Command-D). The Debugger compiles the expression in the context of the selected line and displays its value.

To enter an expression in the Data pane in the current context, type the expression in the entry field of the Data pane and press Enter or Return. Pressing Enter places the expression in the Data column and also leaves it selected in the entry field, as shown in Figure 6-15.

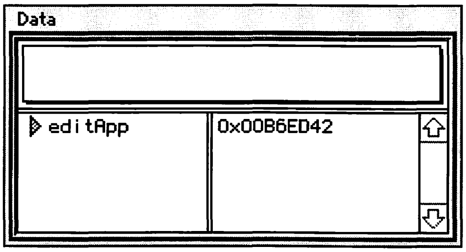


editApp

▷ editApp 0x00B6ED42

Figure 6-15 Pressing Enter to enter an expression

Pressing Return places the expression in the Data column and removes it from the entry field, as shown in Figure 6-16.



▷ editApp 0x00B6ED42

Figure 6-16 Pressing Return to enter an expression

The Debugger evaluates the expression in the current context.

To set the context of a selected expression in the Data pane that you want the Debugger to evaluate, do either of the following:

- Select the line in the Code pane that contains the occurrence of the expression that you want evaluated.
- Select a line in the Code pane and choose **Set Context** from the **Data** menu.

If you have not selected a line in the Code pane, the expression is evaluated in the context of the current statement.

To see the context of an expression selected in the Data pane, choose **Show Context** from the **Data** menu. The statement that is the context of the selected expression is highlighted in the Code pane.

To edit an expression that you want to evaluate:

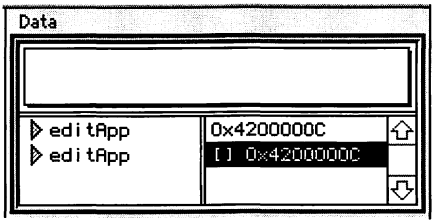
1. Type the expression in the entry field of the Data pane.

2. Press Enter.

To edit an expression that is already in the Data column of the Data pane, select the expression in the Data column. The expression is shown selected in the entry field, ready to be edited.

To change the context of an expression to the current context after you edit it, press Option or Command while pressing Enter.

When you edit an expression in the Data pane, the context is the same as when you first entered the expression. To enable you to compare an expression's value in different contexts, the Debugger lets you have multiple copies of the same expression in the Data pane, as shown in Figure 6-17.



▷ editApp	0x4200000C
▷ editApp	[] 0x4200000C

Figure 6-17 Multiple copies of an expression in the Data pane

To remove an expression from the Data pane:

1. Select the expression you want to remove.

2. Choose **Clear** from the **Edit** menu. Alternatively, press Clear or Esc.

To remove all of the expressions from the Data pane, choose **Clear All Expressions** from the **Data** menu.

Changing the values of variables

The Debugger lets you change the value of any expression that would be legal on the left side of an assignment statement. When you enter an expression, it is displayed in the left column of the Data pane and its value is shown in the right column. For information on changing the value or format of a variable, see the section "Examining the Call Chain," earlier in this chapter.

Evaluating expressions

The Debugger re-evaluates the expressions in the Data pane every time a program stops. An expression whose context is not in the current function is not re-evaluated and its value is cleared from the Data pane, unless it has global scope.

To examine the values of variables:

1. Set the contexts of the variables you want evaluated.

2. Enter the variables in the Data pane.

To prevent an expression from being evaluated:

1. Select the expression.

2. Choose **Locked** from the **Data** menu (Command-L). The Debugger locks the expression with a lock icon.

Locked variables are always displayed, even if they are no longer in context.

You can ensure that the Debugger always evaluates a variable in the context of the current statement by making the variable context-free.

This is useful if you are using the same variable name in several routines and you would like to see the value of the variable whenever you enter one of those routines.

To make a selected variable context-free, choose **Context-free** from the **Data** menu (Command-K). The Debugger marks the variable with a small arrow.

Ending a Debugging Session

To end a debugging session, do one of the following:

- Quit the application.
- Choose **Quit** from the Debugger's **File** menu.
- Choose **ExitToShell** from the Debugger's **Debug** menu.

Preferences and Options

Preferences apply to all projects. Options are project-specific. To learn how to set preferences for the Debugger, see Chapter 25, "The Debugger Menus." To learn how to set options for the Debugger, see Chapter 18, "The Project Menu."

Creating a User Interface with VA

7

Visual Architect is the preferred platform for designing and implementing the user interface for Symantec C++ applications. It acts as a bridge between the programmer and the powerful and diverse capabilities of the THINK Class Library, allowing you to create a complete user interface with a minimal investment of time and energy. This chapter describes the basic functions of Visual Architect.

At this point, you should be comfortable with creating a project using the VA Application project models, editing code, building and debugging your project, and viewing classes. Visual Architect, in fact, can be used at any point in the development cycle following creation of a project.

This chapter describes how to launch Visual Architect and use it to construct and edit views, panes, menus, commands, classes, and Balloon Help. The generation of source code for a project is also covered.

Introduction

You use Visual Architect in conjunction with other Symantec C++ for Power Macintosh tools to construct an application's user interface. Visual Architect lets you develop code and resources using interactive, visual tools, rather than by writing in C++. Using the THINK Class Library, Visual Architect generates customizable source code. Also, Visual Architect lets you test user interface elements without having to build a project in the Symantec Project Manager.

When to use Visual Architect

You can use Visual Architect at any stage in the development of an application—for example, when you are beginning a design or fine-tuning an application's user interface at the end of a project. Typically, you work with Visual Architect throughout the development of an application. You build the application incrementally by going back and forth between Visual Architect and the other Symantec Project Manager tools.

Files produced

Visual Architect maintains one resource file and a set of source files. The resource file, by default named `Visual Architect.rsrc`, is automatically added to a project when you create it with either of the two VA Application project models. The source files are `.cp` and `.h` files. They are written in C++ and automatically are added to a project as they are created by Visual Architect. You are encouraged, and typically will find it necessary, to edit these files in the course of developing an application's user interface.

Note

Studying the commented code generated by Visual Architect helps you understand the structure and implementation of classes in the THINK Class Library.

Launching Visual Architect

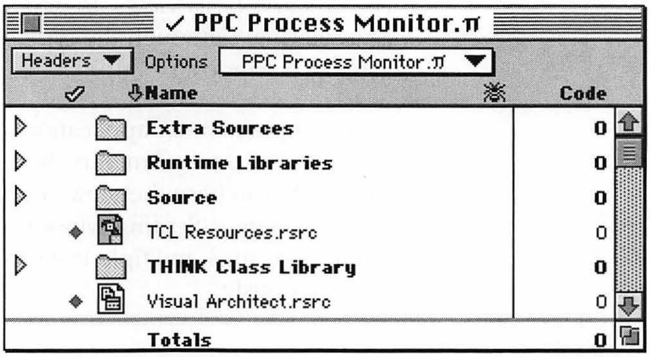
To launch Visual Architect:

1. Create a project using either of the two VA Application project models in the Symantec Project Manager, as described in Chapter 3, "Starting a Project."

Note

You can also choose any customized Visual Architect project model. See Chapter 16, "The File Menu," for details on creating your own project models.

Figure 7-1 shows the resource file named `Visual Architect.rsrc` that is included in your project.



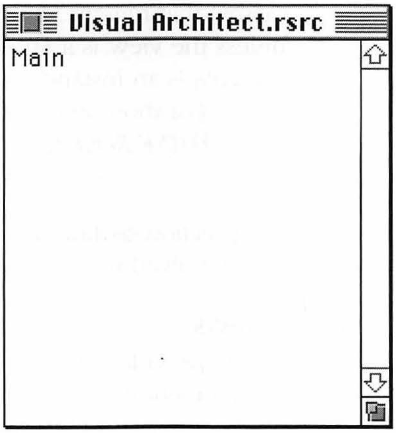
✓ PPC Process Monitor.π

✓	Name	Code
▷	**Extra Sources**	0
▷	**Runtime Libraries**	0
▷	**Source**	0
◆	TCL Resources.rsrc	0
▷	**THINK Class Library**	0
◆	Visual Architect.rsrc	0
	Totals	0

Figure 7-1 Project Manager Project window and selected Visual Architect.rsrc entry

2. Double-click `Visual Architect.rsrc` in the Project window.

   Visual Architect launches and the View List window opens (Figure 7-2).



Visual Architect.rsrc

Main

Figure 7-2 Visual Architect View List window

The View List window in Visual Architect shows a list of the views defined in the `Visual Architect.rsrc` file. Whenever you create a new project in the Symantec Project Manager using either of the VA Application project models, the `Visual Architect.rsrc` file contains a default view called Main.

Constructing Views

Building an application's user interface typically begins with construction of its views. In general terms, a view can be thought of as a window (although the type of view known as a Subview is not actually a window).

Views implement an application's windows, dialog boxes, floating palettes, and tear-off menus. When you create a view in Visual Architect, you base the view on one of several view types, such as modal dialog or floating window, then customize it. When the application runs, multiple instances of that view can be open simultaneously.

Note

The term "view" has different meanings in the THINK Class Library and in Visual Architect. In the THINK Class Library, a view is an instance of a class derived from CView. CView is the base class from which all visual entity classes are derived.

In Visual Architect, however, "view" refers to the set of elements (resources and THINK Class Library classes) that together implement your window (unless the view is a Subview). One of these elements is an instance of a class derived from CView. For more information on CView, see the online *THINK Reference.*

The following section outlines the different view types and describes the processes involved in constructing views.

Types of views

Visual Architect provides nine view types. All but the last view type (Subview) implement windows.

Main Window

Main Window views implement windows that serve as the center of the user's attention. These views typically are used to display the contents of a document, either text or graphics.

Note

The Main view discussed earlier is an instance of a Main Window view.

Window
Window views implement windows that are auxiliary to the application's Main Window views.

Floating Window
Floating Window views implement palettes containing drawing tools, colors, patterns, and so on. These views are drawn in front of all nonfloating windows.

Tear-off Menu
Tear-off Menu views are similar to Floating Window views except that they implement menus that can be "torn off" from the menu bar and placed anywhere on the screen.

Dialog
Dialog views are used to implement general-purpose modeless dialog boxes.

Modal Dialog
Modal Dialog views are used to implement general-purpose modal dialog boxes.

New... Dialog
New... Dialog views are used to implement a special type of modal dialog box, with which the user specifies a document type. This dialog box is displayed in response to a user choosing **New** from the **File** menu when more than one document type is created by the application.

Splash Screen
Splash Screen views are used to implement a special type of modeless dialog box, which is only displayed when the application is starting up.

Subview
Subview views are a special type of view used to implement panoramas within other views. This view type does not implement a window.

Most applications define one or more Main Window views, which serve as the central windows for the application and can be modified to suit your needs. In addition, you can create views to implement additional windows, dialog boxes, palettes, and pop-up menus, and modify them accordingly.

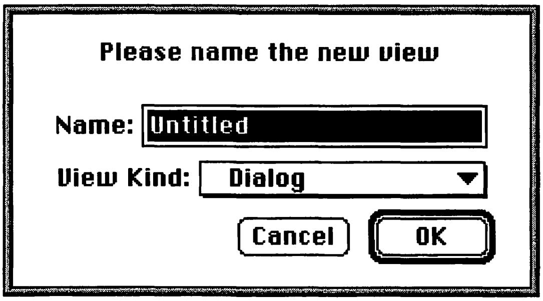
Creating a view

While the same general procedure is used to create all nine types of views, the Dialog view is used as an example of the process in this section.

To add a new Dialog view to an application:

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

   The **New View** dialog box opens, as shown in Figure 7-3, in which you are prompted to provide basic information about the view.

   Please name the new view

   Name: Untitled

   View Kind: Dialog

   Figure 7-3 New View dialog box

2. Type a name for the new view in the Name field.

   The name you specify must be unique within the application.

3. Choose a type of view from the **View Kind** pop-up menu (Figure 7-4).

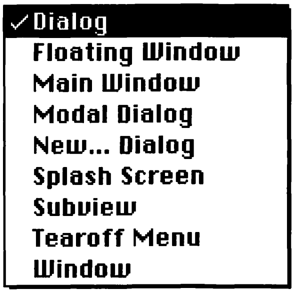
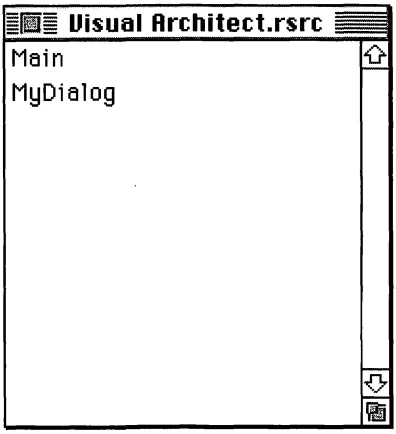


Figure 7-4 View Kind pop-up menu

4. Click OK.

   The name of the new view is displayed in the View List window (Figure 7-5).



Main
MyDialog

Figure 7-5 View List window, with a new view named MyDialog

In addition, a View Edit window for this new view opens (Figure 7-6), using the title supplied in the **New View** dialog box.

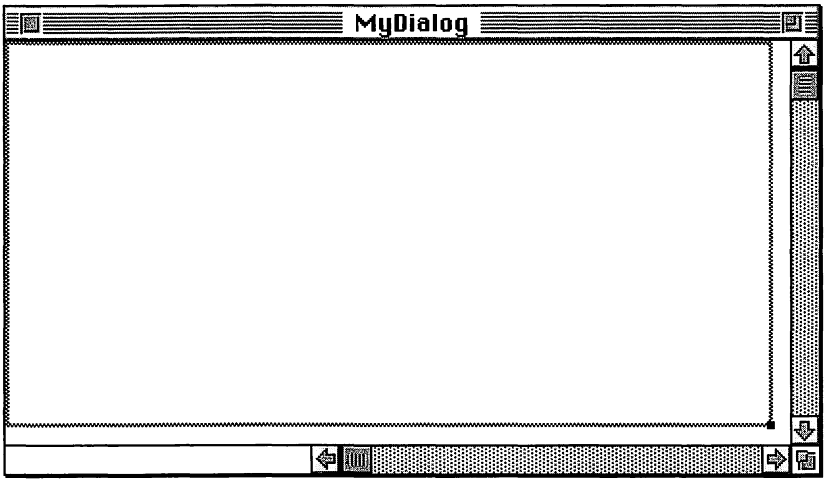


Figure 7-6 View Edit window for the MyDialog Dialog view

The View Edit window resembles the MacDraw™ drawing window. You can construct the elements within a view using the View Edit window. The section "Creating Panes," later in this chapter, describes this process.

Changing the attributes of a view

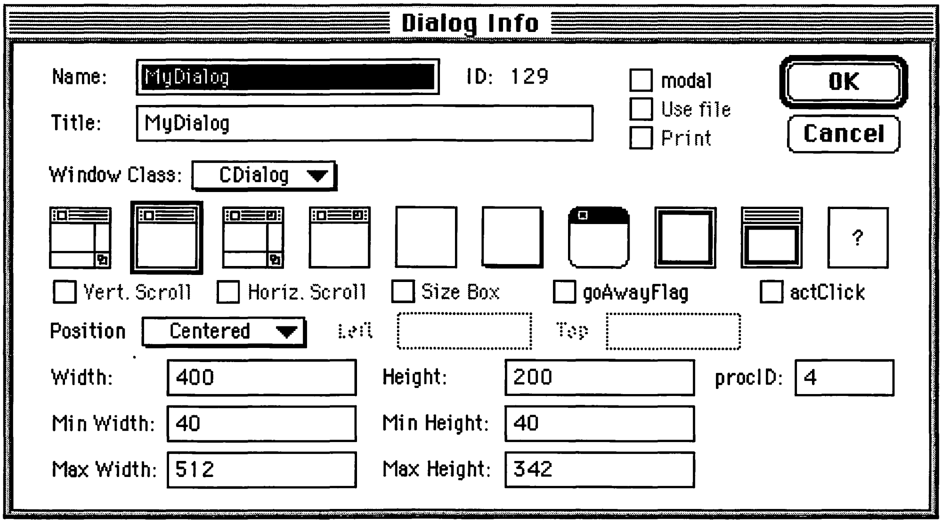
The general attributes of a view, such as its window type and size, are set to default values when you create the view. To change these attributes, use the view's **Info** dialog box. The steps involved in changing view attributes are similar for the different view types. The following steps demonstrate how to edit Dialog views.

To open the **Info** dialog box for a view:

1. Open the view's View Edit window by double-clicking the view's name in the View List window.

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

The **Info** dialog box opens, as shown in Figure 7-7, for a Dialog view.



Dialog Info

Name: MyDialog ID: 129

Title: MyDialog

Window Class: CDialog

□ modal
□ Use file
□ Print

□ Vert. Scroll □ Horiz. Scroll □ Size Box □ goAwayFlag □ actClick

Position: Centered Left Top

Width: 400 Height: 200 procID: 4

Min Width: 40 Min Height: 40

Max Width: 512 Max Height: 342

Figure 7-7 Dialog Info dialog box

Note

The format of the **Info** dialog box is identical for Dialog, Modal Dialog, New... Dialog, Splash Screen, and Window view types. The **Info** dialog boxes for other view types differ. See Chapter 31, “Visual Architect View Menu,” for details.

Naming the view

Views are named when they are created, but the name can be changed. To change the name of a view, type the new name in the Name textbox.

Naming the view’s window

By default, the title of the view’s window when the application runs is the same as the view name. To change the view’s window title, type the new name in the Title textbox.

Setting the window type

You can set the window type used for a view's window in one of two ways:

- Select one of the ten window type icons.
- Type the name of the window type in the procID textbox.

Setting the window position

You can choose one of three positions for a view's window when it appears in the running application. If it should appear in a fixed position, its position is determined by the values in the Left and Top textboxes. If it should be centered, it is centered on the main screen. If it should be staggered, it appears down and to the right of the previous window. To set the positioning of the view's window, choose **Fixed**, **Centered**, or **Staggered** from the **Position** pop-up menu.

Setting the window size

The size of the view's window can be set numerically or graphically. To set the window size numerically, type the appropriate numbers in the Width and Height textboxes.

To set the size of a view's window graphically, use the View Edit window for that view:

1. Bring the View Edit window to the front.

2. Drag the sizing handle of the view's portRect (Figure 7-8).

Note

You may first need to increase the size of the View Edit window.

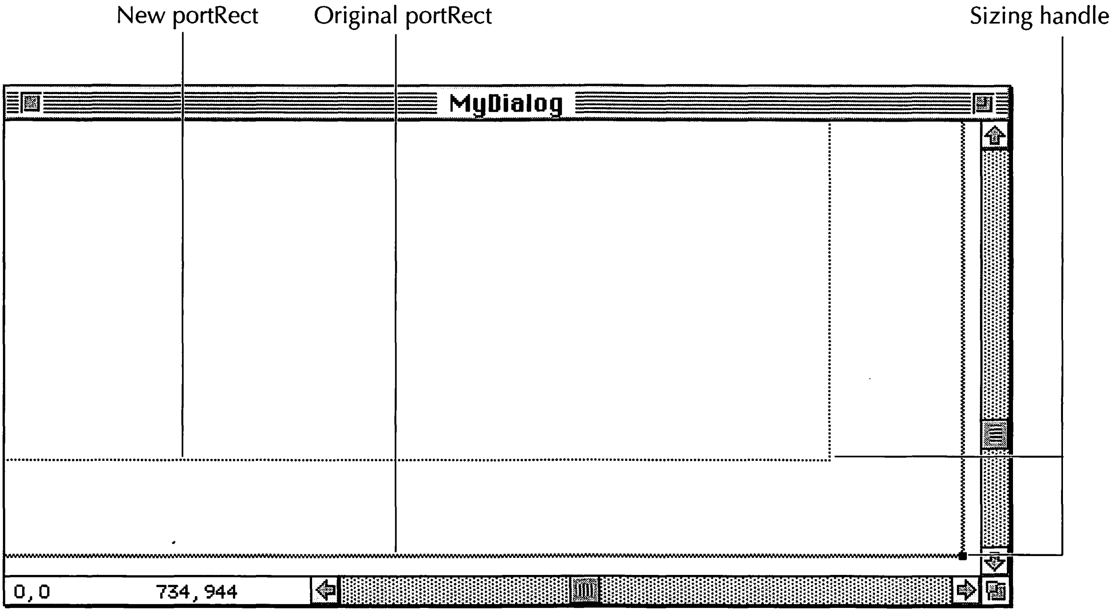


Figure 7-8 Resizing a view using the sizing handle

Creating Panes

You can now create the control and graphical elements to include in the view. These elements are called panes.

Panes are graphical elements that provide visual information, control capabilities, or both.

Note

> Panes are implementations of the THINK Class Library class CPane. All drawing performed by classes in the THINK Class Library occurs within panes, each of which has its own drawing environment. CPane, a visual element class, is derived from CView.

Without any panes, a view is only a blank window. A view must have panes to have functionality. Panes get their functionality by having commands attached to them. Commands are explained later in this chapter.

Twenty different pane classes are available in Visual Architect. These allow you to display static text and graphic elements as well as implement dialog (editable) text fields, buttons, radio buttons, check boxes, scroll bars, pop-up menus, and scrolling text and graphics fields. The individual panes' classes are described in Chapter 34, "Visual Architect Tools Menu."

Adding a pane to a view

You add panes to views using the Tool palette shown in Figure 7-9.

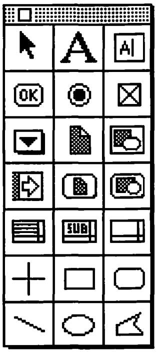


Figure 7-9 Tool palette

You access the Tool palette in either of two ways:

- Choose individual tools from the **Tools** menu.
- Click the **Tools** menu and tear off the Tool palette by dragging it beyond the edge of the menu. An outline of the Tool palette is displayed, which you can position anywhere on the screen.

Note

You can reposition a pane after it is created by dragging it to its new position.

All tools in the Tool palette except the Select tool, correspond to a pane class. The Select tool is used to select one or more panes already added to a view, as described in the section "Selecting a pane" later in this chapter.

You follow similar steps to create panes for the different pane classes. The following directions describe the process for a few pane classes.

Adding a text pane

Views can have two types of text panes: static text and dialog text. Static text panes are used to provide information to the user. Dialog text panes are used to obtain information from the user.

To add a static text pane to a view:

1. Open the View Edit window for the view.
2. Choose the Static Text tool from the Tool palette.
3. Click the cursor in the View Edit window to position the static text pane.

   A blinking insertion point indicates that you should enter the text for the pane.
4. Type the text for the static text pane.

To add a dialog text pane to a view:

1. Open the View Edit window for the view.
2. Choose the Dialog Text tool from the Tool palette.
3. Click the cursor in the View Edit window to position the dialog text pane.

   A dialog textbox opens.

Note

The default text for dialog text panes is specified in the Pane Info window, described in the "Changing pane attributes" section, later in this chapter.

Adding a button pane

Button panes implement the standard Macintosh push buttons. To add a button pane to a view:

1. Open the View Edit window for the view.

2. Choose the Button tool from the Tool palette.

3. Click the cursor in the View Edit window to position the button pane.

   A button is displayed with some default text highlighted.

4. If you want to rename the button, type the text for the name of the button.

Adding a graphic element pane

Views often contain graphic elements that divide the view's window into functional areas, direct attention, or serve as decoration. Visual Architect provides six tools for creating basic graphics panes: Straight Line, Unconstrained Line, Rectangle, Rounded Rectangle, Oval, and Polygon. All these tools function in the standard fashion associated with drawing programs such as MacDraw™.

To add a line pane to a view:

1. Open the View Edit window for the view.

2. Choose either the Straight Line or the Unconstrained Line tool from the Tool palette.

   The Straight Line tool creates a line orientation as a multiple of 45°; the Unconstrained Line tool permits any line orientation.

3. Drag the cursor in the View Edit window to set the starting and ending points for the line.

To add a rectangle or oval pane to a view:

1. Open the View Edit window for the view.

2. Choose the Rectangle, Rounded Rectangle, or Oval tool from the Tool palette.

3. Drag the cursor in the View Edit window to set two opposing corners of the bounding box containing the pane.

Selecting a pane

Once a pane has been added to a view, you need to select it to change its location or its attributes. Select panes using the Select tool from the Tool palette. To select a pane:

1. Open the View Edit window for the view.

2. Choose the Select tool from the Tool palette.

   Choosing the Select tool changes the cursor to an arrow, the standard Macintosh selection cursor.

3. Select the pane.

Depending on the number of panes you want to select, you can select a pane in one of these ways:

- For a single pane, click the pane.

- For multiple panes, click the panes while holding down the Shift key. Alternatively, click an empty part of the drawing area, hold the mouse button down, and drag the cursor until the selection rectangle encompasses the panes you want to select.

If the **Lazy Select** command is enabled in the **Options** menu, the selection rectangle only has to touch a pane for it to be selected.

Deleting a pane from a view

To delete a pane from a view:

1. Open the View Edit window for the view.

2. Select the pane.

3. Click Delete.

Changing pane attributes

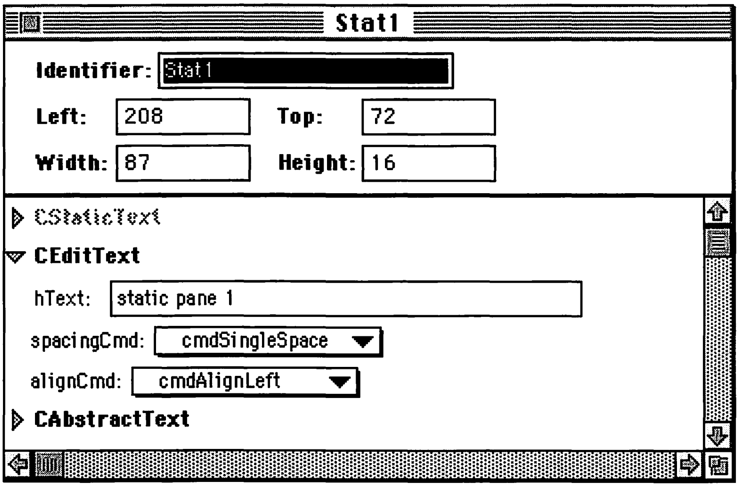
You can change many attributes that define a pane. This process involves changing data members in the class hierarchy of the pane. Use the Pane Info window to change attributes.

To open the Pane Info window for a pane:

1. Open the View Edit window for the view.

2. Select the pane and choose **Pane Info** from the **Pane** menu. Alternatively, double-click the pane in the View Edit window.

Pane Info windows vary in appearance depending on the pane class. All of them are organized similarly, however. The Pane Info window for a static text pane is shown in Figure 7-10.



Stat1

Identifier: Stat1

Left: 208 Top: 72

Width: 87 Height: 16

CStaticText

CEditText

hText: static pane 1

spacingCmd: cmdSingleSpace

alignCmd: cmdAlignLeft

CAbstractText

Figure 7-10 Pane Info window

A Pane Info window is associated with a particular pane. You can have multiple Pane Info windows open simultaneously, each reflecting information about a different pane. When you close a view, any Pane Info windows associated with panes in that view are closed automatically.

Changes made in the Pane Info window are reflected immediately in the target pane. For example, if you type a value in the Width or Height textboxes for the CPane class, the size of the pane in the View Edit window changes as you type.

Editing the pane identifier

The title of the Pane Info window is the identifier for the pane. To edit a pane's identifier, enter changes in the Identifier textbox at the top of the Pane Info window.

Setting the pane size and position

To set the position of the pane relative to the view's window, type an appropriate number in the Left and Top textboxes at the top of the Pane Info window.

To set the size of the pane, type an appropriate number in the Width and Height textboxes at the top of the Pane Info window.

You can also change the size and position of a pane graphically within the View Edit window. To set the size of the pane graphically, drag the sizing handle for the pane, located in the lower-right corner of the pane. To set the position of the pane graphically, drag the pane to the new location.

Setting other pane attributes

All other pane attributes are specific to each pane class, and are changed using the lower portion of the Pane Info window. The lower portion shows the pane's class hierarchy, beginning with the outermost derived class of the pane and ending with the CView class.

The small triangles next to the class names let you access the contents of each class, which are displayed in a subarea below the class name, as shown in Figure 7-10. The triangles exist in two states: closed, when they point to the right, and open, when they point down. The class subarea contains the editable subset of the data members for that class. (See the online *THINK Reference* for definitions of these data members.)

To access the contents of a class, click the triangle next to the class when it is in the closed state. An area below the class name opens, revealing the contents of that class.

To hide the contents of a class, click the triangle next to the class when it is in the open state. The subarea below the class closes, hiding the contents of that class.

Editing the text in a dialog text pane

To edit the text in a dialog text pane:

1. Open the Pane Info window for the dialog text pane.

2. Click the triangle next to the CEditText class to display the editable data members of that class.

3. Make changes in the hText textbox.

Editing text in other panes
You can directly edit the text of static, edit, push button, radio button, or check box panes without using the **Pane Info** dialog box. Select the pane and press Return. You can edit or add any text in the pane, up to 32K characters. When you are finished, click outside the pane. Panes are resized automatically to fit the text.

Trying out a view

From within Visual Architect, you can try out a view to verify its look and feel in the running application. This method allows you to test the view without going through the complete development cycle that involves generating code with Visual Architect, updating the project and building the application in the Symantec Project Manager, and running the application.

To try out a view:

1. Open the View Edit window for the view.

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

A window opens, which shows how the view's window would look in the running application. Controls, such as buttons and scroll bars, and Balloon Help (described later in this chapter) are active in this window.

To close the "try out" window, choose **Close** from the **File** menu. Alternatively, click the window's close box, or the OK or Cancel button, if available.

Building Menus

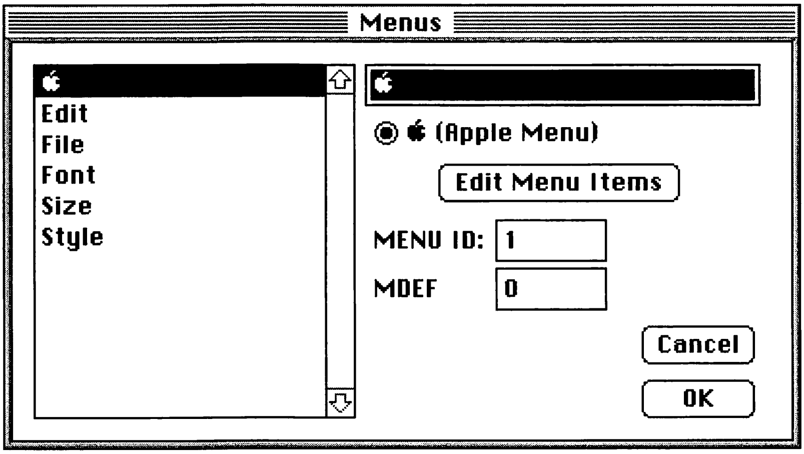
Visual Architect lets you create an application's menus. These menus can appear in the menu bar as standard menus or tear-off menus, or they can appear as submenus or pop-up menus.

As with ResEdit, you construct menus with Visual Architect using a graphical interface. In addition, Visual Architect lets you set up the commands that are sent by menu items. These commands are described in the section "Attaching Commands" later in this chapter.

Creating a menu

To create a new menu:

1. Choose **Menus** from the **Edit** menu to open the **Menus** dialog box (Figure 7-11).



● (Apple Menu)

MENU ID: 1

MDEF 0

Figure 7-11 Menus dialog box

2. Choose **New Menu** from the **Edit** menu (Command-K) or press Return.

   The textbox at the top of the dialog box clears and shows a blinking cursor.

3. Type the title of the new menu in the textbox.

   The new menu title is introduced in the menu list on the left side of the dialog box.

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

Deleting a menu

To delete a menu from the menu list:

1. Open the **Menus** dialog box by choosing **Menus** from the **Edit** menu.

2. Select the menu by clicking its title in the menu list on the left side of the dialog box.

3. Press Delete to remove the menu title from the menu list.

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

Adding a menu to the menu bar

To add an existing menu to the menu bar:

1. Choose **Menu Bar** from the **Edit** menu to open the **Menu Bar** dialog box (Figure 7-12).

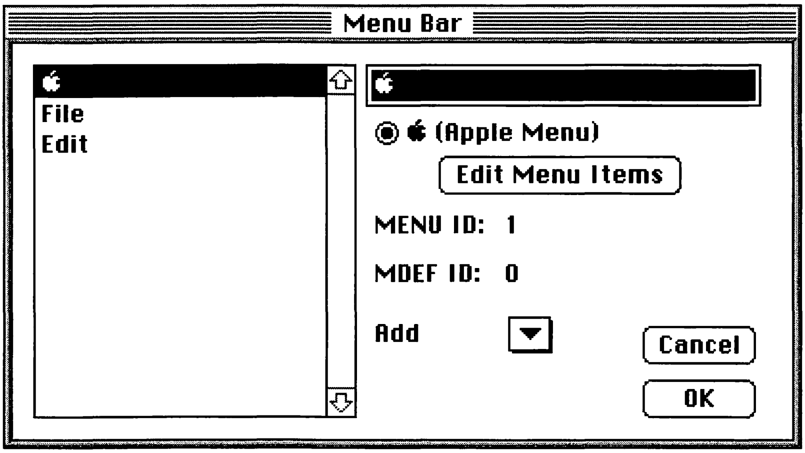


Figure 7-12 Menu Bar dialog box

Note

> The **Menus** dialog box lists all menus defined in the current `Visual Architect.rsrc` file, while the **Menu Bar** dialog box lists only those menus that have been placed in the menu bar. The two are otherwise similar in functionality.

2. Choose the menu you want to add from the **Add Menu** pop-up menu.

   The menu title is displayed in the menu list on the left side of the dialog box. You can reorder the menus within the menu bar at any time by dragging their titles within the menu list.

3. Click OK to close the **Menu Bar** dialog box.

Removing a menu from the menu bar

To remove a menu from the menu bar:

1. Open the **Menu Bar** dialog box by choosing **Menu Bar** from the **Edit** menu.

2. Select the menu by clicking its title in the menu list on the left side of the dialog box.

3. Press Delete.

   The menu is removed from the menu bar, but it is not deleted from the `Visual Architect.rsrc` file.

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

Adding and removing menu items

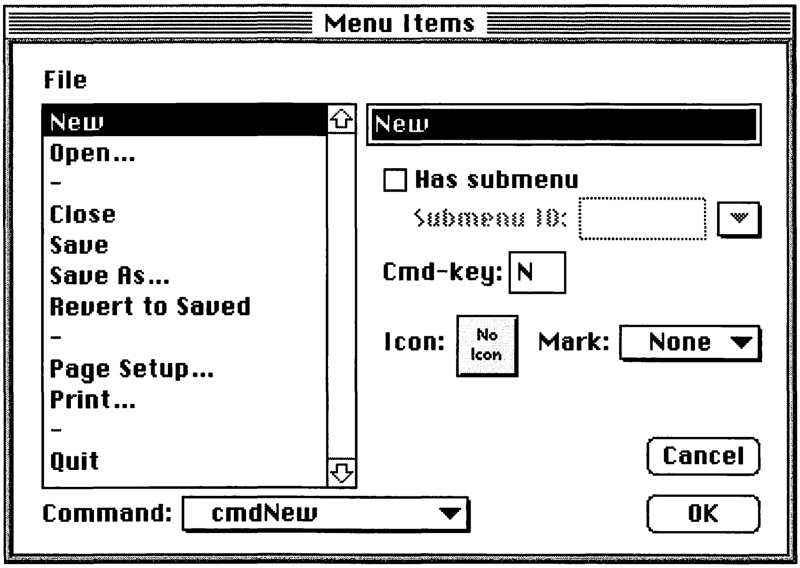
To add a menu item to a menu:

1. Open the **Menus** dialog box by choosing **Menus** from the **Edit** menu (or use the **Menu Bar** dialog box, if the menu is in the menu bar).

2. Select the menu by clicking its title in the menu list on the left side of the dialog box.

3. Click Edit Menu Items.

The **Menu Items** dialog box opens (Figure 7-13).



New

□ Has submenu

Submenu ID:

Cmd-key: N

Icon: No Icon

Mark: None

Command: cmdNew

Figure 7-13 Menu Items dialog box

4. Choose **New Menu Item** from the **Edit** menu (Command-K) or press Return.

   The textbox at the top of the dialog box clears and shows a blinking cursor.

5. Type the name of the menu item in the textbox.

   The new menu item is introduced in the menu item list on the left side of the dialog box.

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

To remove an item from a menu:

1. Open the **Menu Items** dialog box.
2. Select the item in the menu item list on the left side of the dialog box.
3. Press Delete to remove the item from the menu item list.
4. Click OK to close the **Menu Items** dialog box.

Setting a menu item's command key

To assign a keyboard shortcut key to a menu item:

1. Open the **Menu Items** dialog box.

2. Select the item in the menu item list on the left side of the dialog box.

3. Type the shortcut character into the Command-key textbox.

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

Creating a submenu

To create a submenu and attach it to a menu item:

1. Create the menu using the **Menus** dialog box, as described in the section "Deleting a menu," earlier in this chapter.

2. Create the hierarchical menu item in another menu using the **Menu Items** dialog box, as previously described in the section "Adding and removing menu items," earlier in this chapter.

3. In the **Menu Items** dialog box, select the menu item you just created and set the Has Submenu option on.

4. Choose the submenu you created in Step 1 from the **Submenu** pop-up menu.

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

6. Click OK to close the **Menus** or **Menu Bar** dialog box.

Attaching Commands

After creating the user interface elements, you can assign functionality by attaching commands to them.

The role of commands

For a user interface element to be functional, it must generate an action within the application. In the THINK Class Library, this is accomplished by attaching a command to the user interface item.

Visual Architect automatically generates the necessary code for establishing such attachments.

In Visual Architect, commands are attached to panes and menu items. All panes derived from CButton (Push Button, Radio Button and Check Box), CSwissArmyButton (Picture Button, Straight Line, Unconstrained Line, Rectangle, Rounded Rectangle, Oval and Polygon), CIconPane (Icon Button) and CTable (List/Table) can have an associated command. Any menu item, whether accessed from the menu bar or from a pop-up menu, can have an associated command.

A command can have one of three actions in each class that responds to it. It can call a function that you code yourself, open an already defined view, or do nothing. If the action of the command is to call a function, Visual Architect generates skeleton code into the appropriate classes, into which you insert your code to handle the command.

Note

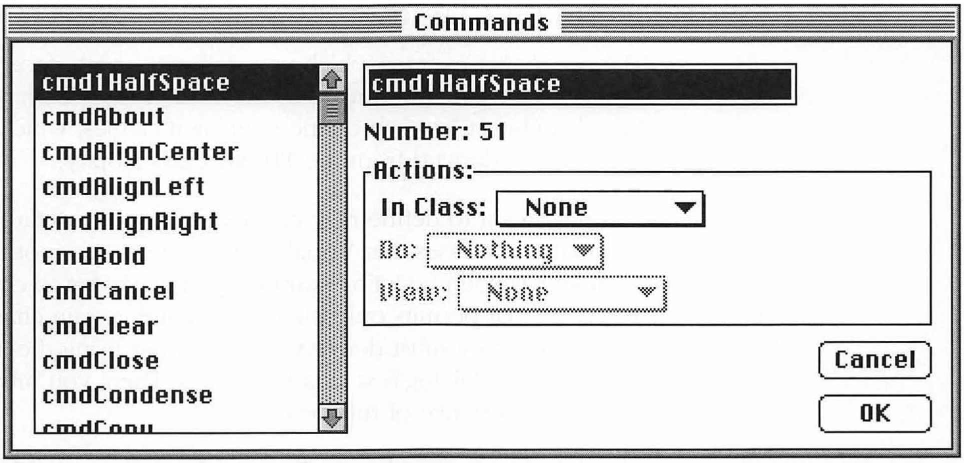
The specific mechanisms by which commands are sent and handled are dictated by the rules of the chain of command (see the section "Chain of command," in Chapter 27, "TCL and VA: Basic Concepts").

Defining a new command

To define a new command:

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

The **Commands** dialog box opens (Figure 7-14).



Commands

cmd1HalfSpace
cmdAbout
cmdAlignCenter
cmdAlignLeft
cmdAlignRight
cmdBold
cmdCancel
cmdClear
cmdClose
cmdCondense
cmdCopy

cmd1HalfSpace

Number: 51

Actions:

In Class: None

Do: Nothing

View: None

Figure 7-14 Commands dialog box

2. Choose **New Command** from the **Edit** menu (Command-K), or press Return.

   The textbox at the top of the dialog box is cleared and displays a blinking cursor.

3. Type the name of the new command into the textbox. The command name must be unique for the project.

   The new command is introduced in the command list on the left side of the dialog box.

4. Choose the classes you want to have respond to the command in the **In Class** pop-up menu.

   Multiple classes can respond to the command. Thus, choosing a class from the pop-up menu does not deselect any currently selected class. To deselect a class, choose it a second time.

5. Choose the action of the command in the **Do** pop-up menu.

6. If the action of the command is to open a view, choose the view to open in the **View** pop-up menu.

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

Defining Classes

The set of classes from which you define user interface elements in Visual Architect is not limited to the THINK Class Library classes. Visual Architect lets you define your own classes, which are directly or indirectly derived from the THINK Class Library.

You may want to define new classes because the default THINK Class Library classes that Visual Architect uses may not be as complete as you need. For example, you may want to create a dialog text pane that permits only the user to enter certain characters. In such a case, you must derive your own class using the THINK Class Library class CDialogText as a base class. Then, you must create a pane as an instance of this new class.

Visual Architect provides mechanisms for deriving and implementing your own classes. It also lets you define some of your class's data members. Once Visual Architect generates the skeleton code for implementing a new class, you must write the code to support the class's member functions and any additional data members.

Creating a new class

To create a new class:

1. Choose **Classes** from the **Edit** menu to open the **Classes** dialog box (Figure 7-15).

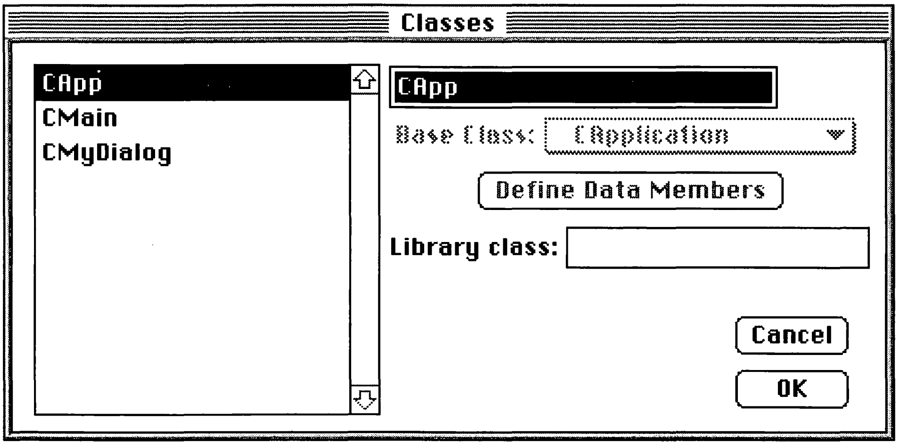


Figure 7-15 Classes dialog box

2. Choose **New Class** from the **Edit** menu (Command-K), or press Return.

   The textbox at the top of the dialog box clears and displays a blinking cursor.

3. Type the name of the new class in the textbox. The class name must be unique for the project.

   The new class is introduced in the class list on the left side of the dialog box.

4. Choose the THINK Class Library class from which to derive your class from the **Base Class** pop-up menu.

Note

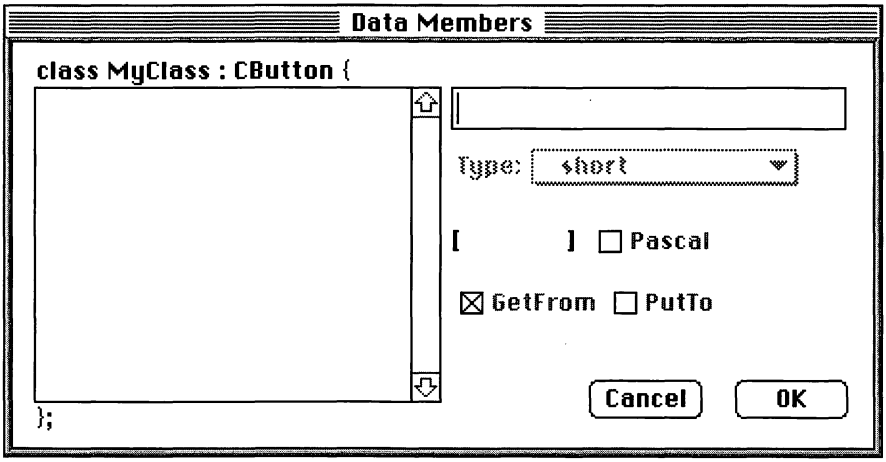
It is also possible to derive classes from your own library classes, rather than directly from THINK Class Library classes. Refer to “Library class textbox,” in Chapter 30, “Visual Architect Edit Menu,” for details.

Defining data members

To define data members for a class:

1. In the **Classes** dialog box, select the class and click the Define Data Members button.

The **Data Members** dialog box opens (Figure 7-16).



Data Members

class MyClass : CButton {

};

Type: short

[ ] □ Pascal

☒ GetFrom □ PutTo

Figure 7-16 Data Members dialog box

2. Choose **New Data Member** (Command-K) from the **Edit** menu, or press Return.

   The textbox at the top of the dialog box clears and displays a blinking cursor.

3. Type the name of the new data member in the textbox. The data member name must be unique for the class.

   The new data member is introduced in the data member list on the left side of the dialog box.

4. Choose the data type for the data member from the **Type** pop-up menu.

5. Click OK.

Changing classes

To change the class for which the pane is an instance:

1. Open the View Edit window for the view and select the pane.

2. Choose the class from the **Class** submenu of the **Pane** menu.

Adding Balloon Help

Once your views, panes, and commands have been laid out, you should add Balloon Help for your user interface elements.

Visual Architect lets you create Balloon Help for your views and panes. It supports four different balloon types, corresponding to the different states your views and panes can assume at run-time.

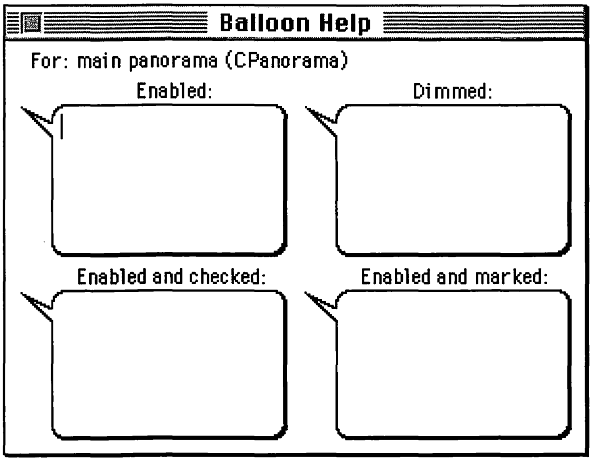
If you want to add Balloon Help to other user interface elements, such as menus, use a resource editor such as ResEdit or Resorcerer.

To add Balloon Help to a view:

1. Open the View Edit window for the view.

2. Click in the window between the panes to confirm that no pane is selected.

3. Choose **Balloon Help** from the **Edit** menu to open the Balloon Help window (Figure 7-17).



Balloon Help

For: main panorama (CPanorama)

Figure 7-17 Balloon Help window

4. Click in the appropriate balloons and type the Balloon Help text.

5. Close the Balloon Help window.

Generating Source Code

After designing the user interface elements in Visual Architect, you must generate the source code for the classes that define these elements. Visual Architect accomplishes this by generating source code files from the resources it created during the design of the user interface. These source files are incorporated automatically into the Symantec Project Manager project.

Next, you must expand the code generated by Visual Architect. Because `Visual Architect.rsrc` is a project entry with a resource file extension, the Symantec Project Manager copies its resources into the application when the application is built.

Visual Architect and the Symantec Project Manager

Visual Architect is coupled with the Symantec Project Manager through Apple events. As a result, it can notify the Project Manager to add source files to a project, as well as to update and run the project. This system simplifies project maintenance by letting you take care of basic management tasks without switching out of Visual Architect.

Note

> The linkage between Visual Architect and the Symantec Project Manager is maintained only while the Project Manager is running and the project to which the `Visual Architect.rsrc` file belongs is open.

Source files created by Visual Architect

Visual Architect generates code that you can modify. More importantly, those modifications are not overwritten the next time Visual Architect generates code. For each class defined in a Visual Architect project, an upper-level class and a lower-level class are created. Two corresponding `.cp` and `.h` source files are generated. One `.cp/.h` pair contains the upper-level class and one `.cp/.h` pair contains the lower-level class.

The upper-level files are only generated once, while the lower-level files are rewritten each time Visual Architect generates source code for the class. All modifications must be made to the upper-level `.cp` and `.h` files. These upper-level classes override the lower-level classes.

Macro files

Visual Architect generates your project source files using a set of macro files, that is, text files written in the Visual Architect macro language. By default, the set of macro files used is the set provided with Symantec C++ for Power Macintosh. You also can customize the way in which Visual Architect generates code by supplying your own macro files. For details on macro files, see "Inside Macro Files," in Chapter 35, "VA: Symantec Project Manager Menu."

Generating source code and updating a project

To have Visual Architect generate source files for a project, choose **Generate** (Command-G) or **Generate All** from the **Symantec Project Manager** menu (Figure 7-18).

Note

> The **Symantec Project Manager** menu title is the Symantec Project Manager application icon.

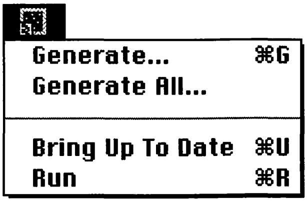


Figure 7-18 Symantec Project Manager menu

Note

> The first time you generate code from a `Visual Architect.rsrc` file, you must choose **Generate All**.

A dialog box opens, showing the process of the code generation and project updating. When the process is complete, the generated source files are placed in the Source group in the Symantec Project Manager project.

To update a Project Manager project from within Visual Architect, choose **Bring Up To Date** from the **Symantec Project Manager** menu (Command-U).

Customizing Visual Architect source files

Visual Architect does not generate all the code necessary to implement your application's user interface. You must perform hand coding to supplement the code it generates. Examples of the types of tasks you need to complete are listed below.

Enabling a menu item

By default, the code generated by Visual Architect disables all but a core set of menu items. You must enable any menu items you create at the appropriate places in your own code.

Handling commands

Code generated for a command is complete only when the command opens a view. Otherwise, you must go to the classes that handle the command and create a `case` statement to support its desired action. When the action of the command is Call, the `case` statement is generated for you, but it contains no code.

Initializing a view

When a view appears at run-time, some panes are activated, others are deactivated, and some have default values defined for them. You must write the code to set up the panes' data members in the view's upper-level class.

Debriefing a view

When a view is closed, you often need to determine the state of the panes and record information that the user entered. Once again, this process must be coded by you.

Advanced Topics 8

As you become proficient with the Symantec C++ for Power Macintosh, you begin to undertake larger programming tasks. This chapter looks at some of the advanced features of the Program Manager designed to help you track programming development in large projects.

The first section covers options sets, which you can use to establish and then reuse groups of options intended for different stages of development. The second section describes precompiled headers, with which you can speed the compilation time of your projects. AppleScript is introduced next; you can use this tool to automate repetitive tasks by recording and running scripts of Symantec Program Manager commands.

Communication with SourceServer and ToolServer is described in the final two sections. For large projects, especially when a team of programmers is involved, SourceServer can help keep track of project source code, and, with ToolServer, programmers can access tools written for Apple's Macintosh Programmer's Workshop (MPW).

Creating Options Sets

Options sets let you establish all the option page settings for your project with one selection. A project can have several options sets, and you select the one to apply to the project. At any time during product development, you can change to another options set to apply a set of options that better accommodate the needs of your project.

Project-specific options are defined in the eight pages of the **Project Options** dialog box. When you modify a project's options through this dialog box, you have the choice of saving those changes as an options set.

For example, you could define one or more options sets for development as well as one for the final release of the product. Each of these options sets would have different options set on and off, as appropriate to that particular stage in development. Then, depending on where you are in the development process, you can apply the appropriate options set easily.

By default, each project has one options set that has the same name as the project. You can create as many options sets for each project as you would like.

Defining a new options set

To define a new options set:

1. Choose **Options** from the Symantec Project Manager **Project** menu to open the **Project Options** dialog box.

2. On the eight pages of the dialog box, set the appropriate options on and off.

3. Choose **Save Options As** from the **Options** pop-up menu at the top of the dialog box (Figure 8-1).

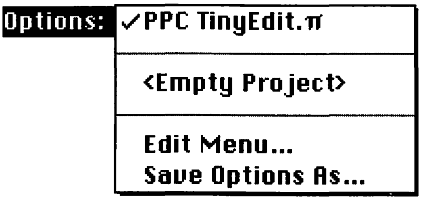
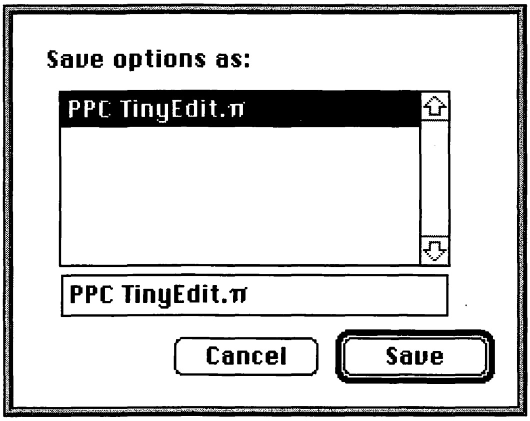


Figure 8-1 Options pop-up menu

The **Save Options As** dialog box opens (Figure 8-2). Note that the default options set, with the same name as the project, is always listed.



Save options as:

PPC TinyEdit.π

PPC TinyEdit.π

Figure 8-2 Save Options As dialog box

4. In the textbox, type a name for the options set you are defining.

5. Click Save.

   The **Project Options** dialog box closes.

To apply an options set, first make sure that toolbars are enabled in the Project window. Choose the options set you want from the **Options** pop-up menu at the top of the Project window.

Modifying options sets

To modify an existing options set:

1. Choose **Options** from the Symantec Project Manager **Project** menu to open the **Project Options** dialog box.

2. Select an options set to modify from the **Options** pop-up menu.

3. Set up the appropriate options.

4. Click Save.

Modifying the default options set for empty projects

You can modify the options set that is automatically applied to all new projects created with the Empty Project project model. You should consider doing so if you find yourself changing the same options in all new projects. To modify this default options set:

1. Choose **Options** from the **Project** menu to open the **Project Options** dialog box.
2. Choose **<Empty Project>** from the **Options** pop-up menu.
3. Set up the options as desired.
4. Click Save.

The options set you just saved is now applied to every project created with an Empty Project project model.

Using Precompiled Headers

Using precompiled headers can greatly speed compilation time, especially for large projects. Precompiled header files are compiled before a project is built or updated. These files are in a format that the compilers can use readily, and they load faster than text header files. Precompiled headers can be included in source files as standard text header files. Source files that are precompiled must contain only declarations and preprocessor symbols.

Included with Symantec C++ for Power Macintosh are several precompiled headers containing the most common declarations used for writing Macintosh programs. Headers are provided for both C++ and C—for example, `PPC MacHeaders++` (for C++) and `PPC MacHeaders` (for C). These are precompiled versions of `Mac #includes.cp` and `Mac #includes.c`, respectively.

Note

> If you are using the Debugger, you should use precompiled headers because they reduce the size of the Debugger tables.

Checking extensions and compiler options

The same translator extension rules apply to precompiling source files as to compiling them. Thus, any header file you want to precompile as a C++ source file must end in `.cp` or `.cpp`, and any header file you wish to precompile as a C source file must end in `.c`. If your header files end in `.h`, you must do one of the following:

- Rename them to have a `.cp`, `.cpp`, or `.c` extension.
- Create a corresponding `.cp`, `.cpp`, or `.c` file that includes the `.h` file(s), and then precompile.
- Change the extensions mapping on the Extensions Mapping page of the **Project Options** dialog box.

You cannot use one compiler's precompiled header in another compiler's source file. Check that the options in effect during the precompilation of a header file are compatible with the ones you expect to use when you are compiling a source file that includes the precompiled header. Also, project prefix statements are not included in your precompiled header (these are defined on the Prefix subpages of the PowerPC C and PowerPC C++ Options pages of the **Project Options** dialog box). Add them manually to the header file before it is precompiled.

Precompiling a header file

To precompile a header file:

1. Select the name of a header file in the Project window, or open a Symantec Project Manager Editor window for the file and bring it to the front.
2. Choose **Precompile As** from the **Build** menu to open the **File Save** dialog box.
3. Enter a name for the precompiled header file and indicate its file path.
4. Click Save to close the dialog box and precompile the header file.

The Symantec Project Manager automatically adds the text header file to your project so that it becomes part of the project's dependency table. The Project Manager can keep track of changes and automatically precompile header files the next time you build the application.

To precompile a header file for which a header has been already generated for a project:

1. Select the header file in the Project window.
2. Choose **Precompile** from the **Build** menu.

   The precompiled header file is generated and saved with the same filename.

To include a precompiled header in a source file, include it as you would any text header file: `#include` *filename.*

Note

Only one precompiled header can be included per source file.

Scripting the Project Manager

Using AppleScript, you can script such common tasks in the Project Manager as opening projects, adding files, bringing projects up-to-date, building targets, and making backups. You can record as a script and run almost any action that can be performed by the Project Manager. Even the Project Manager's Editor windows are scriptable; you can write scripts that examine or modify text within files.

Note

Symantec C++ supplies AppleScript as the default scripting system; however, you can use any other scripting system (such as Frontier) as long as it is compatible with the Open Scripting Architecture.

Recording scripts

You create scripts by recording your actions within the Project Manager. To do this, open the Script Editor and begin recording, switch to the Project Manager to perform a series of actions, and switch back to the Script Editor to turn off recording. The result is a script that repeats the series of actions you have just performed. The steps needed to record scripts are outlined in further detail in the rest of this section.

Opening the Script Editor

You record, test, and manually run your scripts in the Script Editor window, which you open from the Finder. Initially, an untitled Script Editor window opens.

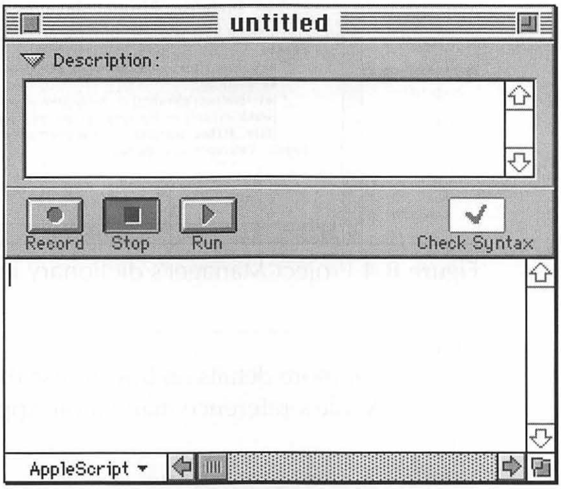


Figure 8-3 Script Editor window

The top pane in the window describes the script. You can save screen space by hiding this pane; click on the arrow to the left of the Description text field's title. The bottom pane displays the script as it is recorded.

Opening the Project Manager dictionary

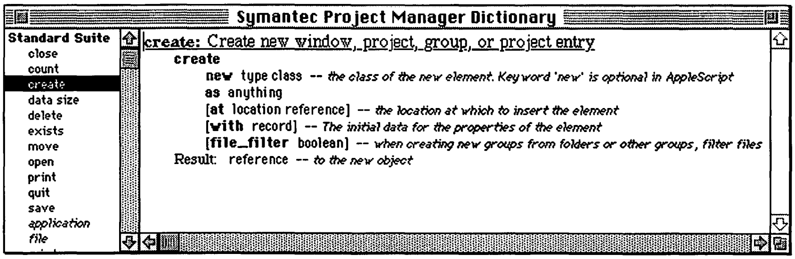
Your first task in the Script Editor is to open the Symantec Project Manager's dictionary. This dictionary includes a complete listing of the available AppleScript commands that the Symantec Project Manager understands, with brief comments explaining each command and parameter. You should refer to this resource when you have questions about the parameters to an AppleScript command.

To open the Symantec Project Manager's dictionary:

1. Choose **Open Dictionary** from the **File** menu in the Script Editor.

2. In the **File Open** dialog box, select the **Symantec Project Manager**.

The Symantec Project Manager dictionary opens. Selecting one of the dictionary entries displays a description of the entry and, if the entry is a command, its syntax and any required or optional parameters (see Figure 8-4).



```
create: Create new window, project, group, or project entry
    create
        new type class -- the class of the new element. Keyword 'new' is optional in AppleScript
        as anything
        [at location reference] -- the location at which to insert the element
        [with record] -- The initial data for the properties of the element
        [file_filter boolean] -- when creating new groups from folders or other groups, filter files
    Result: reference -- to the new object
```

Figure 8-4 Project Manager's dictionary in Script Editor

Note

For more details on how to use the Script Editor, see Apple's reference manual on AppleScript.

Recording a script

Before you begin recording, make sure that the Symantec Project Manager is open. You can record a script to perform almost any action in the Symantec Project Manager. Experiment by turning recording on and examining the AppleScript commands generated when you add files, switch options sets, and type in an Editor window.

To record a script:

1. Open the Script Editor from the Finder.

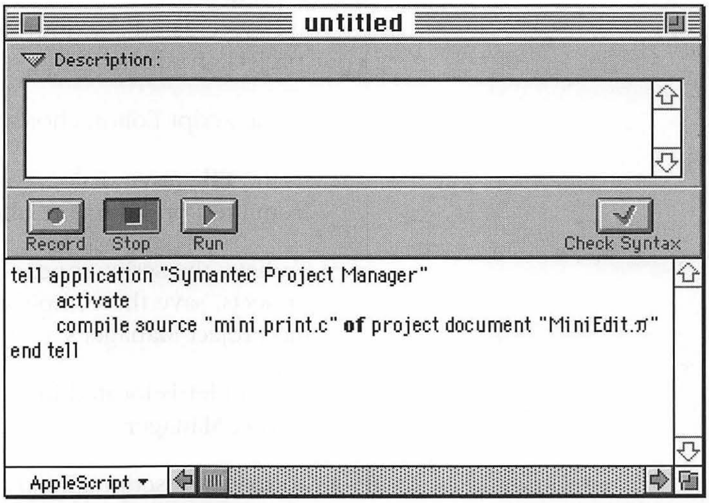
2. Start recording by clicking on the **Record** button.

3. Switch to the Project Manager.

4. Choose one or more actions in the Project Manager window, for example, "compile a file mini.print.c".

5. Switch back to the Script Editor.

   The Script Editor window displays the actions you have recorded (Figure 8-5). In this example, the action recorded was the compilation of a file called `mini.print.c`.



```
tell application "Symantec Project Manager"
    activate
    compile source "mini.print.c" of project document "MiniEdit.π"
end tell
```

Figure 8-5 Script Editor window with script recorded

6. Click the Stop button.

Note

Once a script is recorded and is being displayed in a Script Editor window, you can play it back by clicking Run.

Writing scripts

Once you have recorded a few scripts, you should try writing your own. Start by making changes to a script that you have already recorded. For example, you can try changing a parameter or removing a command. Then try copying pieces from several different scripts to create a script.

The next step would be to write scripts that mix commands to the Project Manager with commands to other applications such as the Finder. For example, you could create a script to make a compressed backup of your current project by combining commands to the Project Manager to prepare the project to be backed up, commands to a compression application to compress the contents of your project's folder into an archive, and finally commands to the Finder to copy the compressed archive to a backup device.

Storing scripts

After you have created a script, you can add it to the **Scripts** menu of your project. To do so:

1. In the Script Editor, choose **Save** from the **File** menu.

2. In the **File Save** dialog box, choose **Compiled Script** from the pop-up menu labeled **Kind** at the bottom.

3. To have the script appear in the **Scripts** menu for all projects, save the compiled script (or an alias for it) in the Project Manager's `(Scripts Menu)` folder.

   This folder is located in the same folder as Symantec Project Manager.

4. To have the script appear only in the **Scripts** menu of a single project, create a folder named `(Script Menu)` in the same folder as the project's Project file and then place the compiled script (or an alias) in it.

Running scripts automatically

The Project Manager can run scripts automatically. You can, for example, specify a script to run when the Project Manager first opens and when it closes. Another option would be having a script that runs when a particular project becomes active or inactive. Further details on both of these options are provided in this section.

Running a script at startup or shutdown

To run scripts automatically when the Project Manager opens or closes:

1. Record a script as described in the section "Recording a script" earlier in this chapter.

2. Select **File Save** from the **File** menu in the Script Editor window.

3. To have the script run when the Project Manager opens, name the script `Startup`.

4. To have the script run when the Project Manager closes, name the script `Shutdown`.

5. Place the scripts file in the system `(Scripts)` folder.

Running a script when a project is opened or closed

Scripts that run automatically when a project opens or closes must be located in a project `(Scripts)` folder. You create this folder and then place it in the folder with the Project file. You might create such a script, for example, to set one or more of the Project Manager's preferences to a particular setting for a specific project. You could run an "Activate" script to set the preference to that setting before a project is started and a "Deactivate" script to reset the preference after you are finished working with the project.

To run this type of script:

1. Create a folder named `(Scripts)` if one does not already exist in the same folder as the Project file.

2. Record a script and select **Save** from the **File** menu.

3. To run the script when the project opens or becomes active, name the file `Activate` and save the compiled script in the `(Scripts)` folder.

4. To run the script when the project closes or becomes inactive, save the compiled script as `Deactivate` and place it in the `(Scripts)` folder.

Note

When you quit the Project Manager, a "Shutdown" script for the Project Manager is run before a "Deactivate" script for the active project.

SourceServer

As projects grow to involve multiple programmers and numerous files, project management of files becomes an important and stabilizing component of product development. To help with the complexity of tracking files and their versions, Symantec C++ for Power Macintosh provides an interface to Apple's SourceServer.

This section is an introduction to SourceServer, and covers such topics as setting up SourceServer databases, storing files, checking files in and out of a database, and retrieving information about the current revision of a file.

SourceServer is a source code control system that stores, tracks differences between, and allows access to versions of project files. It can track any kind of file, including the Project file. It tracks source files and Rez files in a space-efficient manner, but tracks nontext files less efficiently.

Key terms

In order to understand SourceServer, it is important to know the following terms:

- Database
- Checking in
- Checking out
- Revision
- Revision tree
- Branch

Database
A SourceServer file is called a database. A database file contains information about one or more of the project's files.

Checking in
The process of putting files in a SourceServer database is called checking in. The initial check in adds the file to the database via the Worksheet window. Thereafter, check ins are performed through a command on the **Revision** menu.

Checking out
The process of getting a local copy (a file stored on your machine) of a specific version of a file is called checking out.

Revision
A version of a file stored in a database is called a revision.

Revision tree
The database contains a log of all changes made to each file, and the relationship of each change to the next. This log is called a revision tree.

Branch
Each version of a file can have any number of revisions stemming from it. Each one of these revisions can also have one or more parallel levels of revisions stemming from it. This parallel sequence of revisions is called a branch. Revision branches let you easily recover from a revision sequence that does not work, thus allowing you to return to a prior revision of a file. They also let more than one programmer work simultaneously from the same revision of the file, perhaps on different parts of the code, and later have their revisions converge at a subsequent revision.

Figure 8-6 shows an example of a SourceServer database that contains three revision trees with branches. The sequentially numbered circles along a vertical path through this database structure represent the revision numbers. For example, revision 2a1 has one revision labeled 2a2. The numbered circles along a diagonal path represent branch revisions. For example, revision 2a2 has one branch labeled 2a2a1.

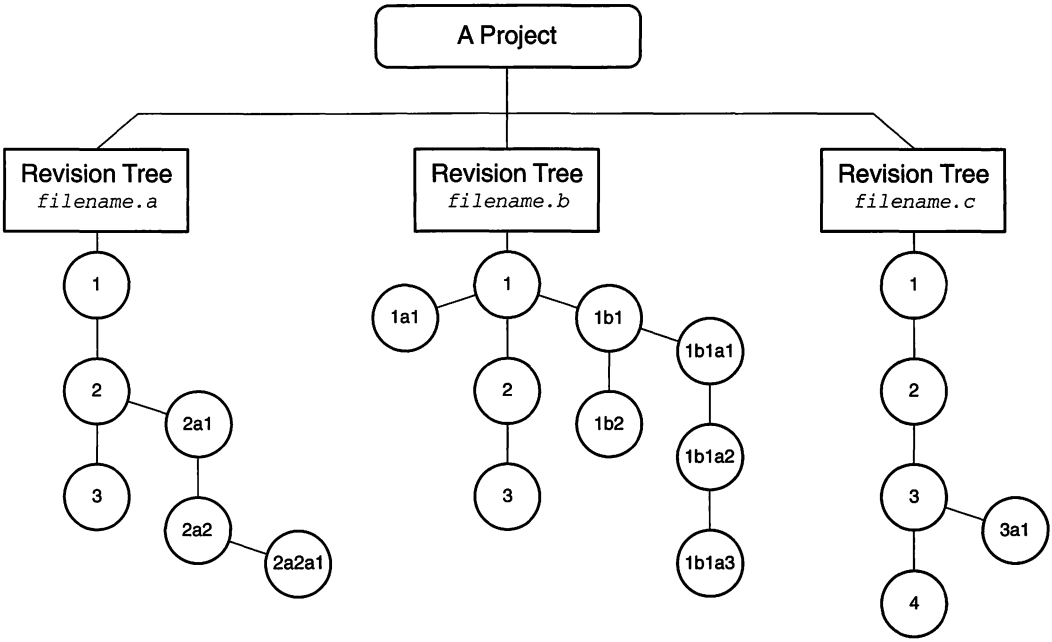


Figure 8-6 SourceServer project database

Note

SourceServer uses the Owner Name field in the Sharing Setup control panel to track who checks what files into and out of the database. SourceServer users must enter a name in the Owner Name field and have the Sharing Setup control panel in their `Control Panels` folders.

Setting up a SourceServer database

To keep track of revisions, the first step is setting up a SourceServer database and the second is mounting the database. Mounting indicates to the Project Manager that files may be checked in and out of the database. This step can be performed with the **Mount Database** command from the **Revision** menu. You can also have the Symantec Project Manager automatically mount databases when the Project file is opened.

Databases can also be nested. This allows the hierarchy of SourceServer databases to match the hierarchy of the folders comprising your project. Nesting also speeds up the rate at which SourceServer commands are performed.

Note

> You can have multiple root-level databases mounted at one time, but there is only one "current database" active at a time.

Creating a database

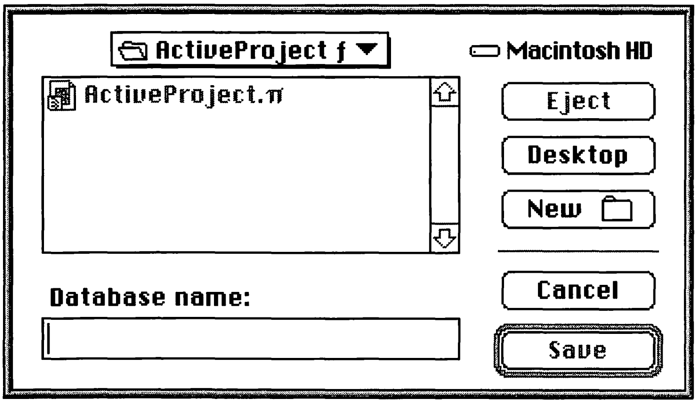
When you create a database, a folder with the name of the database is created and a database file named `ProjectorDB` is added to the database folder. The `ProjectorDB` file is the file in which all SourceServer information is stored.

To create a database in a project:

1. Open the project with which you want to use SourceServer.

2. Choose **New Database** from the **Revision** menu.

A standard **File Save** dialog box opens (Figure 8-7).



Database name:

Figure 8-7 New Database dialog box

3. Name the new database.

4. Click Save or press Return.

   A database folder with the name you have chosen is created and a file called `ProjectorDB` is placed within that folder. The new database becomes the current database (in other words, it is mounted automatically when it is created).

Automating database mounting
To set up a database for automatic mounting:

1. Create an alias for the top-level Projector DB file named `ProjectorDB` and move the alias into the project folder. (The alias must be named "Project name.pdb" to automatically mount the database. For example, if your project is named `MyProject.π` the alias would be `MyProject.π.pdb`.)

2. Double-click the Project file.

   This launches the Symantec Project Manager (unless it is already open), opens the Project file, and mounts the database.

Note

To prevent automounting, hold down the Shift key when opening the project. To mount a database manually once the Project Manager is running, use the **Mount Database** command in the **Revision** menu to select the appropriate database.

Nesting databases

If your SourceServer database is large (containing 50 or more files), the commands in the **Revision** menu may execute slowly. On a server, even a 20-file database can start bogging down. A better solution is to divide the database into nested databases. A nested database is a database located within another database's folder.

To nest a new database, you create it using the preceding procedure and place it in the folder containing the appropriate top-level database (Figure 8-8).

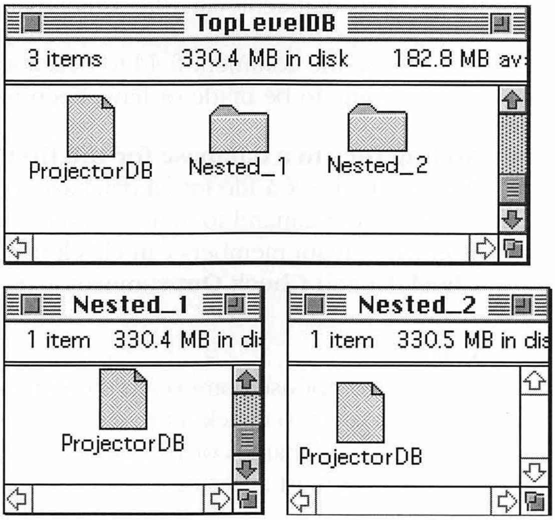


Figure 8-8 Nested databases

Whenever you mount the top-level database, SourceServer automatically mounts all of its nested databases.

Checking files in and out

Once a database is mounted, you can check files into and out of it. You store files in a database when the file is first created or first added to a project. After that, you check the file out to make changes and check it back in when you are done making the changes.

To add new files to the database, you check them in through the Worksheet window.

When someone checks files in or out, SourceServer records who handled the file and when. SourceServer also provides two text fields to aid in tracking versions: the task field and the comment field. Although you can use these fields as you like, you should consider using the following guidelines:

- Use the task field to keep track of the overall task or to describe the goal of a set of modifications. All files checked out for the same task, in other words, can be assumed to have modifications made to them that are aimed at achieving the same goal.
- Use the comment field to describe the changes that are going to be made or have been made in the file.

Adding files to a database for the first time

When you check a file into a database for the first time, you must first send a command to SourceServer from the Worksheet window. Thereafter, team members can check the file in and out via the **Check In** and **Check Out** commands on the **Revision** menu.

Note

Do not use commas in the names of files that you expect to check into a SourceServer database. For any filenames or pathnames that contain spaces, you must add single quotes.

To add files to a SourceServer database:

1. In the Project window, open the file by double-clicking on it or by pressing Return.

Note

> You can only use the **Check In** command with files that have already been added via the Worksheet window.

---

2. Command-click on the filename in the title bar of the Editor window to open the **File Path** pop-up menu (Figure 8-9).

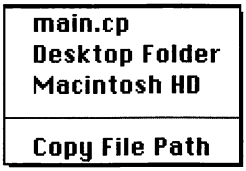


Figure 8-9 File Path pop-up menu

3. Choose Copy File Path.

4. Close the file.

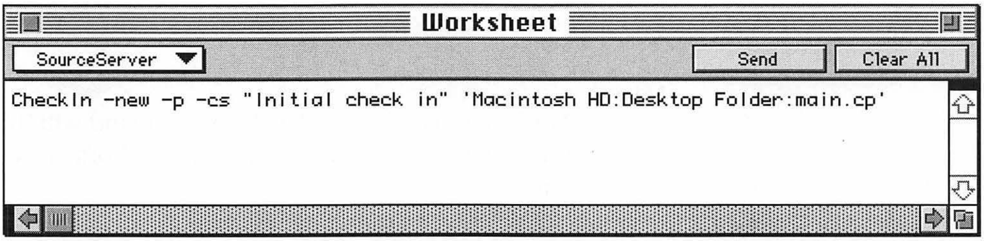
5. Choose **Worksheet** from the **Windows** menu to open the Worksheet window. The pop-up menu at the top left of the window lets you switch back and forth between SourceServer and ToolServer.

6. Type `CheckIn -p -new -cs "Initial check in"`

   Optional parameters follow the command `CheckIn`. The `-new` option indicates that the file is being added for the first time to the database. The `-p` option tells SourceServer to report progress information on the execution of the command. The `-cs` option lets you type in the comment field.

7. Place the cursor at the end of the comment string and paste (Command-V) the file path copied in Step 6.

The file path is pasted into the Worksheet window (Figure 8-10).



```
CheckIn -new -p -cs "Initial check in" 'Macintosh HD:Desktop Folder:main.cp'
```

Figure 8-10 Worksheet window ready for checking in a new file

8. Leaving the cursor at the end of the line, click on the Send button or press Enter.

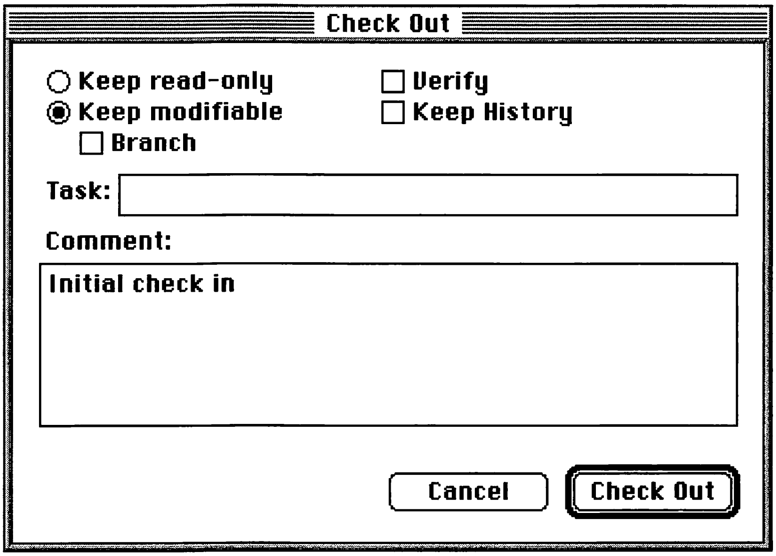
   The cursor drops to the next line and the file is checked into the database.

Checking files out for modification
Team members typically check files out for modification, though sometimes they need the files only for reference.

To check a file out of the database:

1. Make sure the database is mounted.
2. In the Project window, select the file (or files) you want to check out.

3. Choose **Check Out** from the **Revision** menu to open the **Check Out** dialog box.



○ Keep read-only
● Keep modifiable
□ Branch
□ Verify
□ Keep History

Task:

Comment:

Initial check in

Figure 8-11 Check Out dialog box

4. To make changes, click Keep Modifiable. To only read the file, set the option Keep Read-Only on.

5. To verify that the file is undamaged before checking the file out, set the Verify option on.

6. To store the author, check in date, task, and comments with the file, set the Keep History option on.

7. Set Branch on to modify a file that is currently checked out for modification by another team member or to split off a version of the file for a parallel line of development.

8. Enter the task and comments in the respective fields.

9. Click **Check Out.**

The file is now checked out. SourceServer notes the date and the name of the user as well as the name of the task and any comments.

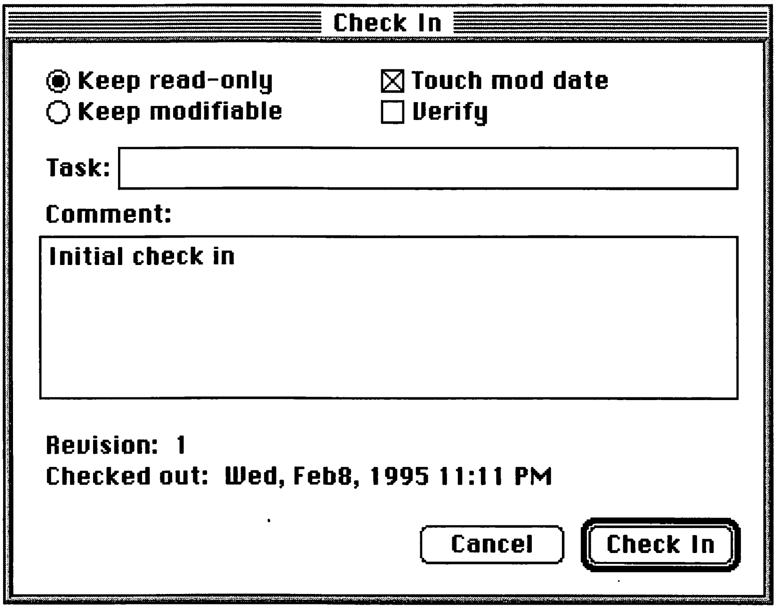
Checking files in

To check a file back in to a database:

1. Make sure the database is mounted.

2. Select the file from the Project window.

3. Choose the **Check In** command from the **Revision** menu to open the **Check In** dialog box (Figure 8-12).



● Keep read-only
○ Keep modifiable
☒ Touch mod date
□ Verify

Task:

Comment:

Initial check in

Revision: 1
Checked out: Wed, Feb8, 1995 11:11 PM

Figure 8-12 Check In dialog box

4. Set the Keep Read-Only option on to make your local copy a read-only version of the file. To keep a modifiable version, click Keep Modifiable instead. This latter option checks the file in and then checks a new modifiable revision back out.

5. To set the modification date of the file to the time it was checked in (as opposed to the time it was last modified), set the Touch Mod Date option on.

6. To verify that the file's contents are not damaged, set the Verify option on.

7. Modify the Task and Comment fields as appropriate for the modifications you made.

8. Click **Check In**.

   The file is checked back into the database, and SourceServer notes the time and the name of the team member as well as any comments added.

Accessing revision information

You can find out information about any version of a file that has been checked into a SourceServer database using the **Get Revision Info** command from the **Revision** menu. Revision information includes the revision number that the file represents, the date the file was checked out, the team member involved, the project, the task, and any comments.

Whenever you open a source file, you can see whether it has been checked into a database and whether it is a read-only or modifiable file. The Editor window's toolbar displays an icon (a pencil) if the file is part of a SourceServer database. For a modifiable file, you see only the pencil at the right edge on the toolbar. For read-only files, the pencil has a line through it.

To retrieve information about a file:

1. Open the file by double-clicking on the filename in the Project window.

2. If the toolbar is not displayed, select **Show Toolbar** from the **Windows** menu.

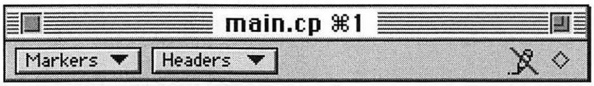


Figure 8-13 Toolbar displaying read-only file icon

Notice that the toolbar displays a pencil with a line through it. The pencil can be in one of three states:

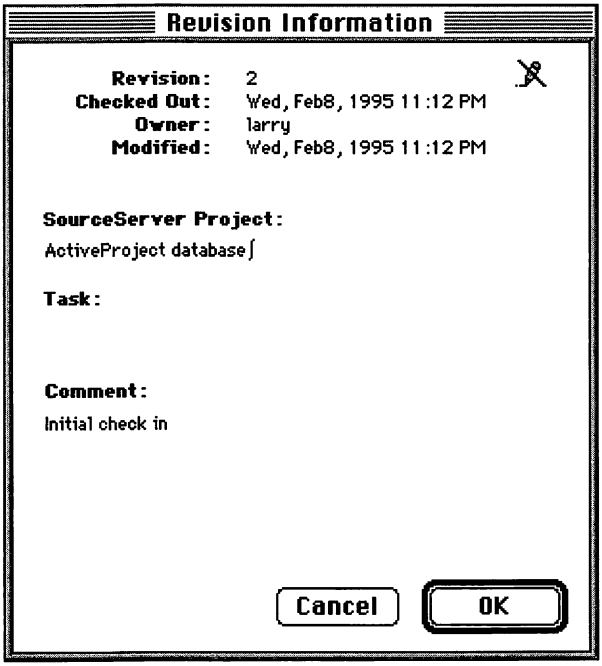
- Solid line means the source file is read-only.

- Dashed line means the source file is modifiable, but you cannot check the changed file back in.

- No line means the source file is modifiable.

  When you check a file in, SourceServer sets your local copy to read-only so that you do not make any changes without checking the file out again.

3. Choose **Get Revision Info** from the **Revision** menu to open the **Revision Info** dialog box (Figure 8-14).



Revision Information

Revision: 2
Checked Out: Wed, Feb8, 1995 11:12 PM
Owner: larry
Modified: Wed, Feb8, 1995 11:12 PM

SourceServer Project:
ActiveProject database∫

Task:

Comment:
Initial check in

Figure 8-14 Revision Info dialog box

This dialog box displays version information about the file.

Note

The database to which the file belongs does not have to be mounted before you choose the **Get Revision Info** command. For more details about using the Worksheet window and the **Revision** menu with SourceServer, see the electronic supplemental information.

Using ToolServer

Using ToolServer, you can access tools created for use in Apple's Macintosh Programmer's Workshop. You communicate with ToolServer using the Worksheet window, which you open by choosing **Worksheet** from the **Windows** menu. This section covers how to set up ToolServer for use with the Symantec Project Manager and how to use ToolServer and MPW tools from the Worksheet window.

Setting up ToolServer

To take advantage of ToolServer, you must first add an alias for ToolServer to the folder in which Symantec Project Manager is located. You must also add an alias for the MPW tools folder to the `ToolServer` folder.

To set up ToolServer:

1. Select ToolServer in the `Apple Tools` folder and make an alias for it.
2. Put the ToolServer alias in the `(Tools)` folder. The `(Tools)` folder is in the `Symantec C++ for Power Mac` folder. The alias must be called ToolServer.

You are now ready to use ToolServer and the MPW tools in conjunction with Symantec C++.

Using MPW tools with ToolServer

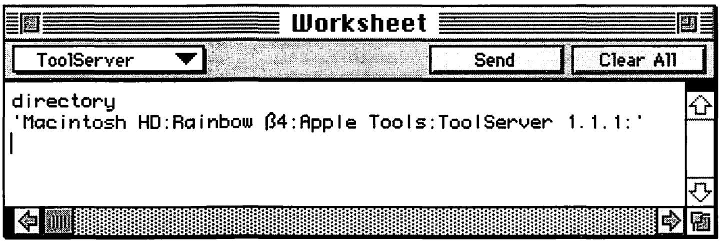
Once you have set up ToolServer, you can make use of MPW tools by opening the Worksheet window and sending commands to ToolServer. The pop-up menu at the top left of the Worksheet window lets you switch back and forth between ToolServer and SourceServer.

Note

If the ToolServer option is disabled, the Project Manager has been unable to locate ToolServer. Close the Worksheet window and from the Finder open the `(Tools)` folder located in the same folder as Symantec Project Manager. The ToolServer application or an alias to it should be located in that folder. See the previous section, "Setting up ToolServer."

To use an MPW tool:

1. Open Worksheet window from the **Windows** menu.
2. If necessary, choose **ToolServer** from the pop-up menu at the top left.
3. To check the current directory, type `directory` and press Enter, Command-Return, or click Send Command.



```
directory
'Macintosh HD:Rainbow β4:Apple Tools:ToolServer 1.1.1:'
```

Figure 8-15 Results of Directory command

4. To change the current directory, type `directory` '<full pathname>' and press Return.

Note

If you do not specify a directory, ToolServer saves results files in the current directory.

5. Type the command for the MPW tool you want to use, type the arguments, and press Enter, Command-Return, or click Send Command. For example, to use the Compare tool, type `compare <file1> <file2>` and press Return.

You may also save the results of the comparison in a file. Append "> <Filename>" at the end of the command line. ToolServer creates a file in the current directory with the filename you supply unless you include a pathname in the file's name. For example, to compare the contents of `file1.cp` and `file2.cp` (both in the current directory) and save the results in a file named `Files.diff`, also in the current directory, type the following:

```
Compare file1.cp file2.cp > Files.diff
```

Symantec C++

Learning by Example (Tutorials)

Part Three

Tutorial Introduction 9

The six tutorial chapters in this part of the manual will help you become familiar with the main features of the Symantec C++ for Power Macintosh development environment. Each tutorial focuses on a couple of key aspects of Symantec C++ for Power Macintosh that you need for writing, compiling, and debugging applications for the Power Macintosh.

What You Will Learn

By performing all the tutorials, you will learn how to:

- Create a new project
- Add source files and resource files to your project
- Correct errors and debug your program
- Use the THINK Class Library and Visual Architect as a basis for full-featured Power Macintosh applications

This collection of six tutorials also demonstrates how Symantec C++ supports different user interfaces and programming styles within one development environment. You can, for example, choose to write your program in C or in C++. You can use a terminal window for I/O or opt for a full-fledged Macintosh user interface. You also have the option to write your program using the framework provided by the THINK Class Library or to write your program entirely with your own C++ code.

Depending on your needs, you may not need to work through every tutorial. If you are only interested in writing programs that use a simple terminal window interface, then complete the tutorial "Hello World," "MiniEdit," and "Object Bullseye." If you are planning to use the THINK Class Library and Visual Architect, which is strongly recommended for larger programs, you should go through all six tutorial chapters and perform all the procedures in the sequence provided.

Hello World

"Hello World" is the traditional example program for all C and C++ programmers. When you have completed this tutorial, you will have a program that displays the words "Hello, World!" in a window on your Power Macintosh. Source code is provided for both C and C++ versions of "Hello World." First, you will create the C version of "Hello World," then the C++ version.

Writing these simple programs will introduce you to the Symantec Project Manager, the main component of Symantec C++ for Power Macintosh. In this tutorial you will create and edit projects, work with source code, and compile and link your project.

The C version of "Hello World" uses the Standard ANSI library; the C++ version uses IOStreams. Both versions simulate a console that displays the message, "Hello, World!" Using these libraries, a programmer can create applications with simple user interfaces that can be ported easily to other systems, such as DOS or Unix.

MiniEdit

"MiniEdit" is a simple Macintosh text editor. It lets you open, read and edit, and save text files to disk. It is a smaller, simpler version of SimpleText, the word processor that is now distributed by Apple with their new system software.

As "MiniEdit" contains much more code than "Hello World," source files are included as a convenience so that you need not type in the entire program. "MiniEdit" introduces the use of Macintosh resource files to your programming. In addition, a syntax error has been inserted in one of the source files in case you want to see how Symantec C++ for Power Macintosh helps you detect and fix syntax errors.

"MiniEdit" is a full-fledged Macintosh application that goes beyond the limitations of the "Hello World" tutorial. It responds to the many kinds of events that most Macintosh programs respond to, including mouse clicks, keystrokes, window update events, and other events. It also has menus, windows, buttons, and additional controls.

Object Bullseye

"Object Bullseye" displays windows that contain circular, square, or triangular bullseyes. By choosing different menu items, you can alter the thickness of the bullseye's concentric shapes.

This tutorial takes you through an introductory tour of the Symantec Debugger. Using the Debugger, you can step through the execution of your program one line at a time. You can examine and modify the values of your variables. You can also trace how and when your program's functions interact and call each other.

"Object Bullseye" is the first tutorial in this series that uses C++ classes, but it purposely uses only a few classes in a simple, straightforward way. The circular, square, and triangular bullseye windows are examples of sibling classes that inherit their general behavior from an abstract bullseye window class.

Vector

"Vector" demonstrates some of the more advanced aspects of both C++ and the Symantec Debugger. The Vector application displays sorted and unsorted lists of numbers, letters, and dates as well as the maximum value of each list in a console window.

"Vector" uses the more complex aspects of C++ programming, including C++ inline functions, templates, and operator overloading. The tutorial shows you how to use the Debugger to debug C++ programs and how to instantiate templates.

Beeper

"Beeper" is a small program that is designed to introduce Visual Architect and THINK Class Library (TCL). TCL contains classes for implementing the Macintosh user interface. TCL is a complete source code framework for developing standard Power Macintosh applications. It is written as a set of C++ classes that can easily be extended and customized for your particular needs.

Visual Architect is a tool for designing graphically your application's user interface. Visual Architect uses the TCL. "Beeper" brings you through some basic steps for generating and building an application. It presents a window with some graphics and a button. The button opens a dialog box in which you enter a number. When you click the Beep button, you computer beeps that number of times.

Process Monitor

Like "Beeper," "Process Monitor" is built using Visual Architect. In this final tutorial, you will explore more of the Visual Architect and will see how a reasonably large program is organized.

"Process Monitor" displays a list of currently running processes (programs). The application also displays three push buttons that let the user enter the Debugger, kill a selected process, and bring a selected process to the foreground. It is a full-fledged Macintosh application that contains multiple windows, panes and subviews, controls, and menus.

When you have completed the tutorials, you may go on to Part 2 for more information on writing a program using Symantec C++, or you may turn to the reference chapters in Parts 4 and 5 for detailed information about specific features.

Tutorial: Hello World ◆ 10

This chapter is a tutorial on building simple applications with Symantec C++. The Symantec C++ Project Manager can be used to create C and C++ Macintosh applications. You will create both types in the course of finishing this tutorial.

Before You Begin

Make sure Symantec C++ for Power Macintosh is installed correctly on your hard drive. Refer to the section "Installing Symantec C++," in Chapter 1, "Overview," for instructions.

Before starting this tutorial, you should be familiar with the basics of working with the Macintosh user interface, such as opening menus and dialog boxes, as well as navigating between folders.

Hello World C Application

Both applications you create in this tutorial write "Hello, World!" to the screen. The steps in creating the C version of the Hello World application include:

- Create a project
- Create and add a source file to the project
- Compile the source file
- Add libraries to the project
- Build and run the application
- Save the application to disk

The following sections explain these steps in detail.

Creating a Project

In Symantec C++, a project is the cornerstone of application development. The project keeps track of all your source files, maintains the dependency information for a project, and contains the object files. The first step in creating an application (or a library) is to create a project. To do so:

Symantec Project Manager

1. Launch the Symantec Project Manager by double-clicking its icon in the Symantec Project Manager folder (by default, this is the `Symantec C++ for Power Mac` folder).

   The **Open Project** dialog box opens (Figure 10-1).

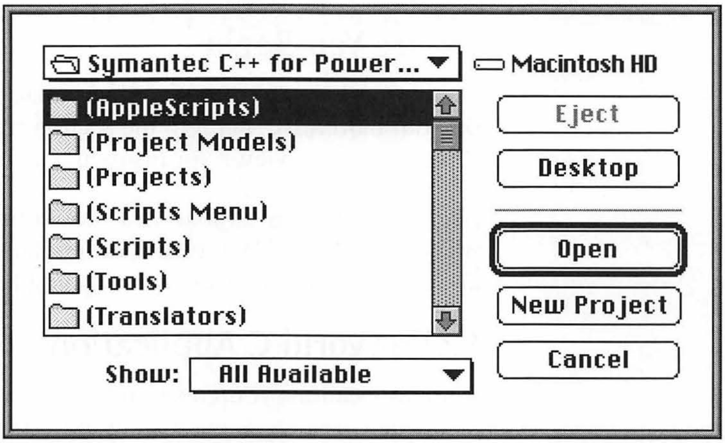
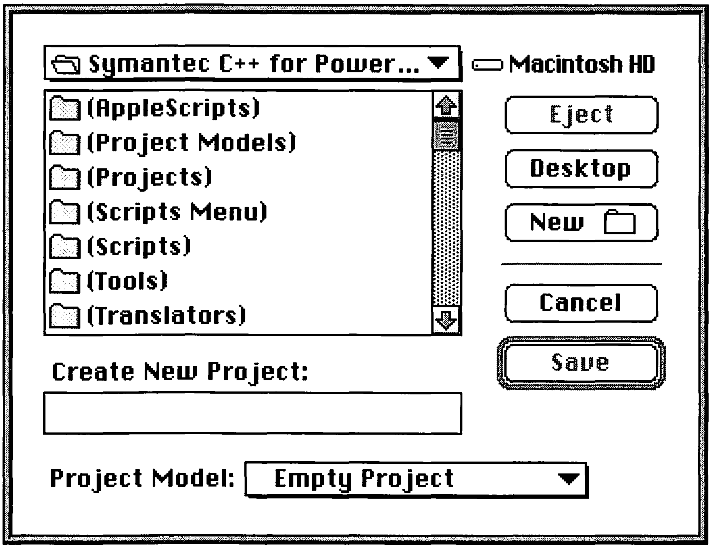


Figure 10-1 Open Project dialog box

2. Click the New Project button.

The **New Project** dialog box opens (Figure 10-2).

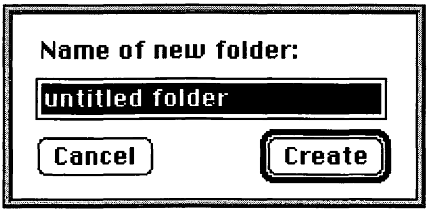


Create New Project:

Project Model: Empty Project

Figure 10-2 New Project dialog box

3. Navigate outside the Symantec Project Manager folder, then click the New (folder) button to open the **New Folder** dialog box (Figure 10-3).



Name of new folder: untitled folder

Figure 10-3 New Folder dialog box

Note

> Do not store projects in the system tree (the folder in which Symantec Project Manager and its subfolders reside). The `(Projects)` folder in the system tree is used only to store aliases to frequently used projects.

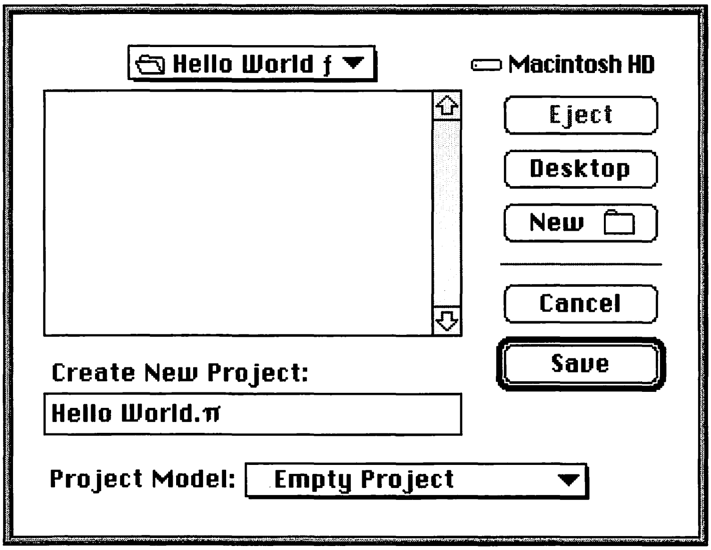
4. In the textbox, type `Hello World ƒ` and press Return. (Press Option-F to create the ƒ symbol.)

   The folder `Hello World ƒ` is created for the project, and you return to the **New Project** dialog box.

5. In the Create New Project textbox, type `Hello World.π` as the project's name. (Press Option-P to create the π symbol.)

6. Check that Empty Project is chosen from the **Project Model** pop-up menu (Figure 10-4).

   You use this model to create empty projects that you will then build from scratch.



Create New Project: Hello World.π

Project Model: Empty Project

Figure 10-4 New Project dialog box ready to create the project

7. Click Save to create the new project `Hello World.π` and close the **New Project** dialog box.

Your new project is now created. The Symantec Project Manager opens the project automatically after creating it, so you should see a Project window, as shown in Figure 10-5.

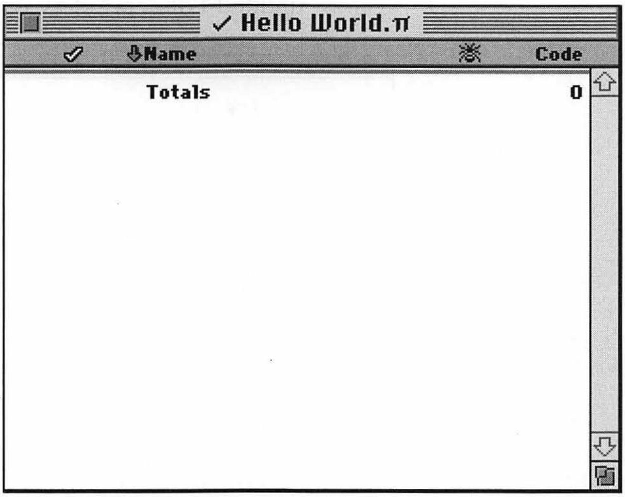


Figure 10-5 Project window for the Hello World project

The Project window lists the names of all files included in a project. Because you created an empty project, no filenames are now displayed in this window.

Creating a source file

You are ready to create a source file and save it using the Symantec Editor. This text editor works like most other text editors on the Macintosh. You can double-click to select words, triple-click to select an entire line, and drag to select a range of text. You can also use the arrow keys to move around a file.

The text editor has an auto-indent feature. It automatically indents and unindents after curly braces. It also does not wrap text when you type past the right edge of the window. Use the horizontal scroll bar at the bottom of the window to view any text that extends beyond the right edge. For more information about the Symantec Editor, see Chapter 4, "Editing a Project's Code."

To create a new source file:

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

   A new, untitled Editor window opens (Figure 10-6).

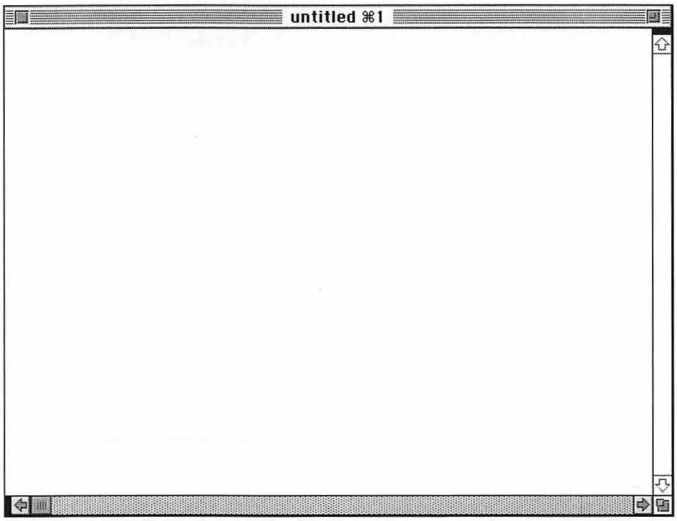


Figure 10-6 Empty Editor window

2. Type the following source code:

```
/*****
* Hello World.c
*
* The hello world C program for
* Symantec C++ for Power Macintosh
*
*****/
#include <stdio.h>
main()
{
   printf("Hello, World!\n");
}
```

3. Choose **Save As** from the **File** menu to save this new source file.

The **File Save** dialog box opens (Figure 10-7).

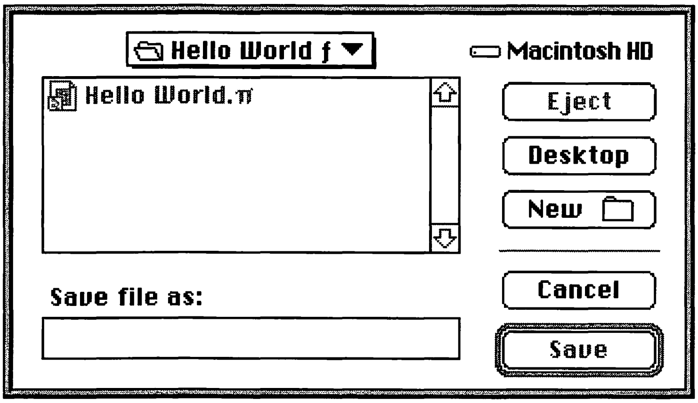


Figure 10-7 File Save dialog box

4. Type `hello.c` into the Save file as textbox and click Save.

Note

Be sure to save the file to the `Hello World ƒ` folder.

The dialog box closes, and the file is saved as `hello.c`. The title bar of the Editor window changes to reflect the new name.

Warning

Be sure to name your file `Hello.c`, not `Hello.cp`. The Symantec Project Manager uses files extensions to identify file types. By default, the C translator is used to compile .c files and the C++ translator to compile `.cp` and `.cpp` files.

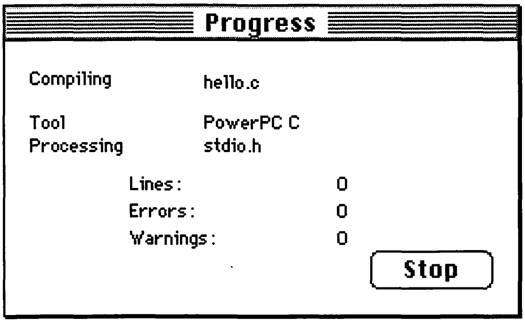
Now that your source file is saved, the next step is to compile it and add it to the project.

Compiling the source file and dealing with errors

*Using the **Compile** command in the Symantec Project Manager is similar to using the* `cc` *command in UNIX. The Symantec Project Manager, however, adds the object code to your project instead of creating a separate object file.*

If you have followed the previous steps, you have an Editor window titled `hello.c` open on your screen. To compile the source file displayed in the window:

1. Choose **Compile** from the **Build** menu to open the **Progress** dialog box (Figure 10-8).



Progress

Compiling hello.c

Tool PowerPC C
Processing stdio.h

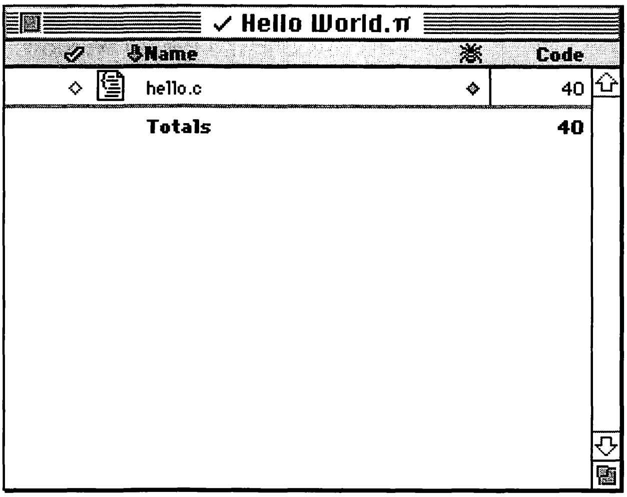
Lines: 0
Errors: 0
Warnings: 0

Figure 10-8 Progress dialog box

2. Watch as the dialog box charts the progress of the current compilation.

   The dialog box shows the current file being compiled, how many lines were processed, and how many errors and warnings were found. If the compilation was successful, then the `hello.c` file is added automatically to the project.

The Project window now lists the `hello.c` file with its code size (Figure 10-9).



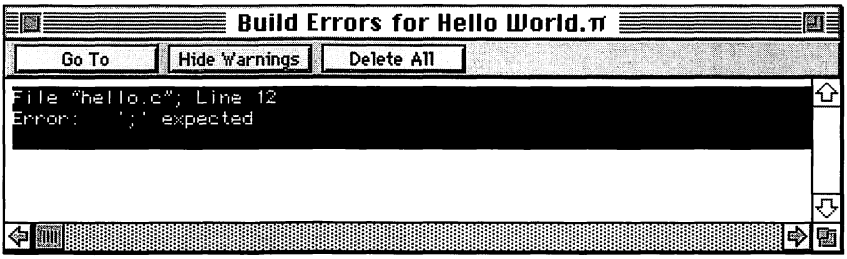
✓	Name		Code
◇	hello.c	◇	40
	Totals		40

Figure 10-9 Project window listing the hello.c file

3. Close the Editor window by clicking its close box.

   You can always bring the window up again by double-clicking `hello.c` in the Project window.

Note that if errors occurred during the compilation, the Build Errors window opens. This window lists all errors found in your file. For example, if you omitted the semicolon from the end of the `printf` statement, the window displays the error shown in Figure 10-10.



```
File "hello.c"; Line 12
Error:    ';' expected
```

Figure 10-10 Build Errors window

In this case, you would double-click on the error message in the window to open the Editor window with the offending line highlighted. After you have resolved all the errors in your code, compile the file again. Your source file is added to the project and displayed in the Project window.

The next step is to add the libraries necessary to link your project.

Adding the libraries

At this point, your project cannot be linked properly because the standard libraries used by it are not yet part of the project. To add the necessary libraries:

1. Choose **Add Files** from the **Project** menu.

   The **Add Files** dialog box opens (Figure 10-11).

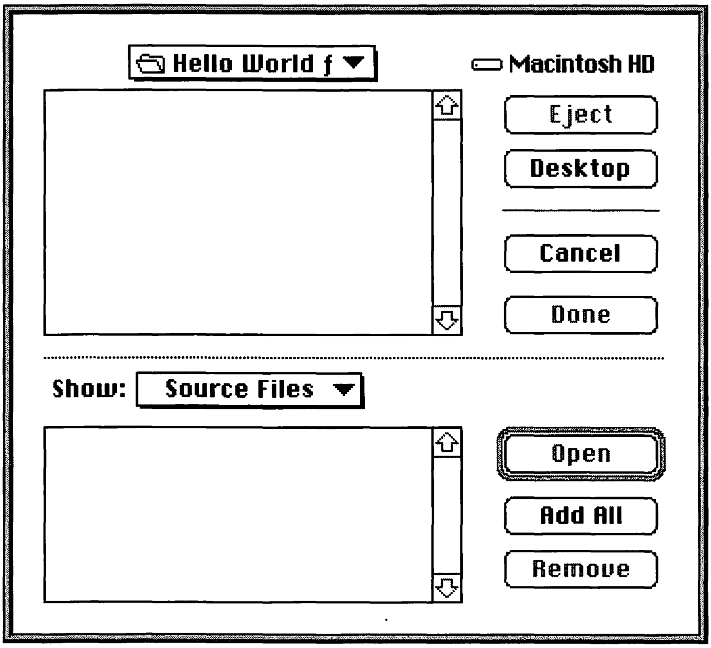


Figure 10-11 Add Files dialog box

At the top of this dialog box is a scrolling list that contains the names of the files in the current directory that are not part of your project. The bottom list contains the files to be added to the project after you click the Done button.

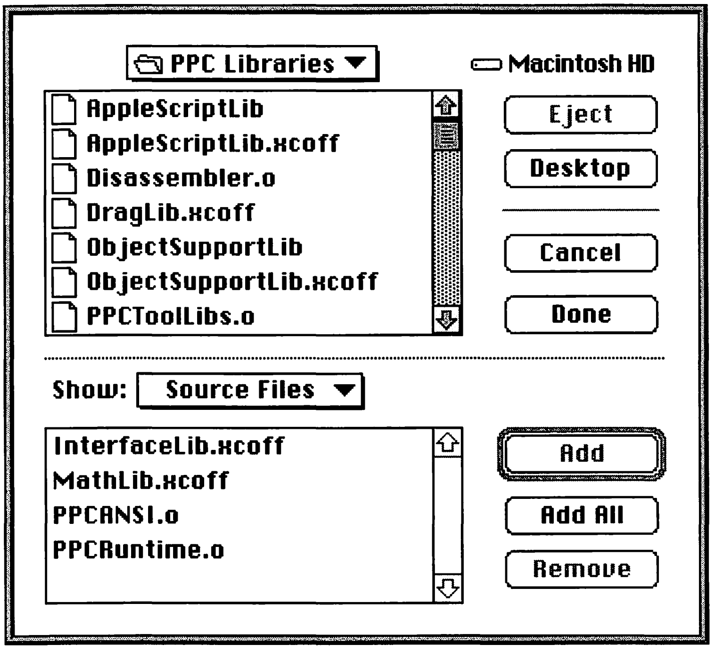
2. Navigate to the `Standard Libraries` folder within the Symantec Project Manager folder (by default, this is the `Symantec C++ for Power Mac` folder).

3. Select the `PPCANSI.o` file in the top list and click Add. Also select the `PPCRuntime.o` file and click Add.

   Notice how the names of the two object files, `PPCANSI.o` and `PPCRuntime.o`, are listed at the bottom of the dialog box.

4. Navigate to the `PPC Libraries` folder within the `Macintosh Libraries` folder in the Symantec Project Manager folder, and add the `InterfaceLib.xcoff` and `MathLib.xcoff` files in the same way as above.

The **Add Files** dialog box should now be displayed as shown in Figure 10-12.



Show: Source Files

Figure 10-12 Add Files dialog box with library files at the bottom

5. Click the Done button to close the **Add Files** dialog box and add the files to the project.

The Symantec Project Manager loads a library automatically when you run the project. Alternatively, you can click the library's name in the Project window, then choose **Compile** from the **Build** menu. For this example, let the Symantec Project Manager load it for you when you run the project.

Note

In this tutorial, you have manually added a source file and several library files to an empty project to learn how the Symantec Project Manager works. The next time you write a similar application, base the project on the ANSI C project model, which automatically adds the correct libraries needed for an application similar to Hello World.

Your project is complete. Now you need to build it and run it.

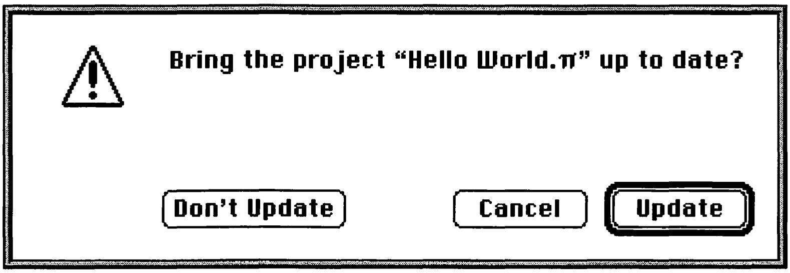
Building and running the application

To run the application without the Debugger:

*The **Run** command creates an "instant run image" that is very similar to an application on disk, but is not permanent. To save a permanent copy of an application, you would use the **Build Application** command, described later in this tutorial.*

1. Hold down the Option key and choose **Run** from the **Project** menu.

   Because you added libraries to your project, the project needs to be updated. The Symantec Project Manager prompts you to do so, as shown in Figure 10-13.



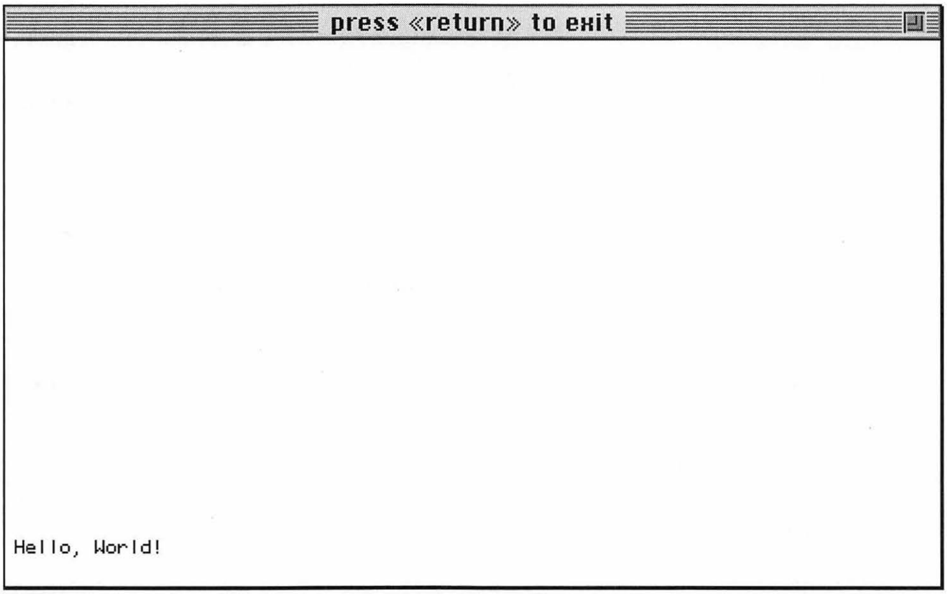
Bring the project "Hello World.π" up to date?

Figure 10-13 Update dialog box

When you bring your project up to date, the Symantec Project Manager compiles your project's files and links the project.

2. Click Update.

The Symantec Project Manager compiles and links the necessary files, then runs the program in a console window, as shown below:



```
Hello, World!
```

Figure 10-14 Hello World running

This program uses the `Standard` library to send output to a console window. A console window is a Macintosh window that behaves like a simple display terminal. The words `Hello, World!` are displayed at the bottom of this window.

To exit the program, press Return or choose **Quit** from the **File** menu.

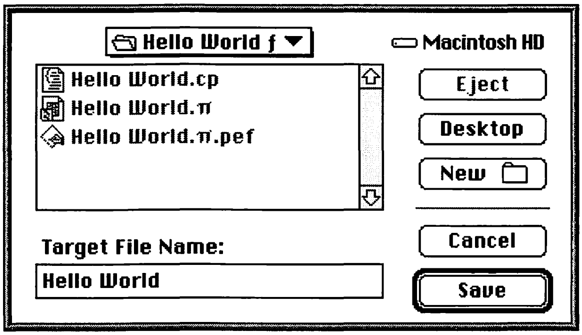
Creating the application

Now that you have seen your application run, you might want to save the application to disk. To make your project into a standalone, double-clickable Macintosh application:

1. Choose **Build Application** from the **Build** menu.

   The **File Save** dialog box opens (Figure 10-15).

Make sure to move to your `Hello World`*ƒ* folder (the default folder may be among the library files) before clicking Save.



Hello World.cp
Hello World.π
Hello World.π.pef

Target File Name: Hello World

Figure 10-15 File Save dialog box

2. Type `Hello World` into the Target File Name textbox. This is the name of the application file.

3. Click Save.

   A dialog box informs you that the Symantec Project Manager is linking your application. When it is finished, the built application is located in the folder you chose.

To run the built application, go to the Finder and open the application's folder. Double-click the application's icon to see it run.

Hello World C++ Application

Now that you have completed the Hello World C application, you can create the Hello World C++ application. Building the C++ application shows you certain aspects of project management that differ slightly from the procedure for creating the C application.

The process needed to create the C and C++ Hello World applications is similar. The steps outlined for the C application are covered only briefly in this section.

Creating a project

First, create an empty project, just as you did for the Hello World C application. If the Symantec Project Manager is already running, choose **New** from the **File** menu to create a new project. For this project, name the folder `Hello World++ ƒ`, and name the project `Hello World++.π`. Make sure that in the **New Project** dialog box, you select Empty Project as the project model.

Adding a source file

Now you are ready to create a source file. To do so:

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

2. In the Editor window that opens, type the following source code:

```
/*****
* Hello World.cp
*
* The hello world program for
* Symantec C++ for Power Macintosh
*
*****/
#include <iostream.h>
void main()
{
   cout << "hello world!" << endl;
}
```

3. Save the file as `hello.cp` by choosing **Save As** from the **Project** menu, typing `hello.cp` in the Save file as textbox, and clicking OK.

   You compiled the source file for the C application. For the C++ version, however, you add the file to your project without compiling it. You will build your whole project later.

4. From the **Project** menu, select **Add "hello.cp"**.

   The `hello.cp` file is now listed in the Project window.

5. Close the `hello.cp` Editor window by clicking in its close box.

Next, you add the standard libraries to the project.

Adding libraries

Follow the steps outlined for the Hello World C application to add the following libraries to your C++ project:

- `PPCANSI.o`
- `PPCCPlusLib.o`
- `PPCIOStreams.o`
- `PPCRuntime.o`

All of these libraries are located in the `Standard Libraries` folder.

After those libraries are added to the project, you are ready to bring the project up-to-date.

Updating the project

Generally, the most common way of checking for compile errors is to use the incremental build feature of Symantec C++, in which the Symantec Project Manager builds your project, translating only those files that have changed since the last time your project was built. To do this, choose **Bring Up To Date** from the **Build** menu. If files need to be compiled, the **Progress** dialog box opens to show the progress of the update.

When the update is finished, the **Progress** dialog box closes. If there were errors in your files, the Build Errors window opens, showing you the location of the errors. Double-click on any entry in the Build Errors window to bring up the source file where the error was found with the offending line highlighted. After you have corrected the errors, save the file, and update the project again.

To run your project now, hold down the Option key and choose **Run** from the **Project** menu.

You also can save your application to disk by choosing **Make Target** from the **Project** menu.

Tutorial: MiniEdit ◆ 11

The MiniEdit tutorial shows you how to use some of the advanced features of the Symantec Project Manager. You build a small text editor based on a sample application described in *Inside Macintosh I.*

In this tutorial you create and run a project, build an application, and use a resource file. One of the source files has a small, intentional bug, allowing you to practice fixing such errors.

Before You Begin

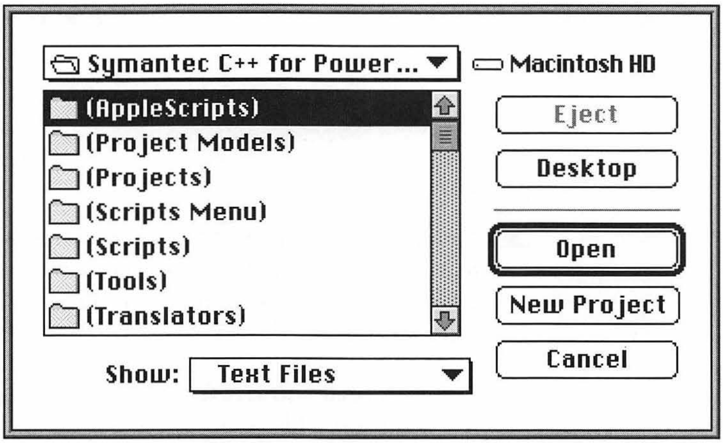
Make sure the `MiniEdit ƒ` folder is on your hard disk, because it contains all the files you need to follow this tutorial. If you followed the installation directions in Chapter 1, "Overview" this folder should be in the `Demos` folder, which is contained in the `Demo Projects` folder. In addition, if you did not work through the Hello World tutorial in the last chapter, you should consider doing so now. That tutorial serves as a quick introduction to the features of the Symantec Project Manager.

Creating the Project

Your first task in the MiniEdit tutorial is to create a project. To do so:

1. Open the folder containing the Symantec Project Manager and double-click its icon.

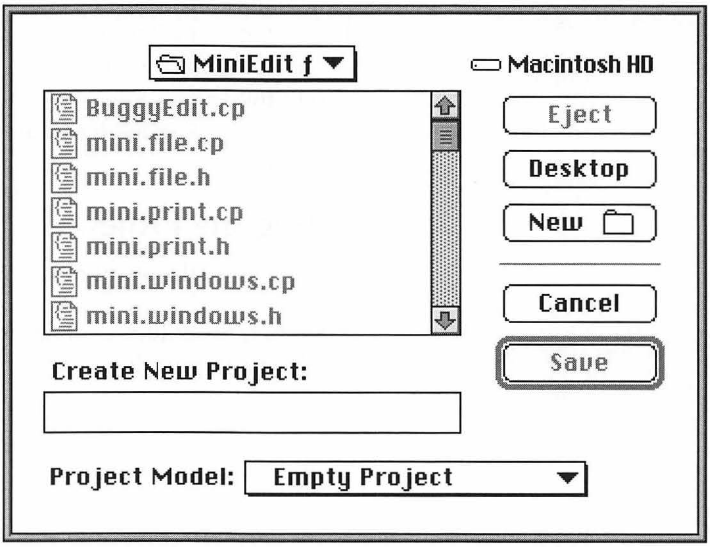
A standard **File Open** dialog box is displayed (Figure 11-1).



Show: Text Files

Figure 11-1 File Open dialog box

2. Navigate to the `MiniEdit ƒ` folder and click the New Project button to open the standard **New Project** dialog box (Figure 11-2).



Create New Project:

Project Model: Empty Project

Figure 11-2 New Project dialog box

3. Choose **C++ Mac Application** from the **Project Model** pop-up menu.

4. Type `MiniEdit.π` in the Create New Project textbox and click Save.

   The Symantec Project Manager creates a new project in the `MiniEdit ƒ` folder named `MiniEdit.π` and displays a Project window (Figure 11-3).

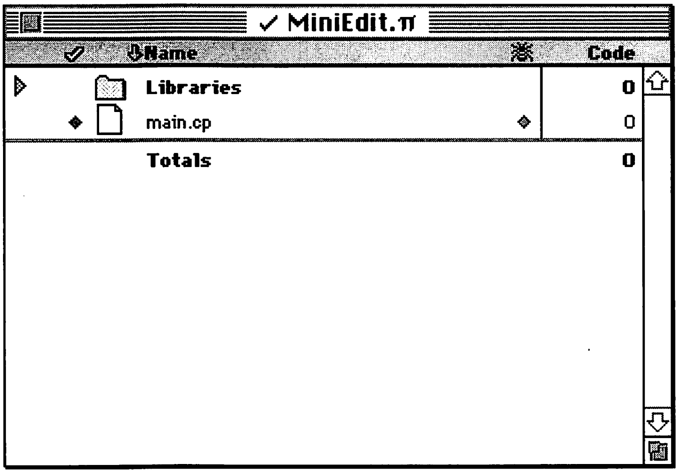


Figure 11-3 Project window for a new Project file

Because all the source files for this application are already written, you need to remove `main.cp` from the project:

1. Click `main.cp` in the Project window.

2. Choose **Remove "main.cp"** from the **Project** menu.

   This removes the file from the project; as a result, the `main.cp` file is no longer listed in the Project window. The file, however, is still in your project folder.

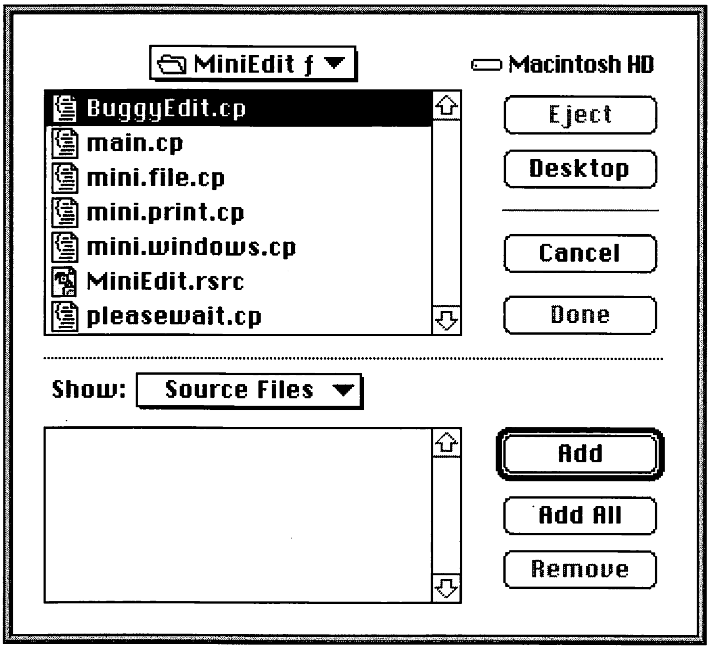
Note

Another option is to drag `main.cp` from the Project window to the Trash. This does not delete the file from your disk. To do that, use the Finder in the standard manner.

Adding the Source Files

The second task is to add source files to the new project. All the source files for `MiniEdit.π` are in the `MiniEdit ƒ` folder.

1. Choose **Add Files** from the **Project** menu to open the **Add** dialog box (Figure 11-4).



Show: Source Files

Figure 11-4 Add dialog box

The top list displays the source files and libraries in the current folder. The bottom list indicates those files that will be added to the project when you click Done.

2. Click Add All to move all the source files in the `MiniEdit ƒ` folder into the bottom list.

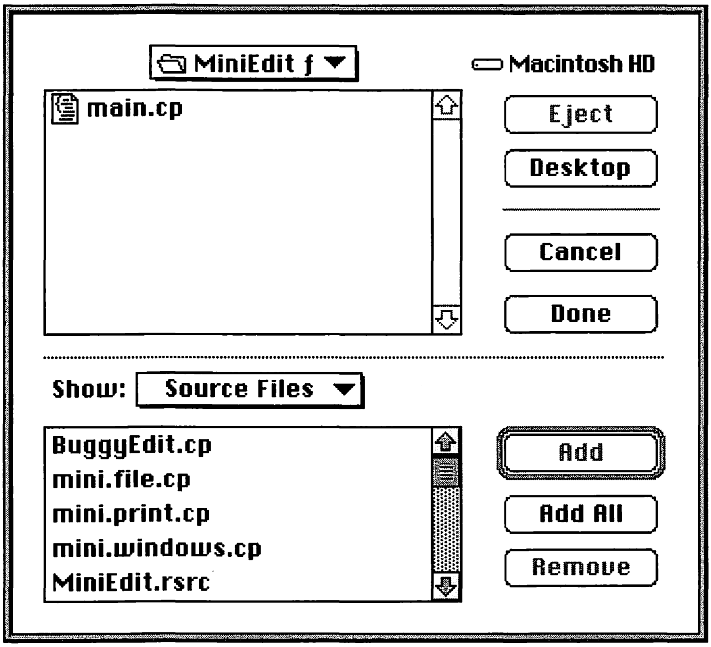
3. Select `main.cp` in the bottom list and click Remove.

   Because `main.cp` is still located in the project folder, it was added to the bottom list along with all the other files in the directory. Because including this file in your project would cause link errors, you should be sure not to re-add it to the project.

Note

If you are willing to select each file in the directory individually, you could click the Add button rather than Add All. With this approach, you do not have to remember to remove the `main.cp` file from the bottom list.

Figure 11-5 shows the **Add** dialog box with the source files displayed in the lower list.



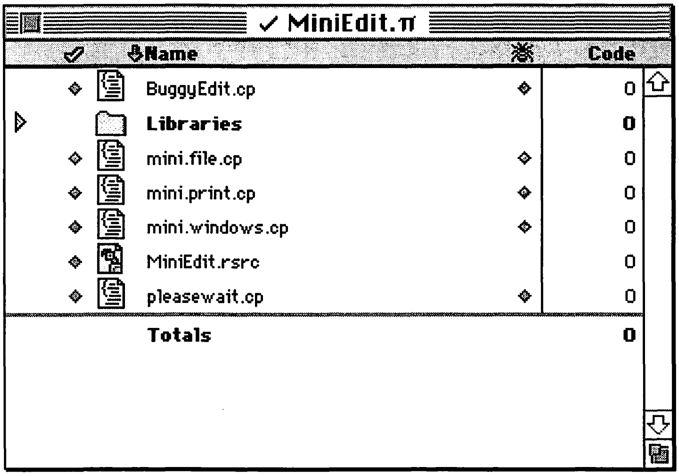
main.cp

Show: Source Files

BuggyEdit.cp
mini.file.cp
mini.print.cp
mini.windows.cp
MiniEdit.rsrc

Figure 11-5 MiniEdit source files ready to be added to the project

4. Click Done. The Project window changes to look like the one in Figure 11-6.



✓ MiniEdit.π

✓	Name	✱	Code
◈	BuggyEdit.cp	◈	0
▷	**Libraries**		**0**
◈	mini.file.cp	◈	0
◈	mini.print.cp	◈	0
◈	mini.windows.cp	◈	0
◈	MiniEdit.rsrc		0
◈	pleasewait.cp	◈	0
	Totals		**0**

Figure 11-6 MiniEdit.π Project window with appropriate files

Note that the Code column displays the object size in bytes for each file. The sizes currently are zero because you have not compiled any files or loaded any libraries.

Compiling and Running the Project

For your project to run, the source files must be compiled and the necessary libraries must be loaded. You can use the **Compile** or **Bring Up To Date** commands from the **Build** menu or the **Run** command from the **Project** menu to compile the files. The Symantec Project Manager uses the Project file to keep track of the files that need to be compiled and performs that task automatically when you run the project.

The Debugger is explored in the next tutorial. For now, you run the project without it. To set your application to run without the Debugger:

1. Choose **Options** from the **Project** menu.

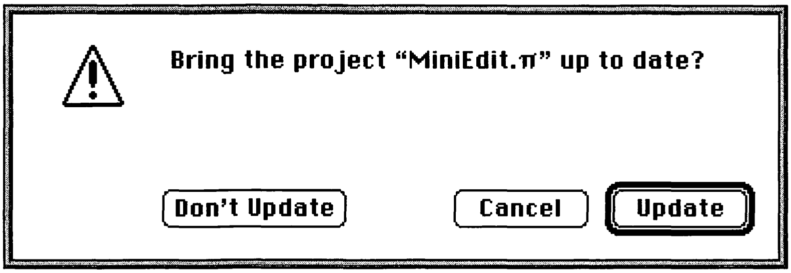
2. On the Project Options page of the **Project Options** dialog box, set the option Run with Debugger off.

3. Click Save.

Now run the application:

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

   None of the files in the project has been compiled, so you are prompted to bring the project up-to-date (Figure 11-7).



Bring the project "MiniEdit.π" up to date?

Figure 11-7 Prompt to update the project

2. Click Update.

   The Symantec Project Manager starts compiling the first file in the project. The **Progress** dialog box indicates the number of lines that have been compiled.

Note

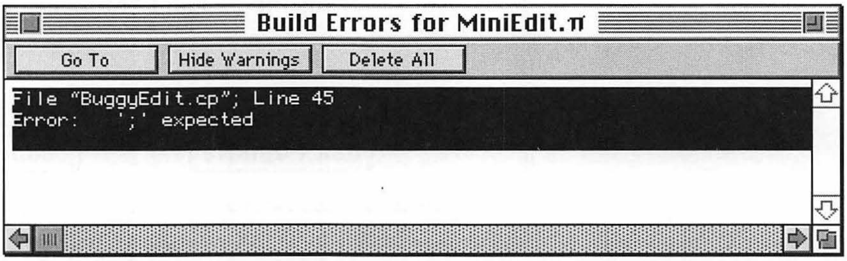
The Symantec Project Manager adds the number of lines in `#include` files in the line count.

Because `BuggyEdit.cp` has a small intentional bug, the Symantec Project Manager opens a Build Errors window, and your application is not run. Fixing the bug is discussed in the next section.

Fixing a Bug

When the Symantec Project Manager finds an error in a source file, it opens a Build Errors window and displays an error message. The compiler continues compiling the rest of the files in the project. To fix the bug, recompile the program, and run it:

1. Read the error message displayed in the Build Errors window (Figure 11-8).

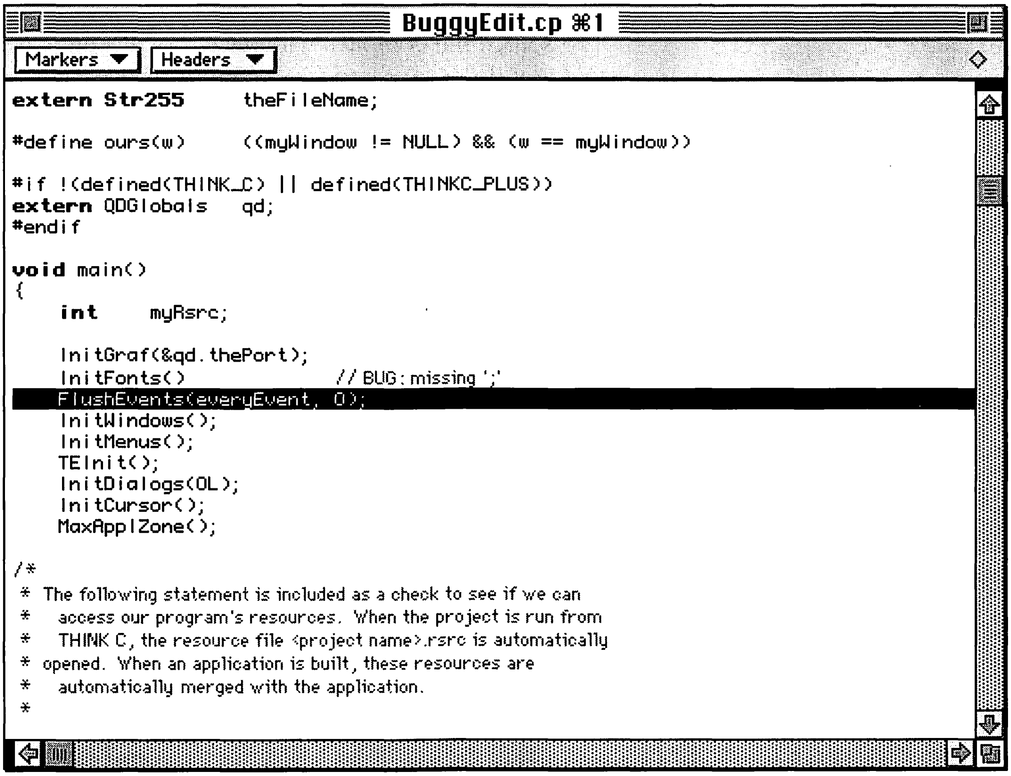


```
File "BuggyEdit.cp"; Line 45
Error:    ';' expected
```

Figure 11-8 Build Errors window

2. Double-click the error message in the Build Errors window to open the Editor window with the line causing the error highlighted.

As you see in Figure 11-9, a semicolon is missing.



```
extern Str255     theFileName;

#define ours(w)    ((myWindow != NULL) && (w == myWindow))

#if !(defined(THINK_C) || defined(THINKC_PLUS))
extern QDGlobals   qd;
#endif

void main()
{
    int     myRsrc;

    InitGraf(&qd.thePort);
    InitFonts()              // BUG: missing ';'
    FlushEvents(everyEvent, 0);
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(0L);
    InitCursor();
    MaxApplZone();

/*
 * The following statement is included as a check to see if we can
 *   access our program's resources. When the project is run from
 *   THINK C, the resource file <project name>.rsrc is automatically
 * opened. When an application is built, these resources are
 *   automatically merged with the application.
 *
```

Figure 11-9 BuggyEdit file with a syntax error

In this example, the C++ compiler wants a semicolon before the statement of line 45, but stylistically it is preferable to place one at the end of line 44.

3. Add the missing semicolon.

4. Compile the file `BuggyEdit.cp` using the **Compile** command from the **Build** menu.

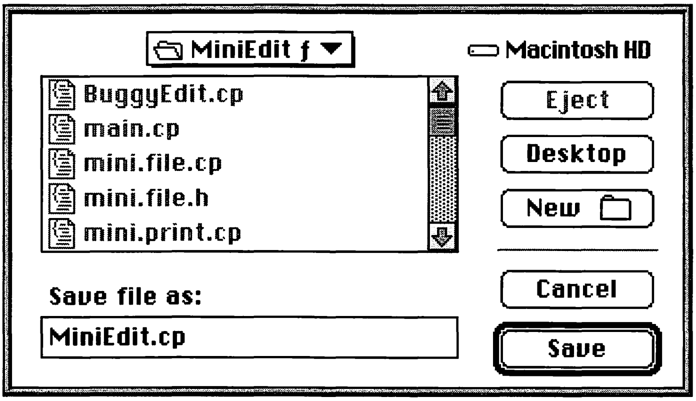
   The source file compiles without errors this time. Note that you do not have to save a file to recompile it.

   Because the `BuggyEdit.cp` file no longer contains a bug, you should save it with a different name.

5. Choose **Save As** from the **File** menu and save the corrected file as `MiniEdit.cp` (Figure 11-10).

Note

Make sure you save the file in the `MiniEdit ƒ` folder.



Save file as: MiniEdit.cp

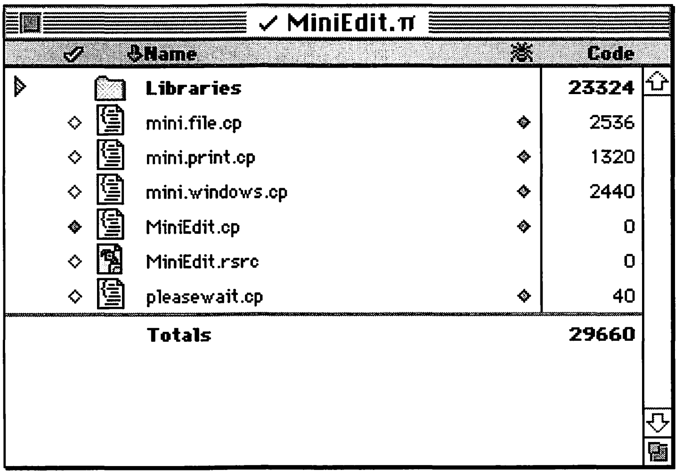
Figure 11-10 File Save dialog box

6. Now click the Project window.

   When you save a file that is already in the project using **Save As**, the file's name is also changed in the Project window.

*To save a file with a different name without affecting the project, use the **Save A Copy As** command.*

The file's object code is now associated with the new name (Figure 11-11).



✓ MiniEdit.π

✓	Name		Code
	Libraries		**23324**
◇	mini.file.cp	◆	2536
◇	mini.print.cp	◆	1320
◇	mini.windows.cp	◆	2440
◆	MiniEdit.cp	◆	0
◇	MiniEdit.rsrc		0
◇	pleasewait.cp	◆	40
	Totals		**29660**

Figure 11-11 MiniEdit.π Project window with a new filename

Running the Project Again

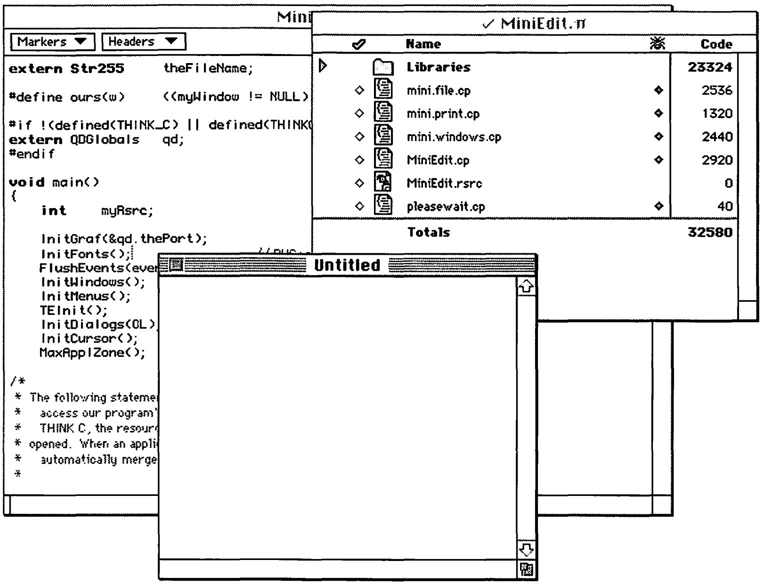
Now that you've fixed the bug, you can try running the project again.

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

   You are prompted to bring the project up-to-date.

2. Click Update.

   The Symantec Project Manager compiles the project and launches it (Figure 11-12).



```
extern Str255      theFileName;

#define ours(w)     ((myWindow != NULL)

#if !(defined(THINK_C) || defined(THINK(
extern QDGlobals   qd;
#endif

void main()
{
    int     myRsrc;

    InitGraf(&qd.thePort);
    InitFonts();
    FlushEvents(eve
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(0L)
    InitCursor();
    MaxApplZone();

/*
 * The following stateme
 *   access our program'
 *   THINK C, the resour
 * opened.  When an appli
 *   automatically merge
 *
```

✓ MiniEdit.π

	✓	Name	⚙	Code
▷		Libraries		23324
	◇	mini.file.cp	◆	2536
	◇	mini.print.cp	◆	1320
	◇	mini.windows.cp	◆	2440
	◇	MiniEdit.cp	◆	2920
	◇	MiniEdit.rsrc		0
	◇	pleasewait.cp	◆	40
		Totals		32580

Figure 11-12 Running MiniEdit

3. Test the MiniEdit application.

You might want to experiment with program code. When you are satisfied with how the project runs, you're ready to turn it into a double-clickable application.

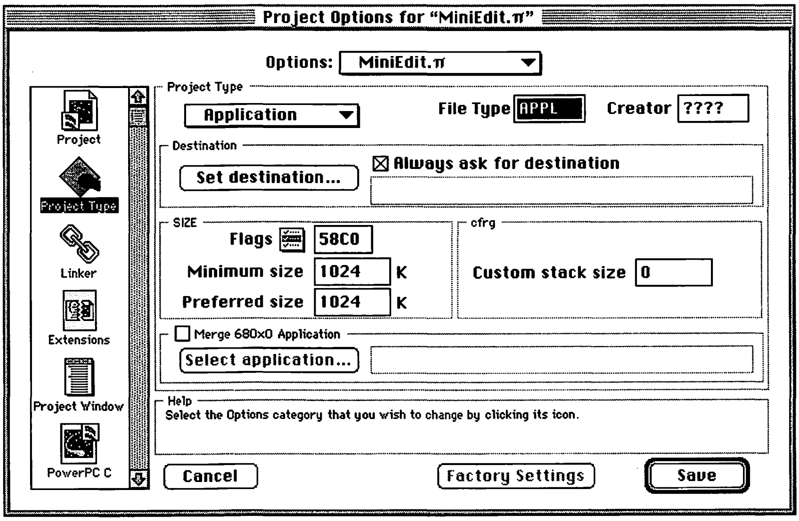
Building the Application

Now you are ready to turn the project into an application:

1. Choose **Options** from the **Project** menu to open the **Project Options** dialog box.

   This dialog box has eight pages. You move among the pages by clicking the appropriate icon to the left.

2. Click the Project Type icon to open the Project Type page of the dialog box (Figure 11-13).



Project Options for "MiniEdit.π"

Options: MiniEdit.π

Project Type: Application

File Type: APPL

Creator: ????

Destination: ☒ Always ask for destination

SIZE: Flags 58C0

Minimum size 1024 K

Preferred size 1024 K

cfrg: Custom stack size 0

☐ Merge 680x0 Application

Help: Select the Options category that you wish to change by clicking its icon.

Figure 11-13 Project Type page

3. In the Creator textbox, type `CEM8`.

   This ensures that your application has the correct icon when you build it.

CEM8 *doesn't stand for anything. It was chosen because it is unlikely that any other application on your disk has that signature.*

4. In the Minimum Size and Preferred Size textboxes, change the default values to 256K.

   The Power Macintosh uses these values to determine how much memory to give to an application. Because MiniEdit is so small, it does not need the default 1024K size.

5. Use the **Flags** pop-up menu to turn off all of the flags except **32-bit Compatible**.

   MiniEdit is a simple program that does not take advantage of the advanced features of System 7.5.

Warning

Do not turn off these flags with any project with which you want to use the Symantec Debugger.

---

6. Click Save.

7. Choose **Build Application** from the **Build** menu (Figure 11-14).

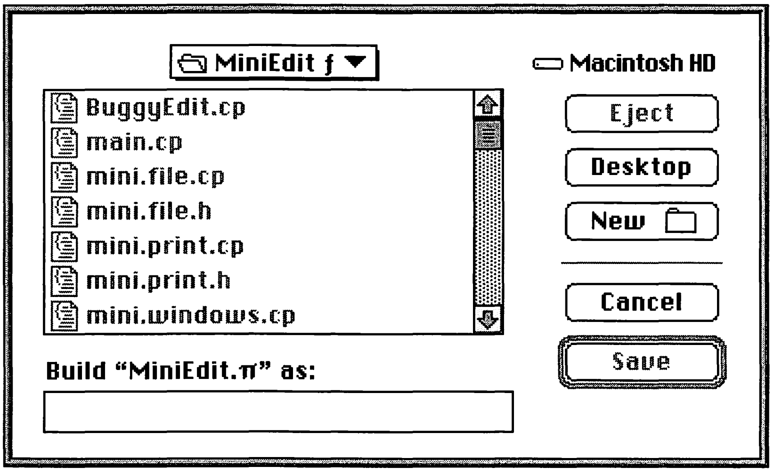


Figure 11-14 Building the MiniEdit application

8. Name the application `MiniEdit` and click Save.

   You now have a new application in the `MiniEdit ƒ` folder.

Using a Resource File

The MiniEdit ƒ folder now contains a file called MiniEdit.π.rsrc. This file contains the resources that the MiniEdit project uses.

When the Symantec Project Manager runs your project, it looks for a file named *projectname*.rsrc (that is, the name of your project plus the characters .rsrc appended to it). This file should contain the resources (such as menus, alerts, and dialogs) that your project uses.

In this case the file is named MiniEdit.π.rsrc. During updates, the Symantec Project Manager builds this file from all of the resource files (.rsrc files) and resource description files (.r files) that are included in your project. MiniEdit.π only includes one resource file, MiniEdit.rsrc. Thus MiniEdit.rsrc and MiniEdit.π.rsrc are identical. If you decide to edit any of the resources used by MiniEdit, be sure to edit them in MiniEdit.rsrc and not in MiniEdit.π.rsrc, because MiniEdit.π.rsrc is rebuilt every time your project is updated.

For more information on resource files, see "Using Symantec Rez" in the Symantec C++ Compiler Guide.

To create a resource file, you can use Symantec Rez or ResEdit. Both are included in your package. MiniEdit.rsrc was created with ResEdit, so there is no resource description (.r) file for it.

Finishing Up

When you're finished working on a project, you can either close the project or quit the Symantec Project Manager by choosing **Quit** from the **File** menu.

Tutorial: Object Bullseye ◆ 12

Object Bullseye shows you how to use the Symantec Project Manager's source-level Debugger. In the process, you build a simple application that draws a series of concentric shapes in a small window. With the **Width** menu you create as part of the application, you can select the width of each of the shapes.

Before You Begin

Make sure the `Object Bullseye ƒ` folder is on your hard disk. If you followed the directions in the section "Installing Symantec C++," in Chapter 1, "Overview," this folder should be inside the `Demos` folder, which is inside the `Project Demos` folder. Also make sure that the Symantec Debugger is in the same folder as the Symantec Project Manager and that `Power Mac DebugServices` is in the `(Tools)` folder.

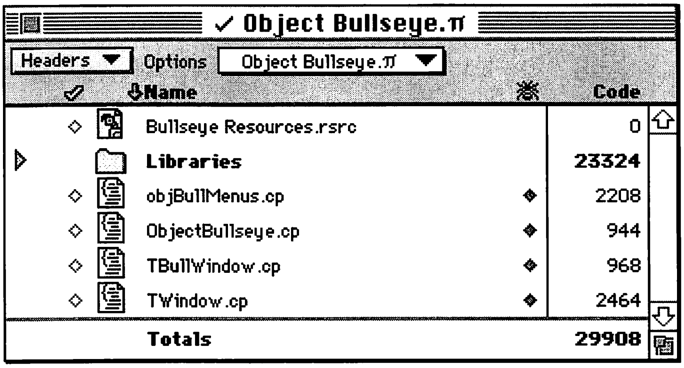
This tutorial assumes you understand the basic mechanics of the Symantec Project Manager. You should know how to open a project, edit source files, and run a project. If you are not familiar with any of these operations, review the previous two tutorials.

Preparing to Use the Debugger

When you are ready to start the Object Bullseye tutorial, you should take the following two preparatory steps:

1. Open the project by double-clicking `Object Bullseye.π` in the `Object Bullseye` folder. Alternatively, select the file and choose **Open** from the Symantec Project Manager's **File** menu.

Object Bullseye consists of four source files, one resource file, and a few libraries (Figure 12-1).



✓ Object Bullseye.π

Headers ▼ Options Object Bullseye.π ▼

✓	Name	🐞	Code
◇	Bullseye Resources.rsrc		0
▷	**Libraries**		**23324**
◇	objBullMenus.cp	◆	2208
◇	ObjectBullseye.cp	◆	944
◇	TBullWindow.cp	◆	968
◇	TWindow.cp	◆	2464
	Totals		**29908**

Figure 12-1 Object Bullseye.π Project window

Note that none of the files has been compiled (the Code field indicates that the code size of each file is 0 bytes).

2. Choose **Options** from the **Project** menu to open the **Project Options** dialog box.

3. On the Project Options page of the **Project Options** dialog box, set on the option Run with Debugger.

   To be able to use the Debugger, your compiled code must contain debug information. By default, the generation of debug information is enabled for this tutorial.

Note

The generation of debug information is controlled from two locations. First is the **Project Options** dialog box (the Debugging subpage of the Power PC C++ Options page). Second is the Debug column in the Project window. As you can see in Figure 12-1, the diamonds in this column are filled by default, indicating that the Symantec Project Manager will generate debugging information for the source files.

Starting a Debugging Session

To start a debugging session:

1. Choose **Run with Debugger** from the **Project** menu.

   The Symantec Project Manager now compiles and loads all the files in the Object Bullseye project. It launches the Symantec Debugger, which opens the Main debugging window and Control palette by default.

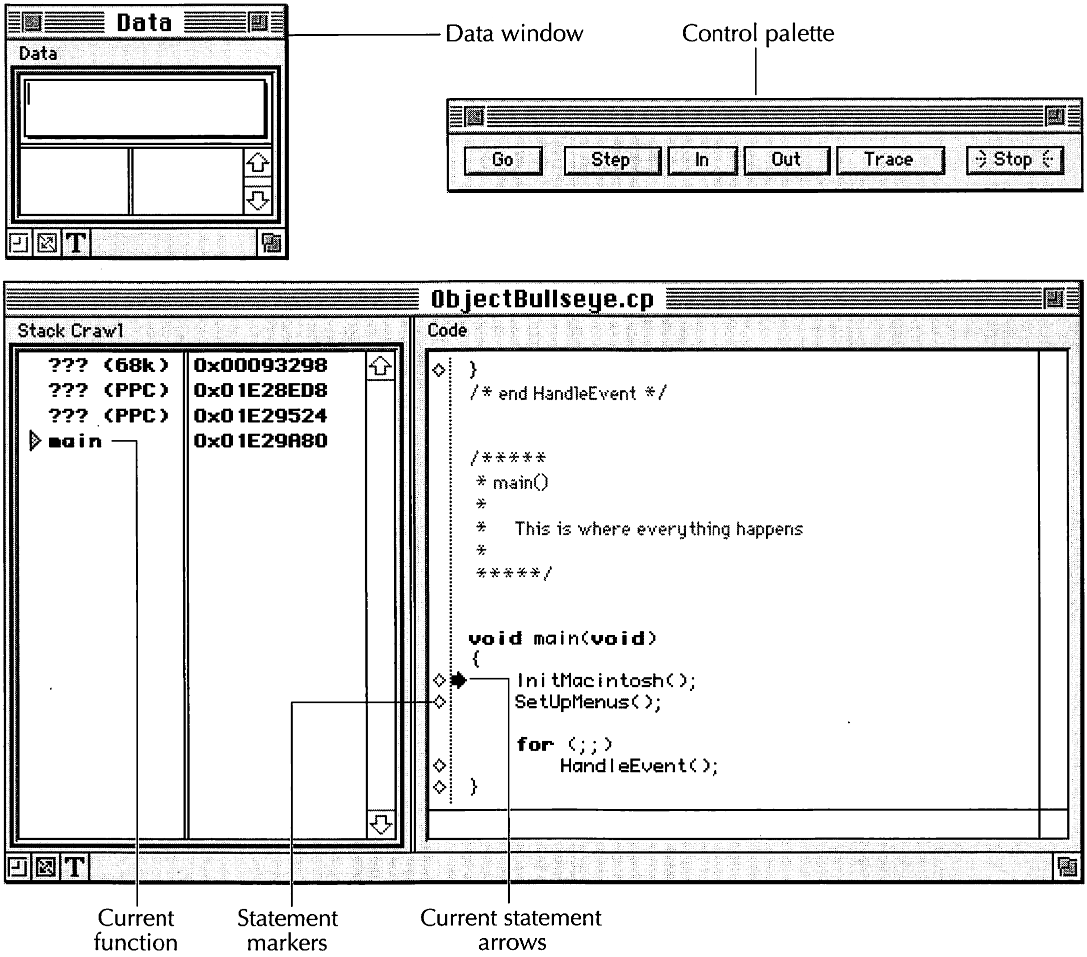
Note

If you had not selected Run with Debugger from the **Project Options** dialog box, the command on the **Project** menu would be titled **Run**. You can use the Option key to toggle between **Run** and **Run with Debugger**.

2. Position the debugging windows on the screen as desired (Figure 12-2).

   The Main debugging window contains two panes, the Code pane and the Stack Crawl pane. The Control palette contains buttons that both control the current process and reflect the state of that process. The Control palette is free-floating.

3. If the Data window is not displayed, open the window by selecting **Data** from the **Windows** menu.



```
??? (68k)   0x00093298
??? (PPC)   0x01E28ED8
??? (PPC)   0x01E29524
main        0x01E29A80
```

```
}
/* end HandleEvent */


/*****
 * main()
 *
 *   This is where everything happens
 *
 *****/


void main(void)
{
    InitMacintosh();
    SetUpMenus();

    for (;;)
        HandleEvent();
}
```

Figure 12-2 Debugger windows

Control palette

The Control palette provides an interactive mechanism for controlling the execution of your program. The names of its buttons match the commands in the Debugger's **Debug** menu. In Figure 12-2, the **Stop** button is highlighted, which indicates that the program is stopped.

Main debugging window

This window displays information about the process currently being debugged. Individual panes are printable. This window contains no close box and remains open throughout a debugging session.

Stack Crawl pane

The Stack Crawl pane displays your program's call chain: the name of the current function and the names of the functions that were called to get to the current function. To the right of the function names, it displays the hex location of the function in memory. You can use this pane to examine the variables in any function by clicking on the triangle next to the function name.

Code pane

The Code pane shows the source text of your program. When you start the Debugger, this pane shows the file that contains the `main()` routine of your application. The black arrow to the left of the first line of the program is the current statement arrow. This indicator shows you the current statement, the one the Debugger is about to execute.

The column of diamonds running along the left side of the Source text contains statement markers. Every line of your program that generates code gets a statement marker. Later, you will use the statement markers to set breakpoints.

Data window

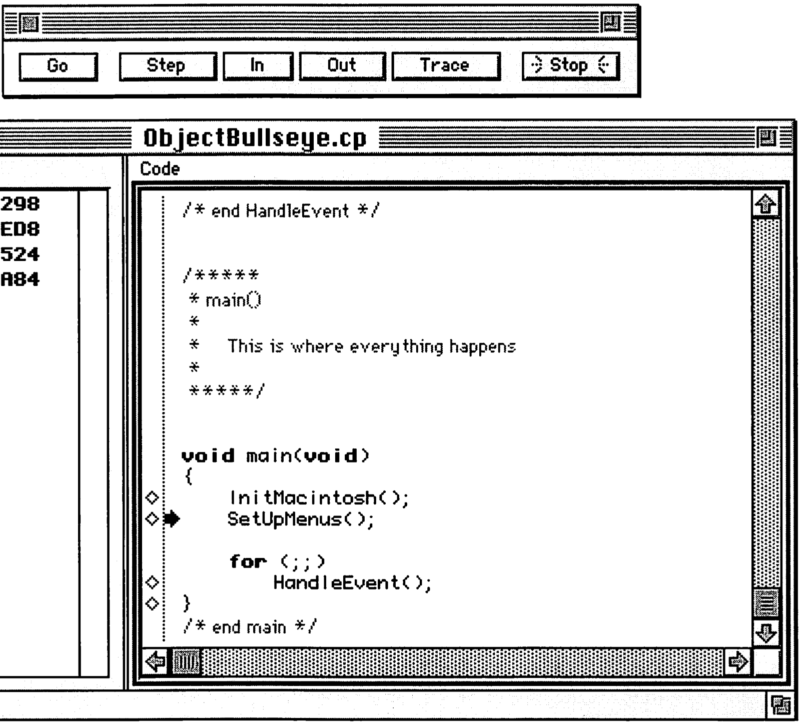
The Data window is used to examine the value of any expression. These may be constants or function results, but the most common use of the Data window is to examine the value of your program's variables.

Controlling Execution Flow

With the Debugger windows open, you are ready to experiment with execution flow by clicking the Control palette buttons and setting breakpoints.

Stepping through statements

Click **Step** in the Control palette (Figure 12-3).



```
??? (68k)   0x00093298
??? (PPC)   0x01E28ED8
??? (PPC)   0x01E29524
main        0x01E29A84
```

```
/* end HandleEvent */


/*****
 * main()
 *
 *   This is where everything happens
 *
 *****/


void main(void)
{
    InitMacintosh();
    SetUpMenus();

    for (;;)
        HandleEvent();
}
/* end main */
```

Figure 12-3 Stepping through the program

The **Step** button lights up for a moment, the current statement arrow moves to the second statement, and the program stops again.

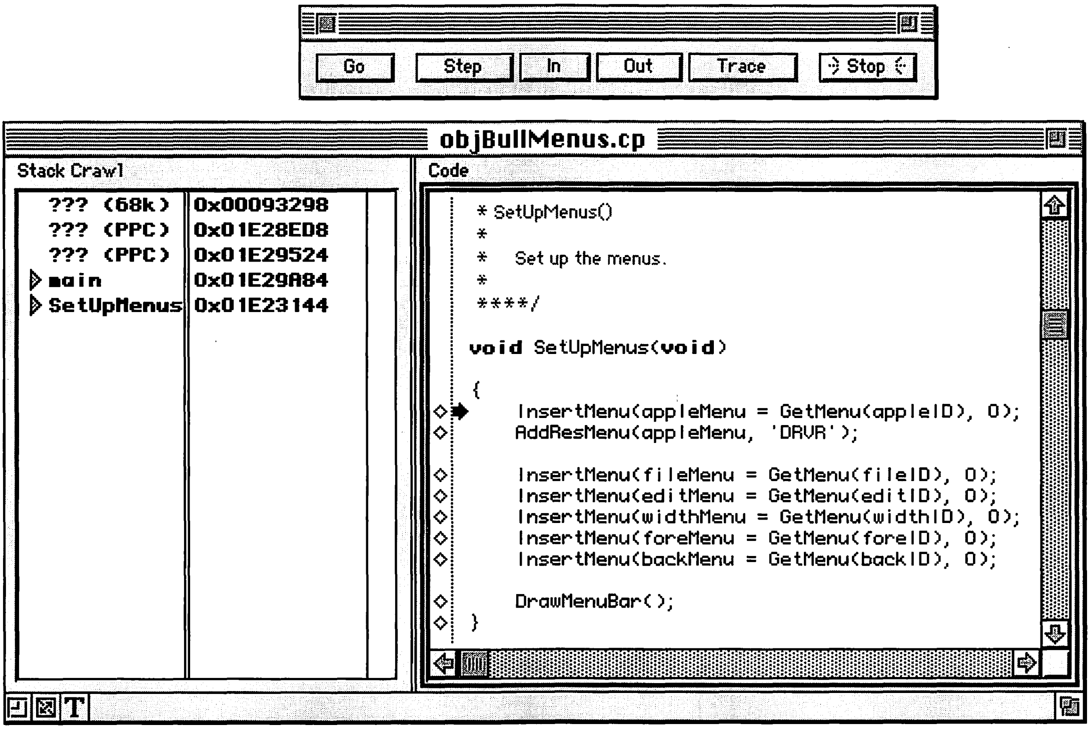
The **Step** button lets you execute your program line by line. You can also choose **Step** from the **Debug** menu or press Command-S.

Stepping into functions

*Instead of clicking **In**, you can also choose **Step In** from the **Debug** menu or press Command-I.*

Now the current statement arrow is pointing to the call to `SetupMenus()`. This function sets up the menus for Object Bullseye. To see how `SetUpMenus()` works, click **In** on the Control palette.

Now the current statement arrow points to the first line of the `SetUpMenus()` function (see Figure 12-4).



```
??? (68k)    0x00093298
??? (PPC)    0x01E28ED8
??? (PPC)    0x01E29524
main         0x01E29A84
SetUpMenus   0x01E23144
```

```
 * SetUpMenus()
 *
 *   Set up the menus.
 *
 ****/

void SetUpMenus(void)

{
    InsertMenu(appleMenu = GetMenu(appleID), 0);
    AddResMenu(appleMenu, 'DRVR');

    InsertMenu(fileMenu = GetMenu(fileID), 0);
    InsertMenu(editMenu = GetMenu(editID), 0);
    InsertMenu(widthMenu = GetMenu(widthID), 0);
    InsertMenu(foreMenu = GetMenu(foreID), 0);
    InsertMenu(backMenu = GetMenu(backID), 0);

    DrawMenuBar();
}
```

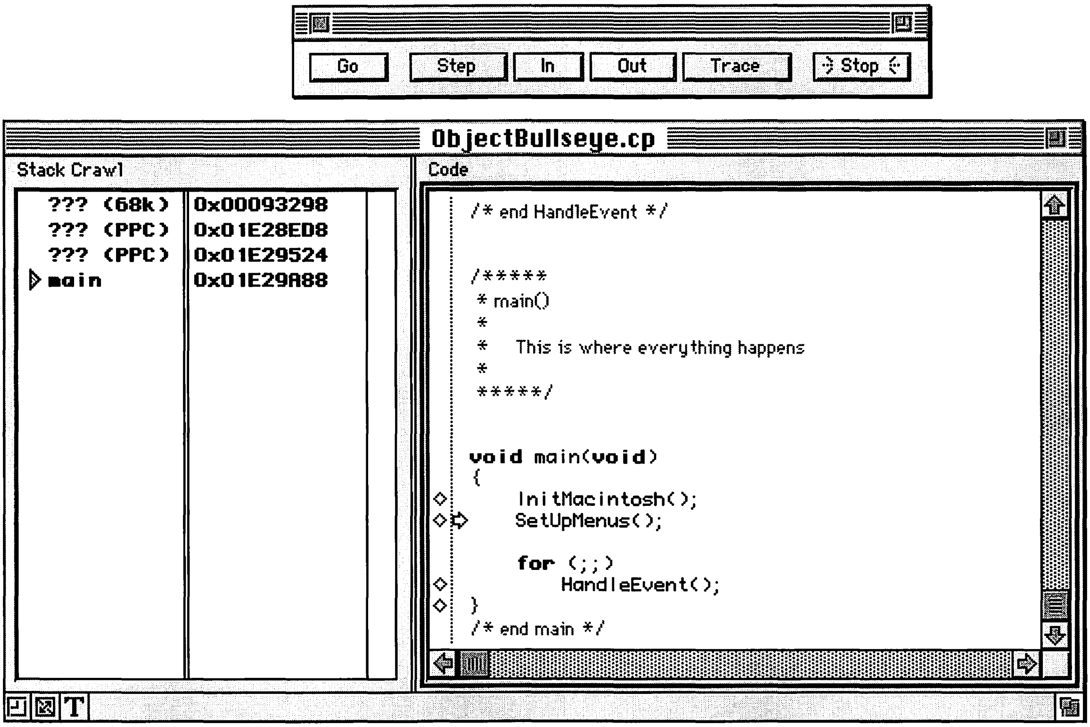
Figure 12-4 Inside SetUpMenus()

Note

The current statement arrow does not have to be right before a function call for the **In** button to work. The **Step In** command executes every statement until the program counter is no longer in the current function. Another way to think of the **Step In** command is: “Keep going until you fall into a function.” **Step In** also stops execution if you fall out of the current function.

Stepping out of functions

Click **Out** to leave the `SetUpMenus()` function. The Code Pane now shows that the Debugger has just finished executing the function `SetUpMenus()`. The hollow arrow indicates this (see Figure 12-5). Sometimes you will also see a down-pointing arrow indicating that the line contains a function call that has not yet returned.



??? (68k)	0x00093298
??? (PPC)	0x01E28ED8
??? (PPC)	0x01E29524
▷ main	0x01E29A88

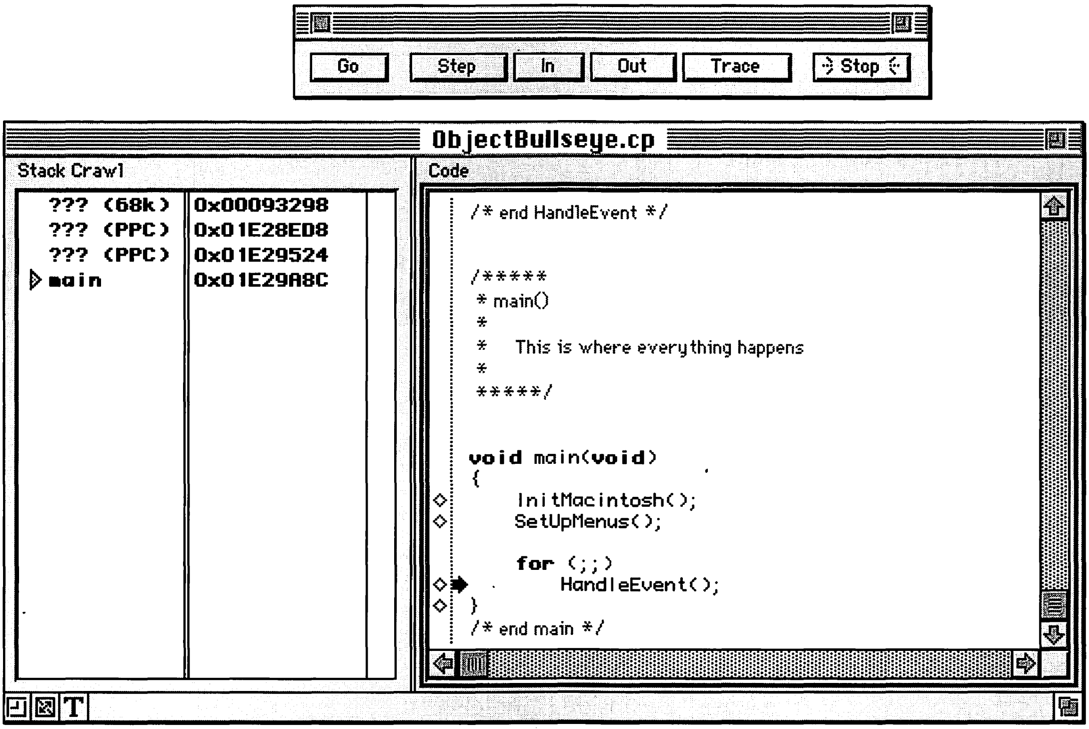
```
   /* end HandleEvent */


   /*****
    * main()
    *
    *   This is where everything happens
    *
    *****/


   void main(void)
   {
◇      InitMacintosh();
◇⇨     SetUpMenus();

       for (;;)
◇          HandleEvent();
◇  }
   /* end main */
```

Figure 12-5 Outside SetUpMenus()

The **Out** button steps through each statement in the current function until the execution leaves the function.

Tracing every statement

Click **Step** once so the current statement arrow points to the call to `HandleEvent()` (Figure 12-6).



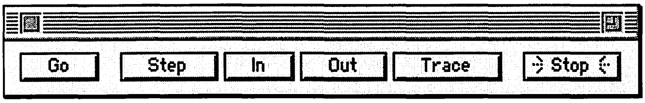
```
??? (68k)   0x00093298
??? (PPC)   0x01E28ED8
??? (PPC)   0x01E29524
main        0x01E29A8C
```

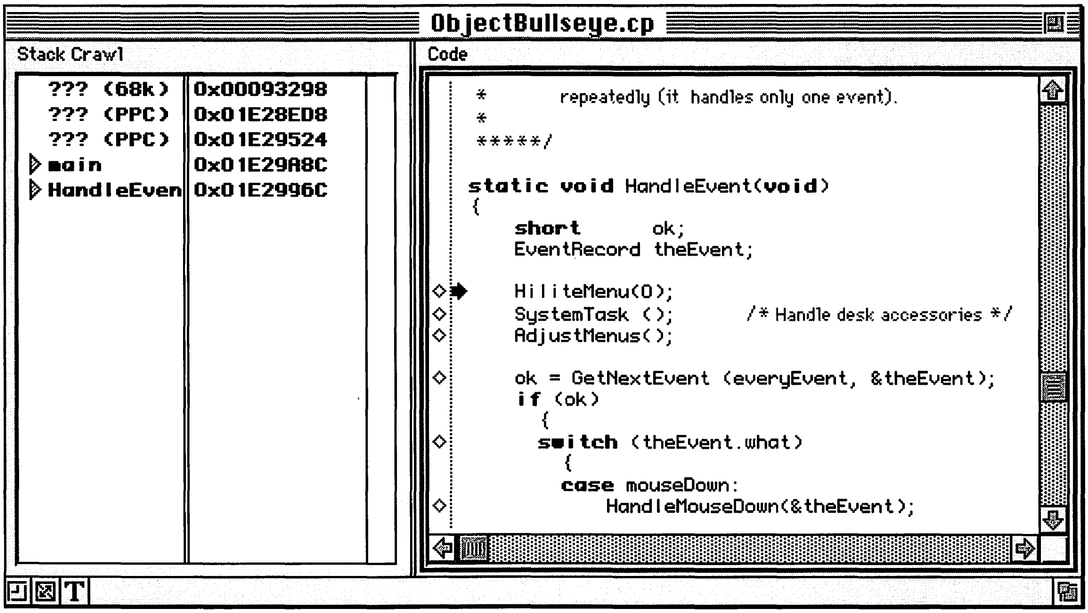
```
/* end HandleEvent */


/*****
 * main()
 *
 *   This is where everything happens
 *
 *****/


void main(void)
{
    InitMacintosh();
    SetUpMenus();

    for (;;)
        HandleEvent();
}
/* end main */
```

Figure 12-6 Current statement arrow points to HandleEvent()

*Instead of clicking **Trace**, you can choose **Trace** from the **Debug** menu or press Command-T.*

Now click **Trace**. The current statement arrow points to the first statement of the `HandleEvent()` function (Figure 12-7).





```
??? (68k)     0x00093298
??? (PPC)     0x01E28ED8
??? (PPC)     0x01E29524
main          0x01E29A8C
HandleEven    0x01E2996C
```

```
 *       repeatedly (it handles only one event).
 *
 *****/

static void HandleEvent(void)
{
    short       ok;
    EventRecord theEvent;

    HiliteMenu(0);
    SystemTask ();          /* Handle desk accessories */
    AdjustMenus();

    ok = GetNextEvent (everyEvent, &theEvent);
    if (ok)
      {
      switch (theEvent.what)
        {
        case mouseDown:
            HandleMouseDown(&theEvent);
```

Figure 12-7 Inside HandleEvent()

Tracing takes you to the next statement even if it has to step into a function. If you were to continue tracing, you would stop at every statement. Stepping, on the other hand, *never* dives into a function.

Note

The **In** button actually does a trace until the current statement arrow leaves the current function.

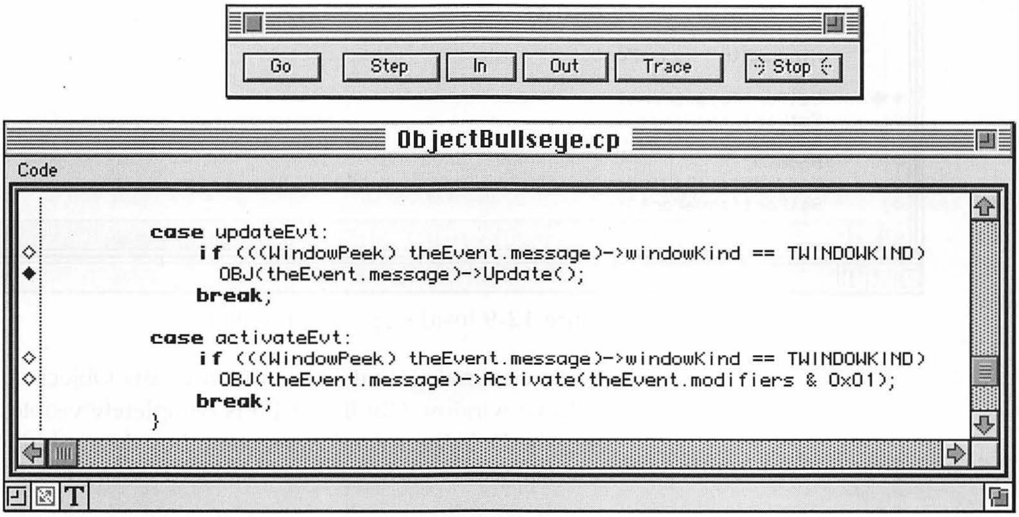
Setting a breakpoint

When a new window is created, the program gets an Activate event the first time through the event loop. In Object Bullseye, all the program does on Activate events is call `InvalRect()` on the whole window, so the second time through the event loop, it gets an Update event.

You could **Step** or **Trace** to verify that this is what really happens. A faster way is to set a breakpoint at the function that redraws the window:

1. Scroll down in the Code pane until you get to the code that handles Update events.

2. Click the statement marker to the left of the call to the `Update` function (see Figure 12-8).



```
          case updateEvt:
◇              if (((WindowPeek) theEvent.message)->windowKind == TWINDOWKIND)
◆                OBJ(theEvent.message)->Update();
               break;

          case activateEvt:
◇              if (((WindowPeek) theEvent.message)->windowKind == TWINDOWKIND)
◇                OBJ(theEvent.message)->Activate(theEvent.modifiers & 0x01);
               break;
          }
```

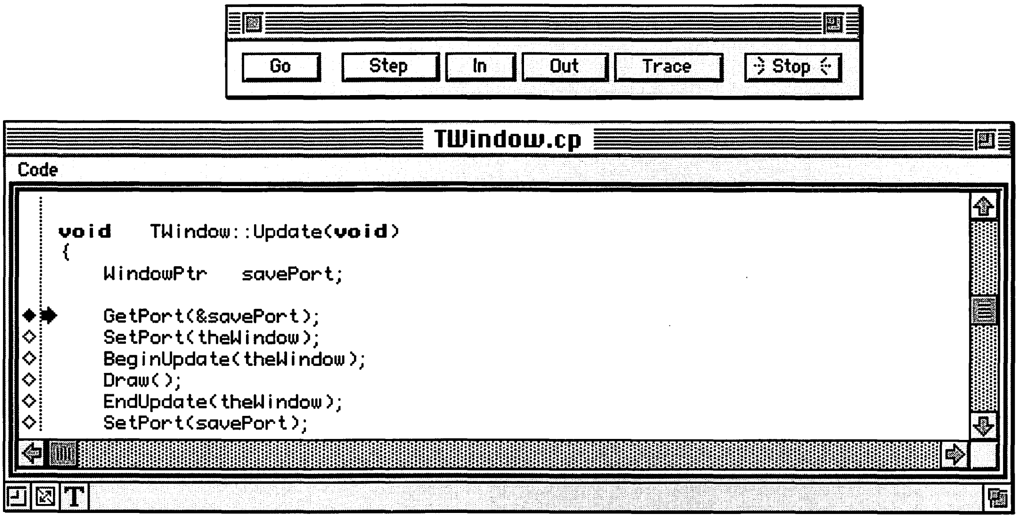
Figure 12-8 Setting a breakpoint

The hollow diamond fills in to indicate that you have set a breakpoint. You can set as many breakpoints as you like this way. When your program is about to execute a statement that has a breakpoint, it will stop. To remove a breakpoint, click the filled diamond.

3. To start your program running, click **Go**.

4. Now select **New Circle** from the **File** menu to open a new window. The Debugger is brought forward with the current statement arrow at your breakpoint.

5. Click **In** to step into the `Update()` function (Figure 12-9).



```
void    TWindow::Update(void)
{
    WindowPtr   savePort;

    GetPort(&savePort);
    SetPort(theWindow);
    BeginUpdate(theWindow);
    Draw();
    EndUpdate(theWindow);
    SetPort(savePort);
```

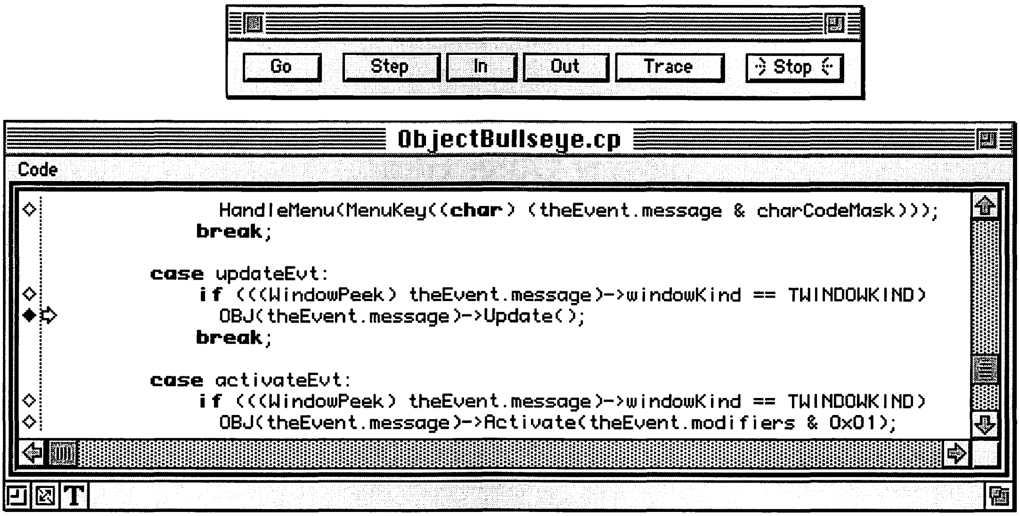
Figure 12-9 Inside Update() function

6. Before continuing, make sure that the new Object Bullseye window ("Bullseye 1") is completely visible. Drag the Debugger windows so that they do not hide the new window.

7. Click **Step** three times to watch how the program draws a bullseye in the window.

8. When the current statement arrow is pointing to the window's `Draw()` function, click **In** to step into the `Draw()` function, then use **Step** to see how `Draw()` works.

To stop, click **Out**. Whether you **Step**, **Trace**, or step **Out**, you eventually end back at the call to `Update()`. If you were inside `Draw()`, you will have to click the **Out** button twice; once to exit `Draw()` and a second time to exit `Update()` (Figure 12-10).



```
            HandleMenu(MenuKey((char) (theEvent.message & charCodeMask)));
          break;

        case updateEvt:
          if (((WindowPeek) theEvent.message)->windowKind == TWINDOWKIND)
            OBJ(theEvent.message)->Update();
          break;

        case activateEvt:
          if (((WindowPeek) theEvent.message)->windowKind == TWINDOWKIND)
            OBJ(theEvent.message)->Activate(theEvent.modifiers & 0x01);
```

Figure 12-10 Outside Update() function

Note that the current statement arrow is hollow. This means that there are still some instructions left to execute in the statement. You see right-pointing hollow arrows when the statement is making an assignment or cleaning up the stack after stepping out of a function. You see down-pointing hollow arrows when the line contains a function call that has not returned yet.

Before you go on, clear the breakpoint. Just click the diamond.

Letting the program run

Click **Go** to let the program run. You can set and clear breakpoints while your program is running.

When you click in the Main debugger window to set breakpoints, your application goes to the background and the Debugger comes to the foreground. If you click **Go** when your program is running, the Debugger brings it to the foreground.

Stopping the program

To stop your program:

1. Click a Debugger window.
2. Click **Stop** or press Command-Period.

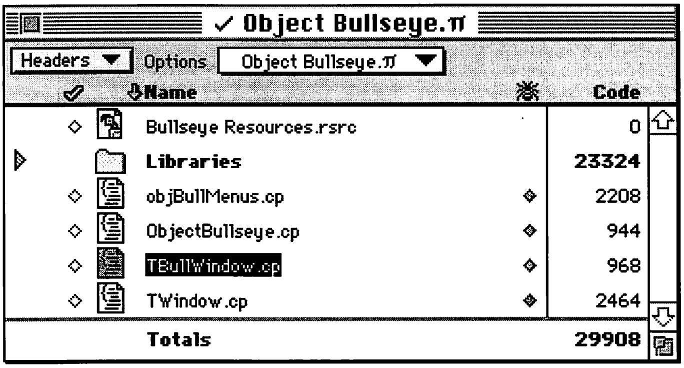
   Your program stops as it's coming out of one of the event-fetching routines (`GetNextEvent()` or `WaitNextEvent()`).

Viewing Other Files

The Code pane usually shows the file that contains the current statement. To look at another file in the Debugger (to set breakpoints in it, for example), you tell the Symantec Project Manager to send the text to the Debugger:

*Instead of clicking a file in the Project window, you can select a line in an open source text window and then use **Debug File** (or Command-I).*

1. Bring the Project Manager to the front (Figure 12-11).
2. Click the name of a file.



✓	Name		Code
◇	Bullseye Resources.rsrc		0
▷	**Libraries**		**23324**
◇	objBullMenus.cp	◆	2208
◇	ObjectBullseye.cp	◆	944
◇	TBullWindow.cp	◆	968
◇	TWindow.cp	◆	2464
	Totals		**29908**

Figure 12-11 Click the name of a file

3. Choose **Debug File** from the Symantec Project Manager's **Project** menu (Figure 12-12).

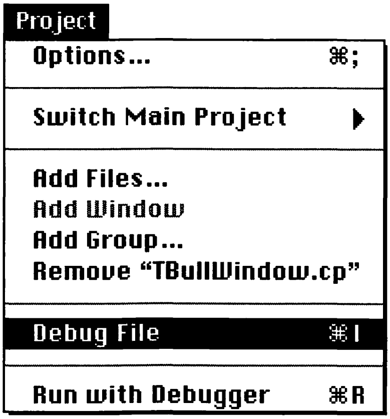
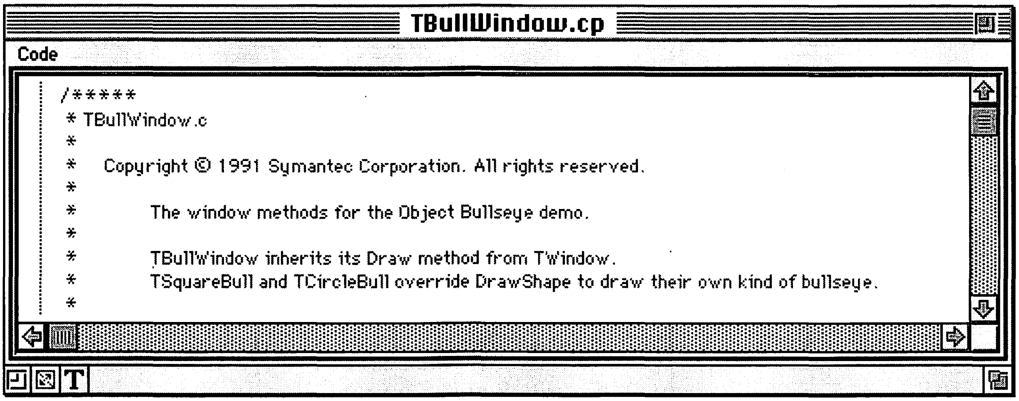


Figure 12-12 Debug File command

The file that you chose appears in a Code pane of a new Debugger window.



```
/*****
 * TBullWindow.c
 *
 *   Copyright © 1991 Symantec Corporation. All rights reserved.
 *
 *        The window methods for the Object Bullseye demo.
 *
 *        TBullWindow inherits its Draw method from TWindow.
 *        TSquareBull and TCircleBull override DrawShape to draw their own kind of bullseye.
 *
```

Figure 12-13 Viewing another file in the Project window

4. Examine the file and set breakpoints in it.

Once you have set breakpoints, you may close this new window. When you run the program, the Debugger stops at your breakpoint and displays it in the Code pane of the Main debugging window.

Examining and Setting Variables

Tracing your program's execution lets you see what your program is doing. But to really fix bugs, you need to be able to examine the variables. You use the Data window for this task or expand the stack frame.

Before you begin:

1. Quit the current debugging session by choosing **Quit** from the Debugger's **File** menu.

2. Choose **Run with Debugger** to begin a new session.

Looking at the Data window

The Data window appears to the right of the Code pane. If it is hidden, you can select it by choosing **Data** from the **Windows** menu.

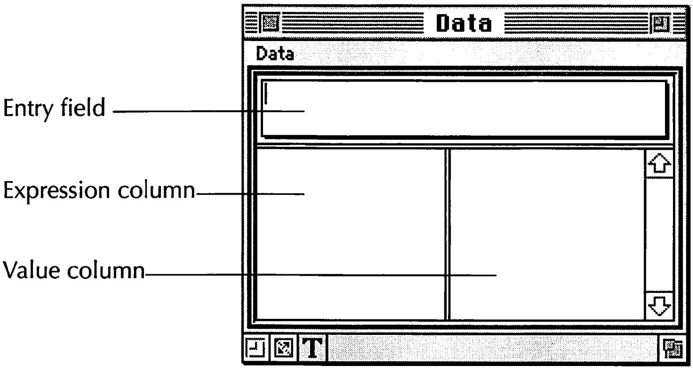


Figure 12-14 Debugger's Data window

Expressions you type into the entry field appear in the left column when you press Return or Enter. (Pressing Enter leaves the expression selected. Pressing Return leaves the entry field empty so you can type the next expression.)

You can drag the horizontal or vertical bars to make a subpane larger or smaller.

To remove an expression from the Data window, select it and choose **Clear** from the **Edit** menu or press Clear.

Examining variables

Suppose you want to watch the value of the `menuID` variable in the `HandleMenu()` function. To do so:

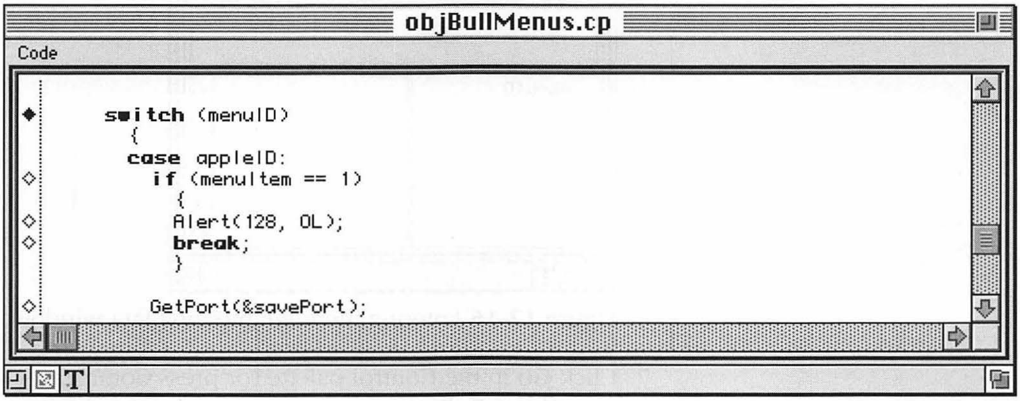
1. Make sure the `objBullMenus.cp` file is displayed in the Code pane.

   If it is not, bring the Project window to the front, click the name `objBullMenus.cp`, and choose **Debug File** from the **Project** menu.

2. Scroll down until you see the `HandleMenu()` function. Alternatively, choose **Go To Marker** from the **Source** menu and select `HandleMenu()` from the **Markers** dialog box.

3. Set a breakpoint at the `switch` statement (Figure 12-15).

   Remember that you can set breakpoints even while your program is running.



```
◆    switch (menuID)
       {
       case appleID:
◇        if (menuItem == 1)
           {
◇          Alert(128, 0L);
◇          break;
           }

◇        GetPort(&savePort);
```

Figure 12-15 Selecting the context for the Debugger

4. Click once on the line that contains the `switch` statement to select it.

   You select a line to give the Debugger a context for evaluating `menuID`. In this case, you are saying you want to know the value of `menuID` right before the `switch` statement.

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with dynamic storage—in other words, if it refers to nonstatic variables local to a function. All other expressions have global scope.

5. Click the Data window.

   You will see the insertion point blinking in the entry field.

6. Type `menuID` in the entry field and press Return (Figure 12-16).

   The Debugger compiles the expression (it may take a moment) in the context of the selected line. Right now, the Data window does not show a value for `menuID` because the program is not stopped there.

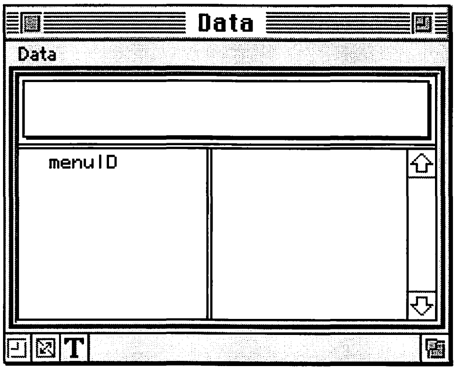
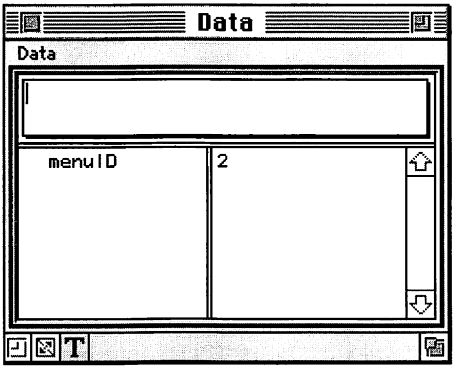


Figure 12-16 Entering menuID into the Data window

7. Click **Go** in the Control palette (or press Command-G) to run Object Bullseye.

8. Create a new window by choosing **New Triangle** from the **File** menu.

Your program stops at the breakpoint when you release the mouse button, and the value of `menuID` appears in the value column (Figure 12-17).



menuID	2

Figure 12-17 Examining the value of menuID

Any time your program stops, the Debugger displays the values of expressions that have global scope. It displays the values of expressions with local scope whose context is the same as the current function and it clears the values of local expressions whose context is not the current function.

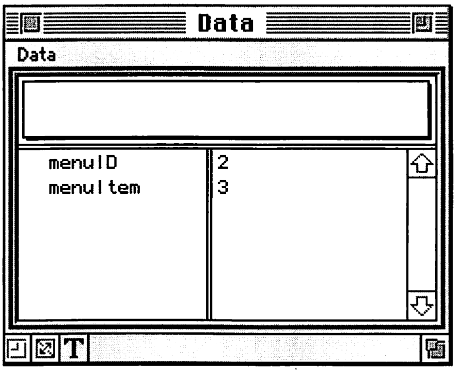
Changing the value of a variable

You can also use the Data window to change the value of a variable, as follows:

1. Click in the Data window again and type `menuItem`.

   This variable contains the number of the selected menu item.

2. Press Return to have the Debugger show you its value (Figure 12-18).

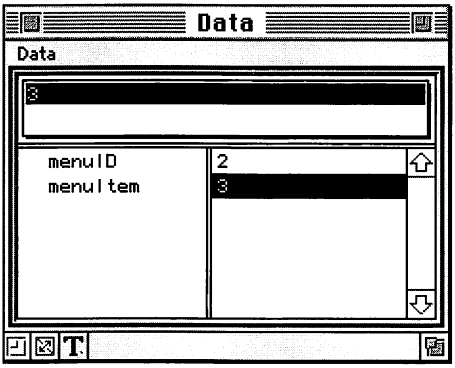


menuID 2
menuItem 3

Figure 12-18 Entering menuItem in the Data window

To change the value of a variable, click its value and type a new one in the entry field. When you click Enter, the value of the variable changes. Here's an example:

1. Click the value of `menuItem` (the right column) to select it. Its value, 3, appears in the entry field as well. Now type `2` as a new value for `menuItem`.



3

menuID 2
menuItem 3

Figure 12-19 Changing the value of menuItem

2. Click Enter to assign the new value to the variable.

3. Click **Go** to have the Object Bullseye program resume, behaving as if you had chosen **New Square** from the **File** menu.

Note

You can enter the same expression more than once in the Data window. You might want to do this to lock one of the expressions so you can compare it to the same expression later in the program. See the section "How and when the source Debugger evaluates expressions" later in this chapter.

4. Click in a Debugger window to switch back to the Debugger.

5. Now choose **Clear All Breakpoints** from the **Source** menu to make sure no breakpoints are set before you go on to the next section.

6. Click **Go** to start the program running again.

Examining structs, classes, and arrays

The Data window lets you examine and modify structures, classes, and arrays, not just simple variables. When you display a structure or union in the Data window, its value appears as `struct 0x`*nnnnnnn* or `union 0x`*nnnnnnn*. Arrays appear as `[] 0x`*nnnnnnn*.

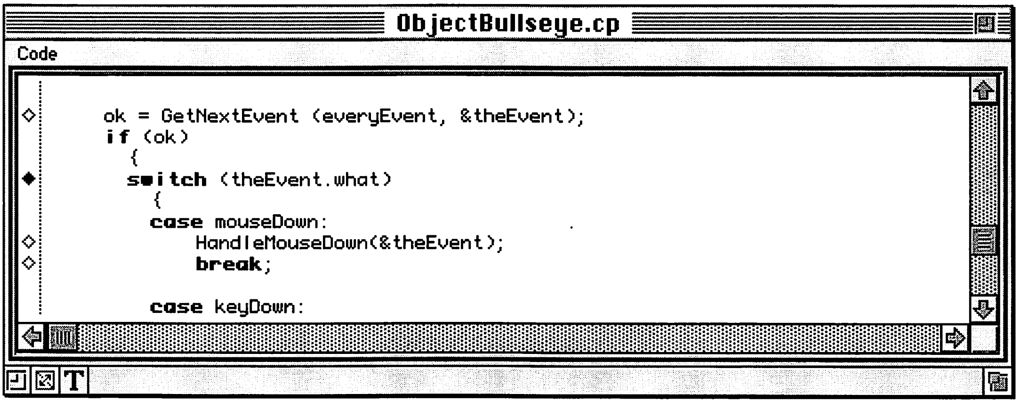
When you click on the triangle next to a structure's name, the Debugger expands the structure and displays all of its fields.

Note

Any information presented here about structures applies to unions and classes as well.

1. Make sure the Object Bullseye program is still running.

2. Display the file `ObjectBullseye.cp` in the Code pane, and set a breakpoint on the line right after the call to `GetNextEvent()` in the function `HandleEvent()` (Figure 12-20).



```
◇     ok = GetNextEvent (everyEvent, &theEvent);
      if (ok)
        {
◆       switch (theEvent.what)
          {
          case mouseDown:
◇             HandleMouseDown(&theEvent);
◇             break;

          case keyDown:
```

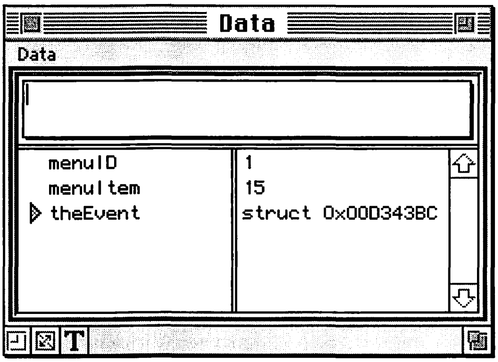
Figure 12-20 Setting a breakpoint in HandleEvent()

3. Click the Object Bullseye window.

   The program stops at the breakpoint.

4. Type `theEvent` in the entry field of the Data window and press Return. Alternatively, select `theEvent` in the Code pane and then choose **Copy to Data** from the **Edit** menu.

The Debugger displays the word `struct` and the address of the structure (Figure 12-21). If you cannot see the entire value, click the center separator bar and drag it to the left. Alternatively, you can make the window larger.



```
  menuID          1
  menuItem        15
▷ theEvent        struct 0x00D343BC
```

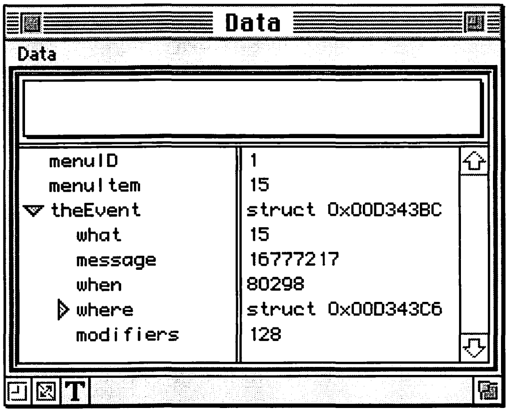
Figure 12-21 Entering theEvent in the Data window

Note

If you do not select a line to give a variable a context, the Debugger uses the current statement as the control.

---

5. Click the triangle to the left of `theEvent`. The Debugger expands `theEvent` record structure and displays all of its fields (Figure 12-22).



```
  menuID          1
  menuItem        15
▽ theEvent        struct 0x00D343BC
    what          15
    message       16777217
    when          80298
  ▷ where         struct 0x00D343C6
    modifiers     128
```

Figure 12-22 Looking at fields in a structure

You can edit the values of the fields, but you cannot edit the names.

The `what` field indicates that you're looking at a Resume event (`what = 15`).

6. Click **Go** once more and the event will be an Update event (`what = 6`).

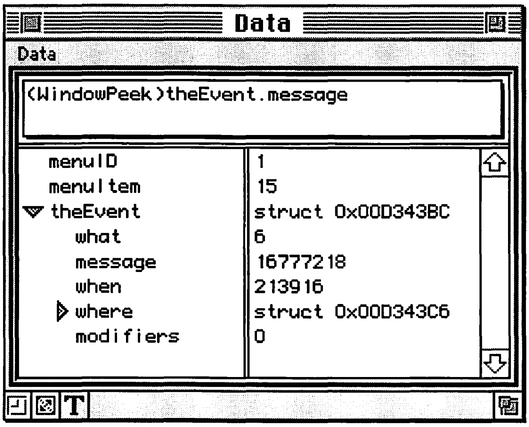
   In Update events, the `message` field points to the window record that gets the Activate event.

7. Double-click the `message` field in the Data window.

   The Debugger enters a new expression in the main Data window: `theEvent.message`. Edit the expression to read `(WindowPeek)theEvent.message` so you can look at the Window Record (Figure 12-23).

Note

Double-clicking any item in the left column of the Data window creates a new entry in the Data window.



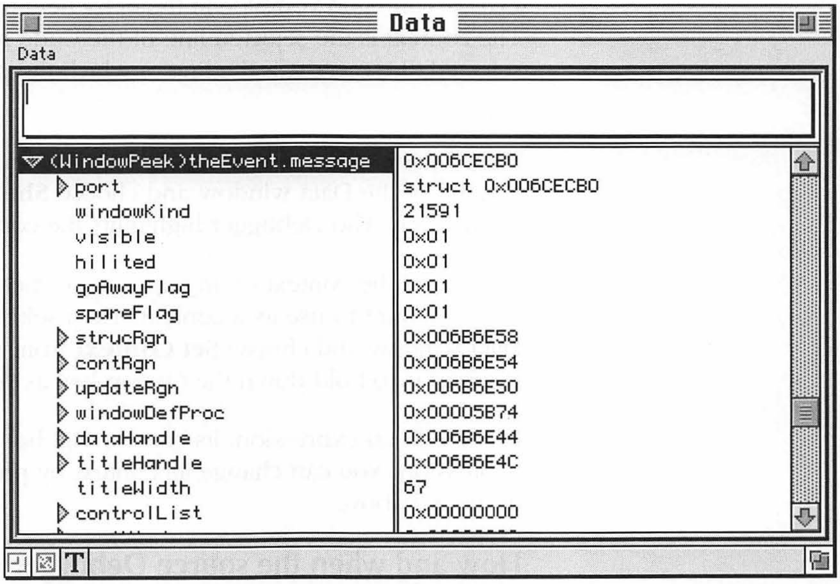
```
(WindowPeek)theEvent.message

menuID         1
menuItem       15
▼ theEvent     struct 0x00D343BC
   what        6
   message     16777218
   when        213916
 ▷ where       struct 0x00D343C6
   modifiers   0
```

Figure 12-23 Entering (WindowPeek)theEvent.message

8. Press Return so that the structure is evaluated and added to the list of expressions in the Data window.

9. Now click on its triangle to expand the structure.

   You will need to scroll to see the entire structure, since the window record is too large to fit in the window (Figure 12-24).



```
▽ (WindowPeek)theEvent.message   0x006CECB0
   ▷ port                        struct 0x006CECB0
     windowKind                  21591
     visible                     0x01
     hilited                     0x01
     goAwayFlag                  0x01
     spareFlag                   0x01
   ▷ strucRgn                    0x006B6E58
   ▷ contRgn                     0x006B6E54
   ▷ updateRgn                   0x006B6E50
   ▷ windowDefProc               0x00005B74
   ▷ dataHandle                  0x006B6E44
   ▷ titleHandle                 0x006B6E4C
     titleWidth                  67
   ▷ controlList                 0x00000000
```

Figure 12-24 Examining the fields of (WindowPeek)theEvent.message

   Scroll down to the `titlehandle` field and click on its triangle.

When you double-click the value of a pointer variable, the Debugger inserts a dereferenced expression in the Data window and displays its value. To see a pointer as an array, change its format to Address, as explained in the next section, "Expressions and Contexts."

C and C++ compilers do not enforce array bounds. If the array was declared with array bounds, those bounds are used. Otherwise, the Debugger uses default array bounds of ten elements. You can change array bounds by using **Set Array Bounds** on the **Data** menu.

Expressions and Contexts

You can enter any expression that does not have a potential side effect. That means you cannot enter assignment statements or expressions involving ++, --, or +=.

Every expression you type in the entry field is compiled in a context. The context is the selected line of the Code pane. If no line is selected, the context is the line to which the current statement arrow points.

To see the context of an expression, click the expression in the left column of the Data window and choose **Show Context** from the **Data** menu. The Debugger highlights the context in the Code pane.

To change the context of an expression, click the Code pane at the line you want to use as a context. Then select an expression in the Data window and choose **Set Context** from the **Data** menu. A shortcut is to hold down the Option key as you press Enter.

If you edit an expression, its context will be that of the original expression. You can change its context by pressing Option-Enter, as described above.

How and when the source Debugger evaluates expressions

Every time your program stops, the Debugger evaluates the expressions in the Data window. It displays the values for expressions with both global and local scope. The values of expressions that do not have a global or local context are cleared to make the window less cluttered.

If you want to make sure that the Debugger does not clear or re-evaluate a value, select it and choose **Locked** from the **Data** menu. A small lock icon appears next to the expression. This command is useful if you want to compare the value of the same expression at different times. You can also lock expressions to keep their values from being cleared when they go out of scope.

Display formats

The way the Debugger displays an expression depends on the expression's type. You can change the format with the formatting commands in the **Data** menu. To change a format, select an expression from the Data window. Then choose the format from the Debugger's **Data** menu.

Not all formats are available for all types, as shown in Table 12-1. Defaults are in italics.

Table 12-1 Display formats available

Type	Formats Available
integers	*Signed Decimal*, Unsigned Decimal, Hexadecimal, Character
unsigned	*Unsigned Decimal*, Signed Decimal, Hexadecimal, Character
pointers	*Pointer*, Address, Hexadecimal, C String, Pascal String
arrays	*Address*, C String, Pascal String
structures	Address
unions	Address
classes	Address
functions	Address
floating point	Floating Point
fixed point	Fixed Point

The display formats look like this:

Table 12-2 Display format examples

Format	Examples
signed decimal	`4523345, -1`
unsigned decimal	`4523345, 65535`
hex	`0xA09E1487`
char	`'c', 'TEXT'`

Table 12-2 Display format examples *(Continued)*

Format	Examples
C string	`"abcdef\nghi\33"`
Pascal string	`"\pabcdef\nghi\33"`
pointer	`0x007A7000`

address	`[] 0x0009FE44,`
	`struct 0x0008FC14`

floating point	`1961.0102`
fixed point	`1961.0102`

The C string and Pascal string formats display nonprinting characters in backslash form. Whenever it can, the Debugger uses the built-in escape characters (`\n`, `\r`, `\a`); otherwise, it uses `\nn`, where `nn` is an octal value.

Of course, you can use typecasting to use formats that are not normally available. For example, if you really wanted to see the integer `i` as a C string, you would type this expression: `(char *)i`.

To see any pointer as an array, just change its format to Address. Then, when you use the triangle to expand it, you see an array of elements instead of just the value of what the pointer points to.

Quitting the Debugger

To quit the Debugger, quit your application. If for some reason you cannot use your application's **Quit** command, choose **ExitToShell** from the Debugger's **Debug** menu. However, do not do this routinely, because **ExitToShell** can by-pass clean-up operations in some applications.

Tutorial: Vector ◆ 13

This tutorial highlights some of the unique features of Symantec C++. Vector is a small application that uses C++ templates to implement vectors and a sorting function. Vector displays how to:

- Debug inline functions
- Use, debug, and instantiate templates
- Use and debug instantiation files

During this tutorial, you will use templates with Symantec C++, but no effort is made to provide a general reference on the subject. For more information on templates, see *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup and *The C++ Programming Language, Second Edition* by Bjarne Stroustrup.

What You Should Know

Before you try this tutorial, you should know how to use the Symantec Project Manager and the Symantec Debugger.

How to Proceed

Before you proceed with this tutorial, run the Vector project without the Debugger to familiarize yourself with the way the application works. If **Run with Debugger** is displayed on the **Project** menu instead of **Run**, hold down the Option key to toggle to **Run,** then choose **Run**.

When you have observed Vector in action, choose **Run with Debugger** from the **Project** menu. If **Run** is displayed on the **Project** menu instead of **Run with Debugger**:

1. Set the option Run with Debugger on (found on the Project Options page of the **Project Options** dialog box).

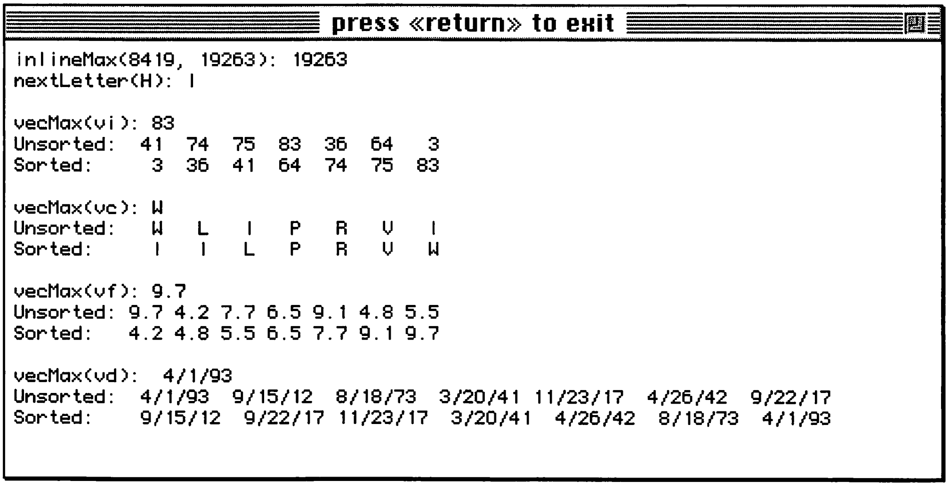
2. With the Debugger running, choose **Preferences** from the Debugger's **Edit** menu.

3. On the Debugger's **Preferences** dialog box, set the Save Expressions and Breakpoints preference off.

   Because you will be running the project several times, you will find it more convenient if you do not have breakpoints left over from previous runs.

About the Vector Project

The Vector project is an application that uses inline functions and templates to demonstrate how these features work in Symantec C++. The project uses templates to implement a vector (array) class of any type, finds the maximum value in a vector, and sorts any kind of vector. Vector displays its output, including the results of two inline functions, on the console (Figure 13-1).



```
inlineMax(8419, 19263): 19263
nextLetter(H): I

vecMax(vi): 83
Unsorted:  41  74  75  83  36  64   3
Sorted:     3  36  41  64  74  75  83

vecMax(vc): W
Unsorted:   W   L   I   P   R   V   I
Sorted:     I   I   L   P   R   V   W

vecMax(vf): 9.7
Unsorted: 9.7 4.2 7.7 6.5 9.1 4.8 5.5
Sorted:   4.2 4.8 5.5 6.5 7.7 9.1 9.7

vecMax(vd):  4/1/93
Unsorted:  4/1/93  9/15/12  8/18/73  3/20/41 11/23/17  4/26/42  9/22/17
Sorted:    9/15/12  9/22/17 11/23/17  3/20/41  4/26/42  8/18/73  4/1/93
```

Figure 13-1 Vector's console window

In this tutorial, you run the Vector application several times. After it runs, notice that the title of the window changes from "console" to "press «return» to exit." To exit Vector, press Return, or choose **Quit** from the application's **File** menu when the console window is active. If the Debugger is active, you can choose **Quit** from the Debugger's **File** menu or **ExitToShell** from the Debugger's **Debug** menu.

Most of the procedures in this tutorial start from the file `main.cp`, so you may want to look at that file before you continue.

Debugging Inline Functions

To cut down on the overhead of function calls for small functions, C++ allows you to specify "inline" functions. Instead of generating code for a function call, Symantec C++ generates the code for the function in which the function call would otherwise be displayed.

In C++ there are two ways to declare inline functions. One way, for global functions, is to use the `inline` specifier. The second way is to provide the definition (not just a declaration) for a member function in a class declaration.

In Vector, the function `inlineMax()` in the file `main.cp` and the function `nextLetter()` in the file `next.h` are global inline functions. In the template class `vector` in the file `vector.h`, the constructor, destructor, and member function `size()` are inline functions because the definitions are provided in the class declaration.

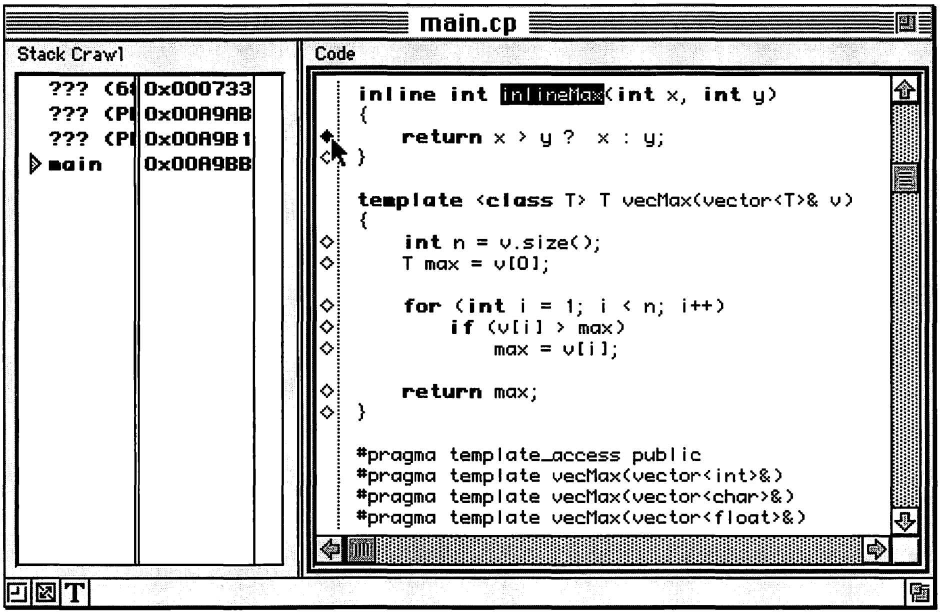
Although inline functions behave syntactically like normal functions, you cannot debug them because the compiler does not generate a function call for them. That means you cannot set a breakpoint in an inline function and you cannot step into an inline function.

To debug inline functions, turn on the Use Function Calls for Inlines option on the Debugging subpage of the PowerPC C++ Options page of the **Project Options** dialog box. When this option is set on, Symantec C++ generates normal function calls for inline functions.

Note

The inline specifier is an indicator to the compiler that it should try to generate the code directly instead of creating a function. Just as not every variable declared `register` necessarily ends up in a register, a function declared `inline` may not actually be an inline function.

1. If the function is in a source file, set a breakpoint in the inline function as you would for any other function. Choose **Run with Debugger** from the **Project** menu. When the Main debugger window is displayed, Command-click the title bar to open the **Markers** pop-up menu, then choose `inlineMax` . The Code pane scrolls to the `inlineMax()` function and you can then set a breakpoint by clicking a diamond (Command-B), as shown in Figure 13-2.



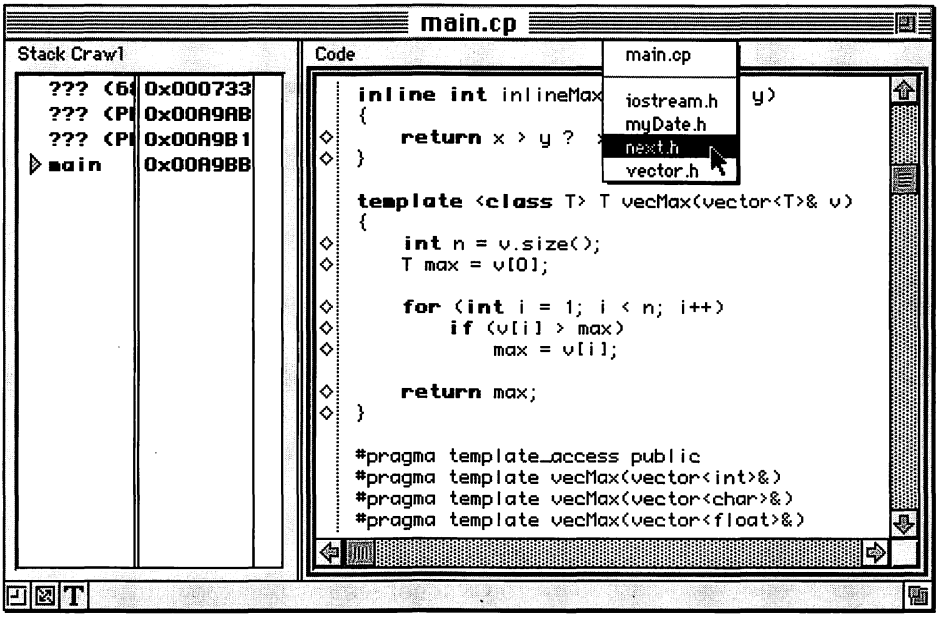
```
??? (6   0x000733
??? (P   0x00A9AB
??? (P   0x00A9B1
main     0x00A9BB
```

```
inline int inlineMax(int x, int y)
{
    return x > y ?  x : y;
}

template <class T> T vecMax(vector<T>& v)
{
    int n = v.size();
    T max = v[0];

    for (int i = 1; i < n; i++)
        if (v[i] > max)
            max = v[i];

    return max;
}

#pragma template_access public
#pragma template vecMax(vector<int>&)
#pragma template vecMax(vector<char>&)
#pragma template vecMax(vector<float>&)
```

Figure 13-2 Setting a breakpoint in inlineMax()

2. If the inline function is in a header file, Option-click the window's title bar to open the **Headers** pop-up menu and choose the header file that contains the function you want.

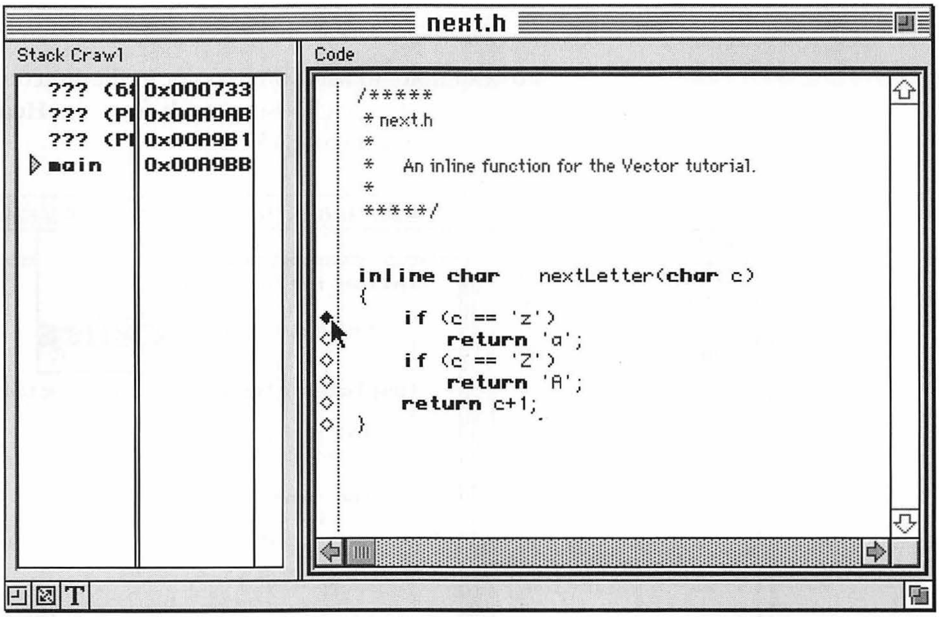
   For example, to set a breakpoint in the function `nextLetter()`, choose **next.h** from the **Headers** pop-up menu (Figure 13-3).



```
inline int inlineMax             y)
{
    return x > y ?
}

template <class T> T vecMax(vector<T>& v)
{
    int n = v.size();
    T max = v[0];

    for (int i = 1; i < n; i++)
        if (v[i] > max)
            max = v[i];

    return max;
}

#pragma template_access public
#pragma template vecMax(vector<int>&)
#pragma template vecMax(vector<char>&)
#pragma template vecMax(vector<float>&)
```

Figure 13-3 Opening a header file in the Code pane

The header file is displayed in the Code pane. You can now set a breakpoint in the `nextLetter()` function (Figure 13-4).



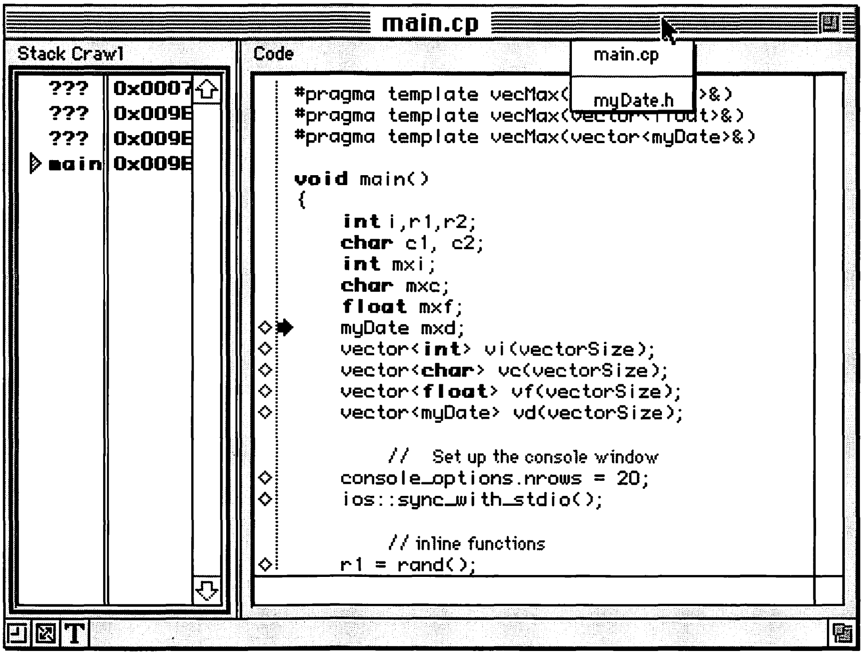
```
??? (6( 0x000733
??? (PI 0x00A9AB
??? (PI 0x00A9B1
main    0x00A9BB
```

```
/*****
 * next.h
 *
 *   An inline function for the Vector tutorial.
 *
 *****/


inline char    nextLetter(char c)
{
    if (c == 'z')
        return 'a';
    if (c == 'Z')
        return 'A';
    return c+1;
}
```

Figure 13-4 Setting a breakpoint in nextLetter()

Note

If the Use Function Calls for Inlines option is set off, you will not see any breakpoint diamonds next to the `inlineMax()` function, and the `next.h` file isn't available in the pop-up menu because it does not generate any code. This is illustrated in Figure 13-5.



```
??? 	0x0007
??? 	0x009E
??? 	0x009E
main	0x009E
```

```
#pragma template vecMax(          >&)
#pragma template vecMax(vector<float>&)
#pragma template vecMax(vector<myDate>&)

void main()
{
	int i,r1,r2;
	char c1, c2;
	int mxi;
	char mxc;
	float mxf;
	myDate mxd;
	vector<int> vi(vectorSize);
	vector<char> vc(vectorSize);
	vector<float> vf(vectorSize);
	vector<myDate> vd(vectorSize);

		//	Set up the console window
	console_options.nrows = 20;
	ios::sync_with_stdio();

		// inline functions
	r1 = rand();
```

Figure 13-5 Pane with Setting Function Calls for Inlines option off

Using and Debugging Templates

The template mechanism in C++ lets you define "container classes" and "generic functions" without giving up type checking. Vector provides examples of both a container class (the vector class) and a generic function (the sorting routine).

As the names imply, template functions and template classes are not actual functions and classes. Rather, they are schematics for building real functions and classes for types that you specify. This section introduces some useful techniques for using and debugging templates in Symantec C++.

Instantiating templates

The file `instance.cp`*, which is located in the same folder as the Vector project, contains the code from this section, so you can experiment with it.*

The compiler never generates code for template definitions. Consider this trivial function template:

```
template<class T> T sq(T v)
{
   return v * v;
}
```

Symantec C++ compiles the template, but it does not generate code for it. To generate code, you need to instantiate the function. There are two ways to do this: By using the template itself or through a #pragma directive.

You can instantiate a template function by using it as follows:

```
void byUse()
{
   int i = 5;
   float f = 3.14;

   i = sq(i);
   f = sq(f);
}
```

In this example, the compiler instantiates two versions of the function sq(). One version takes an int argument and the other takes a float argument.

You use the #pragma directive as follows:

```
#pragma template sq(int)
#pragma template sq(float)
```

These two directives instantiate the int and float versions of the function sq().

Instantiation is similar for class templates. The only difference is the syntax of the #pragma directive:

```
#pragma template tVector<int> // instantiate an
                              // int version of
                              // tVector
```

This tutorial shows you two ways to use templates. The first, called simple templates, is a straightforward instantiation by use. The second employs instantiation files and the template pragma to instantiate specific versions of template functions.

A simple template is one that you include through a header file or that you write in your code. The advantage of this type of template is that you do not have to create special files for each instance of a template function. The disadvantage is that you cannot debug every instance of a template function or class.

Instantiation files use a more elaborate arrangement of header and source files. This produces one file in the project for each instance of a template function or class, each of which you can debug.

Templates and debugging information

Knowing how the Symantec C++ compiler generates debugging information for template functions and member functions is key to understanding the difference between simple templates and instantiation files.

For source files, Symantec C++ generates debugging information for each function or member function for which source code is available. This works well for normal functions and member functions, because there is always a one-to-one correspondence between the source code and the object code, even for overloaded functions. Consider these two functions:

```
int max(int a, int b)
{
   return a > b ? a : b;
}

char max(char a, char b)
{
   return a > b ? a : b;
}
```

The Symantec C++ compiler generates object code and debugging information for the function `max(int,int)` and for the function `max(char,char)`.

Now consider this template function:

```
template<class T> T tMax(T a, T b)
{
   return a > b ? a : b;
}

void foo(int i, int j, char x, char y)
{
   int n;
   char c;

   n = tMax(i, j)
   c = tMax(x, y);
}
```

The Symantec C++ compiler does not generate object code or debugging information for the template itself. It generates code only when the function is instantiated. In the example above, Symantec C++ generates object code for the functions `tMax(int,int)` and `tMax(char,char)`.

Because source code is available, the compiler also generates debugging information. Because the debugger requires a one-to-one correspondence between source code and object code, however, the debugging information applies to only one instance of the `tMax()` function. Unfortunately, you cannot tell the instance that has debugging information. If you set a breakpoint in `tMax()`, you might stop in `tMax(char,char)` or in `tMax(int,int)`.

Debugging simple templates

In Vector, the template function `vecMax()` in the file `main.cp` uses the simple template approach. This is how `vecMax()` appears in `main.cp`:

```
template <class T> T vecMax(vector<T>& v)
{
   int n = v.size();
   T max = v[0];

   for (int i = 1; i < n; i++)
      if (v[i] > max)
         max = v[i];

   return max;
}
```

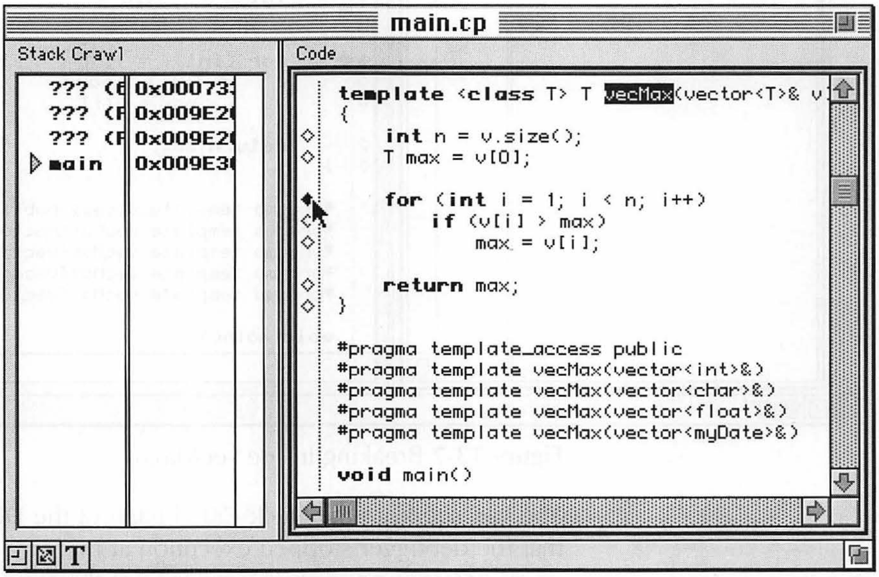
The function takes a vector of a particular type `T` as an argument and returns the largest element in the vector. This function works for any type for which the greater than operator is defined.

*You can use the **Disassemble** command in the **Build** menu to examine the code that Symantec C++ generates for each version.*

In the function `main()`, there are four calls to `vecMax()` for the built-in types `int`, `char`, and `float`, and the user-defined type `myDate`. The appearance of each call to `vecMax()` creates a new instance of the function. When you compile the file `main.cp`, Symantec C++ generates code for four different versions of `vecMax()`. As explained above, the compiler generates debugging information for only one of those instances.

To see what that looks like, make sure that the Run with Debugger option is set on, and choose **Run with Debugger** from the Project Manager's **Project** menu.

Command-click the title bar to get the Markers pop-up menu and choose `vecMax`. The code pane scrolls to the definition of `vecMax()` near the beginning of the file `main.cp`. Set a breakpoint in the template function `vecMax()`(Figure 13-6).

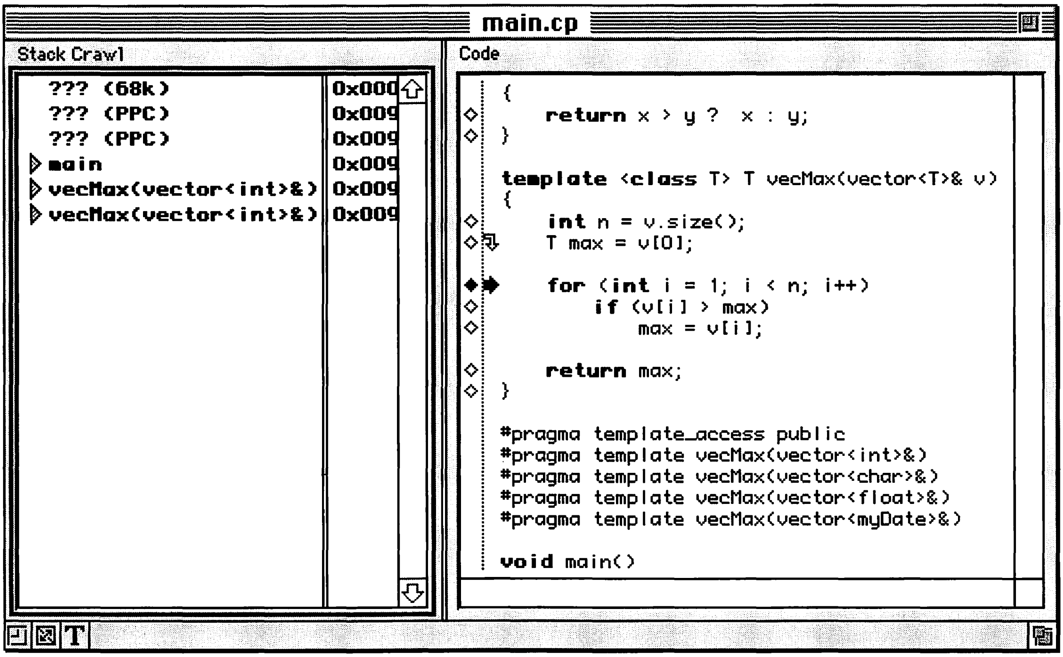


```
??? (6  0x00073
??? (F  0x009E2(
??? (F  0x009E2(
main    0x009E3(
```

```
template <class T> T vecMax(vector<T>& v)
{
    int n = v.size();
    T max = v[0];

    for (int i = 1; i < n; i++)
        if (v[i] > max)
            max = v[i];

    return max;
}

#pragma template_access public
#pragma template vecMax(vector<int>&)
#pragma template vecMax(vector<char>&)
#pragma template vecMax(vector<float>&)
#pragma template vecMax(vector<myDate>&)

void main()
```

Figure 13-6 Setting a breakpoint in vecMax()

Now click **Go** and notice where execution stops (Figure 13-7).



```
??? (68k)              0x000
??? (PPC)              0x009
??? (PPC)              0x009
main                   0x009
vecMax(vector<int>&)   0x009
vecMax(vector<int>&)   0x009
```

```
{
    return x > y ?  x : y;
}

template <class T> T vecMax(vector<T>& v)
{
    int n = v.size();
    T max = v[0];

    for (int i = 1; i < n; i++)
        if (v[i] > max)
            max = v[i];

    return max;
}

#pragma template_access public
#pragma template vecMax(vector<int>&)
#pragma template vecMax(vector<char>&)
#pragma template vecMax(vector<float>&)
#pragma template vecMax(vector<myDate>&)

void main()
```

Figure 13-7 Breaking inside vecMax()

You can see from the Stack Crawl pane of the Debugging window that the debugger stopped execution at the `vecMax(vector<int>&)` instance of the template function. Click **Go** again, and notice that the program does not stop this time.

Note

> The current version of Symantec C++ generates debugging information for the first instantiation of a template function. However, you should not rely on this behavior; it may change in the future.

As you can tell from the output in the console window, even though Symantec C++ generated debugging information for only one instance of the `vecMax()` function, it did generate code for the other three versions.

If you are writing an uncomplicated template function such as `vecMax()`, there is nothing wrong with using the simple template technique. For such functions, you probably will not need the Symantec Debugger at all. But if you are writing a more complex template function or a template class, you may need to be able to debug every instance or a particular instance of the template. To do so, you need to use the instantiation file technique.

Using template instantiation files

The template instantiation file technique forces the Symantec C++ compiler to generate debugging information for each instance of a template function or class. The Vector application uses this technique for the `vector` class and for the `selection()` function, which implements a selection sort.

To use this technique, you must use the `#pragma template-access` directive. This directive instructs the compiler how to instantiate templates and whether the templates should be public or private. There are three options:

- `#pragma template-access static`: This is the default setting. Each template is instantiated every time it is referenced. Multiple files that refer to the same template each produce an instantiation. Template functions or classes instantiated this way have local linkage (as the name `static` implies).

Note

For restrictions regarding template use and this `#pragma`, see the *Symantec C++ Compiler Guide.*

For example, in `main.cp` the `vecMax()` function could be instantiated in this manner. If this function appeared in a header file, it would be instantiated in each source file that referred to it. Each such instantiation generates a static copy of the function.

- #pragma template-access extern: This #pragma informs the compiler that no templates should be instantiated under any circumstances. Instantiation refers to the generation of code/data for a template class. The template class body is always parsed by the compiler whenever it is referenced.

  This template-access directive appears in vector.h and selection.h to inform the compiler that no code/ data should be generated for these templates. This #pragma must be used in conjunction with template-access public because otherwise undefined symbol errors would be generated for template classes and functions.

Note

Inline template functions are always inlined (if they can be), regardless of the #pragma template-access directive that is in effect.

- #pragma template-access public: This #pragma informs the compiler that any templates explicitly expanded with the #pragma template directive should be generated as externally defined symbols in this translation unit. It is used in conjunction with #pragma template-access extern when explicit instantiation files are created. vector<x>.cp uses this technique to generate the code/data for specific instances of the template class.

Using this technique, you first create a file for the template class declaration or the template function declaration. In the Vector project, the vector class declaration is in the file vector.h. Include this file in any other source file that needs the declaration of your template class. The selection() template function declaration is in the file selection.h. Include this file in any source file that calls for the template function to provide a function prototype for selection().

Next, you create an implementation file for the definitions of the template member functions and the template function. By convention, the name of this file is the name of the class or function. For the vector class, the implementation file is the file vector.cp. It contains the definitions of all the template functions that are not defined as inline functions. For the selection() function, the implementation file is the file selection.cp. As you will see, you do not add the implementation file to the project.

Instead you create an instantiation file for each instance of the template class or template function. By convention, the name of this file follows the same format as #pragma template directive. The instantiation file for the int version of the vector class is named vector<int>.cp. The instantiation file for the float version of the selection() function is named selection(float).cp. The contents of the instantiation file look like this:

```
#include "vector.cp"
#pragma template_access public
#pragma template vector<char>
```

The #include statement brings in the implementation file. The #pragma template_access public directive ensures that the scope of the instantiation of the class and its member functions is public. If you leave out this directive, the template_access extern directive in the header files prevents any templates from being expanded, and the class and its member functions are not available to other files in the project. As you read earlier, the directive #pragma template vector<char> asks the Symantec C++ for Power Macintosh compiler to instantiate a char version of the vector class.

The instantiation file for template functions looks the same. The only difference is the function syntax for #pragma template:

```
#include "selection.cp"
#pragma template_access public
#pragma template selection(vector<float>)
```

Add the instantiation file to the project.

Note

The other files in the project must be compiled with the `#pragma template_access extern` for this technique to work.

As you can see in the Vector Project window, there are four instantiation files for the `vector` class and four instantiation files for the `selection` function.

Debugging with instantiation files

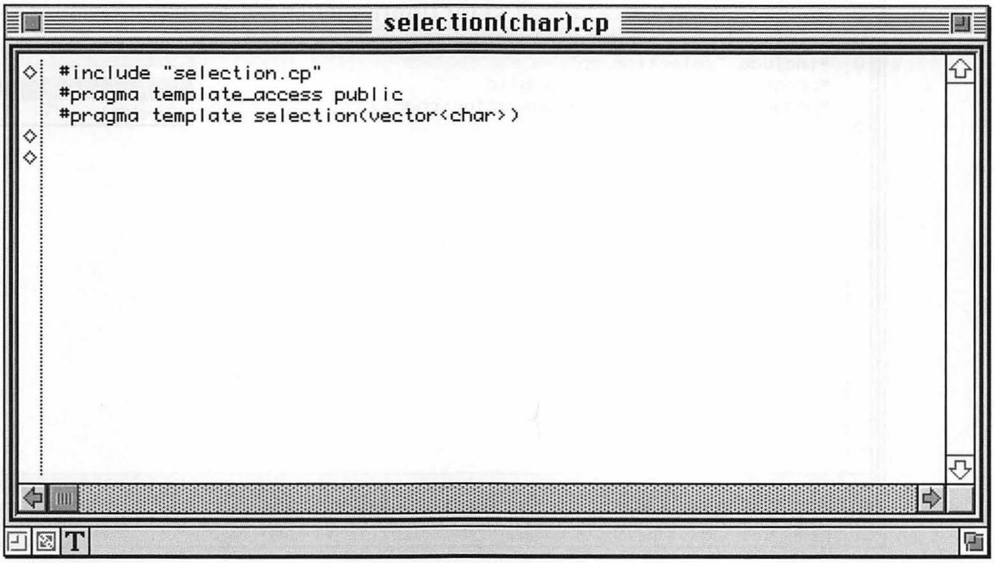
When you use the template instantiation file technique, Symantec C++ generates object code for only one instance of the template per file. When the debugging option is set on, it generates debugging information only for that instance, so the Debugger is able to maintain a one-to-one relationship between object code and source code.

Because the instantiation file in the project doesn't contain code, use a pop-up menu from a Debugger window's title bar to reach the implementation file that the instantiation file includes.

Here's how to set a breakpoint in the `char` version of the `selection()` function.

Choose **Run** or **Run from Debugger** from the Project Manager's **Project** menu to run Vector. Click the Project window to make it active or choose **Vector.π** from the Debugger's **Windows** menu.

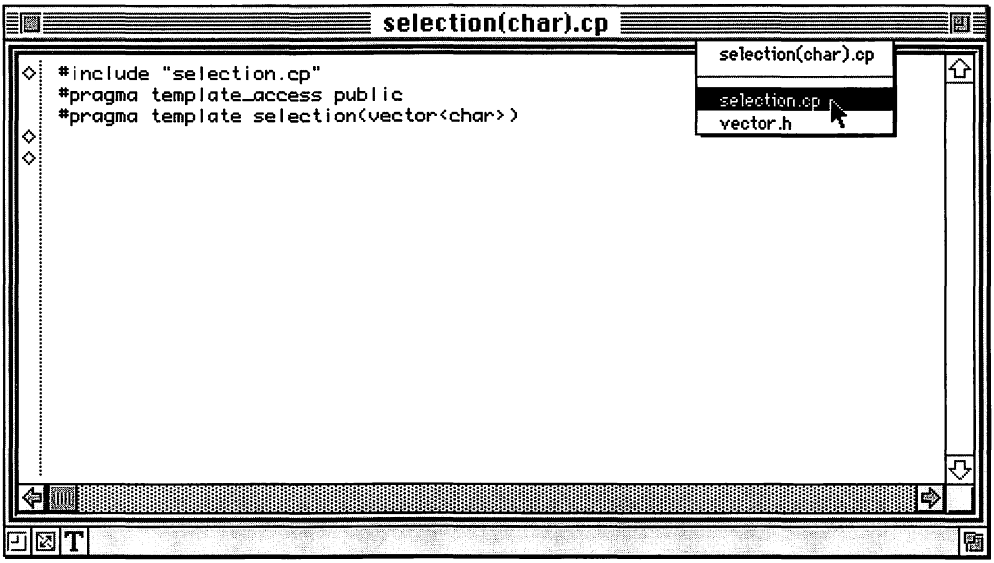
Click the filename `selection(char).cp` in the Project window, and choose **Debug File** from the Project Manager's **Project** menu. The file is displayed in the Code pane of a Debug Browser window, as shown in Figure 13-8.



```
#include "selection.cp"
#pragma template_access public
#pragma template selection(vector<char>)
```

Figure 13-8 Instantiation file in the Code pane

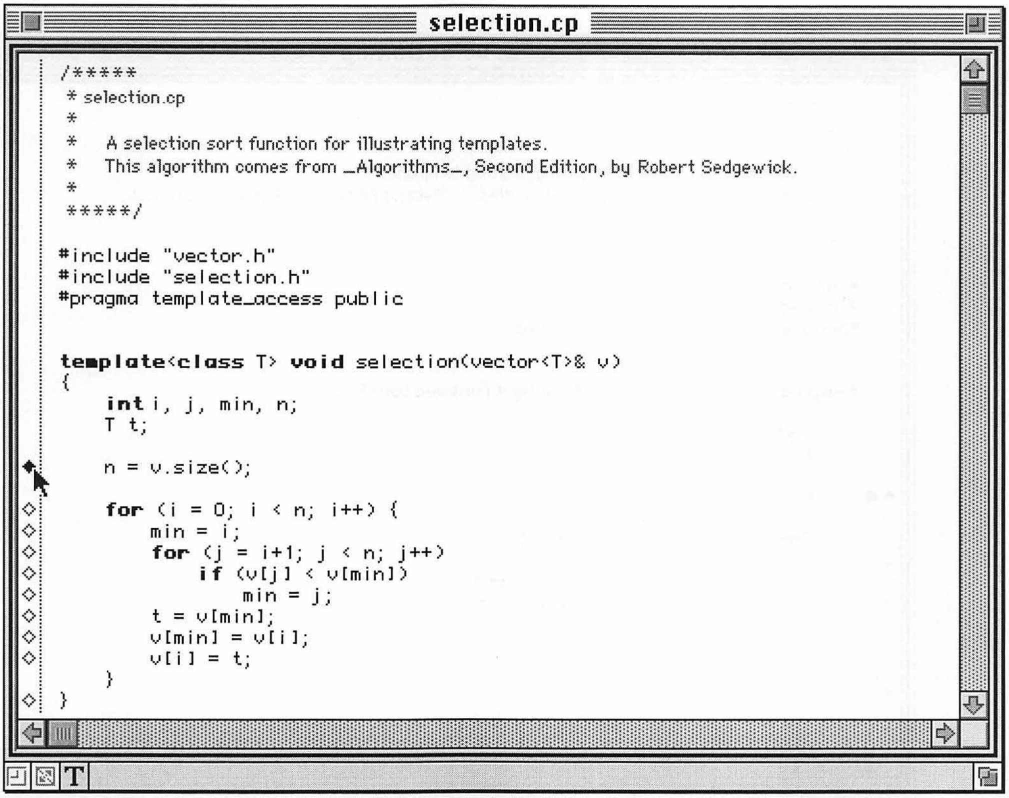
To see the code in the included implementation file `selection.cp`, Option-click the Debugger window's title bar to open the pop-up menu. Choose the implementation file from the pop-up menu, as shown in Figure 13-9.



```
#include "selection.cp"
#pragma template_access public
#pragma template selection(vector<char>)
```

Figure 13-9 Choosing the implementation file

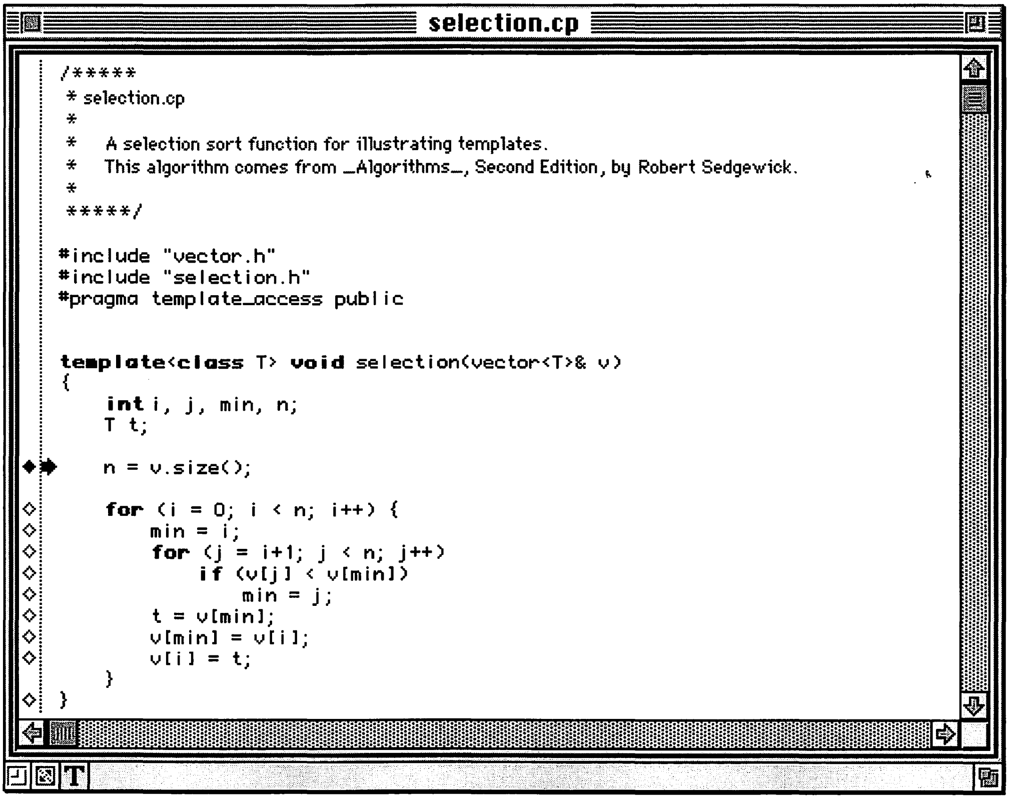
Now the implementation source code is displayed in the Code pane of a Debug Browser window. Set a breakpoint in the function `selection()`, as shown in Figure 13-10.



```
/*****
 * selection.cp
 *
 *    A selection sort function for illustrating templates.
 *    This algorithm comes from _Algorithms_, Second Edition, by Robert Sedgewick.
 *
 *****/

#include "vector.h"
#include "selection.h"
#pragma template_access public


template<class T> void selection(vector<T>& v)
{
    int i, j, min, n;
    T t;

    n = v.size();

    for (i = 0; i < n; i++) {
        min = i;
        for (j = i+1; j < n; j++)
            if (v[j] < v[min])
                min = j;
        t = v[min];
        v[min] = v[i];
        v[i] = t;
    }
}
```

Figure 13-10 Setting a breakpoint in the implementation file

Click the Control palette's **Go** button (Command-G). Note that execution does not stop in the `int` version of `selection()`. It stops only for the `char` version of `selection()`, as shown in Figure 13-11.



```
/*****
 * selection.cp
 *
 *   A selection sort function for illustrating templates.
 *   This algorithm comes from _Algorithms_, Second Edition, by Robert Sedgewick.
 *
 *****/

#include "vector.h"
#include "selection.h"
#pragma template_access public


template<class T> void selection(vector<T>& v)
{
    int i, j, min, n;
    T t;

    n = v.size();

    for (i = 0; i < n; i++) {
        min = i;
        for (j = i+1; j < n; j++)
            if (v[j] < v[min])
                min = j;
        t = v[min];
        v[min] = v[i];
        v[i] = t;
    }
}
```

Figure 13-11 Stopping in the char version of selection()

You can use the same steps to set breakpoints for other instantiations of `selection()` or to set breakpoints in the `vector` class member functions for specific types.

What to Do Next

This tutorial showed you some basic techniques for working with inline functions and templates in Symantec C++. If you are just learning C++, you might find Vector useful for trying out different things.

Create wrapping subscripts

The vector class shows how to write a subscript operator. Rewrite it so that the subscripts wrap around when they're out of range. For example, if the vector object `foo` contains ten items, the in-range subscript references are `foo[0]` to `foo[9]`. If you use any out-of-range subscript, like `foo[10]`, the current subscript operator always returns the first element in the vector, `foo[0]`. But if the subscript operator wrapped, it would return `foo[0]` for `foo[10]`, `foo[1]` for `foo[11]`, `foo[2]` for `foo[12]`, and so on.

Add new methods to myDate

- A >> operator that reads in a date from the console or a file. First, write the function to read dates written such as `05/04/95`. Then, extend it to read dates written such as `Apr. 28, 1996`. Declare the function this way:

  ```
  friend ostream& operator>>

      (ostream& s, const myDate& d);
  ```

- A > operator that returns whether one date is later than another. For example, June 30, 1995 is later than December 13, 1994. Declare the function this way:

  ```
  friend int operator>

      (const myDate& d1, const myDate& d2);
  ```

- A + operator that adds a number of days to a date and returns a new date. For example, 21 days + December 12, 1994 = January 2, 1995. You have to write two functions: one that takes the date first and the number of days second, and another that takes the number of days first and the date second. These functions need to take into account leap years and the number of days in each month. Declare one of the functions this way:

```
friend myDate operator+
    (const myDate& date,
     const int& days);
```

- A - operator that subtracts one date from another and returns a number of days. For example, March 15, 1995 - January 25, 1995 = 49 days, and July 6, 1995 - August 10, 1995 = -35 days. This function needs to take into account leap years and the number of days in each month. Declare the function this way:

```
friend int operator-
      (const myDate& d1, const myDate& d2);
```

Write a new sort function

Use the `selection()` function in the file `selection.cp` as a starting point for a different kind of sort. You can find sorting algorithms in many computer science textbooks and tutorials.

Create a new class and sort it

Write your own class; for example, classes for strings, times, or people's addresses. Create a vector from it and sort it.

Change the vecMax() function into a member function

Make the `vecMax()` function a member function of vector. Remember, the new member function won't take a vector object as an argument. Instead, call the function in this way:

```
i = myVector.max();
```

Create a template function

The code that prints out the vector maximum, the unsorted vector, and the sorted vector is repeated four times in `main()`. Write a template function that does its work.

Tutorial: Beeper 14

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.

About the Tutorial

This tutorial serves as an introduction to Visual Architect. The application you build, called Beeper, calls up a dialog box containing an edit text field. This field can be used to indicate the number of times a Macintosh beeps. The dialog box also contains a push button that starts the beeping.

Getting Started

Before you can build the user interface portion of Beeper using Visual Architect, you need to create a Beeper project. You are probably familiar with these steps from other tutorials in this manual—for example, the Hello World tutorial in Chapter 10. As a reminder, the steps for creating a project are also included here:

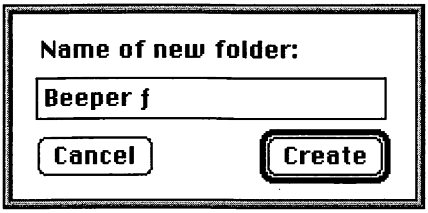
1. From the Finder, launch the Symantec Project Manager, which resides in the `Symantec C++ for Power Mac` folder.

   The **File Open** dialog box opens.

2. Choose **New Project** to open the **File Save** dialog box.

3. Navigate to a location outside the `Symantec C++ for Power Mac` folder (in other words, outside the system tree), and click New.

   The **New Folder** dialog box opens (Figure 14-1).



Name of new folder:

Beeper ƒ

Figure 14-1 New Folder dialog box

4. Type `Beeper ƒ` and click Create. (Press Option-F for ƒ.)

   You have now created a folder named `Beeper ƒ`. The **File Save** dialog box becomes frontmost again.

5. Type `Beeper.π` in the Create New Project textbox. (Press Option-P for π.)

6. Choose **VA Application** from the **Project Model** pop-up menu, and click Save.

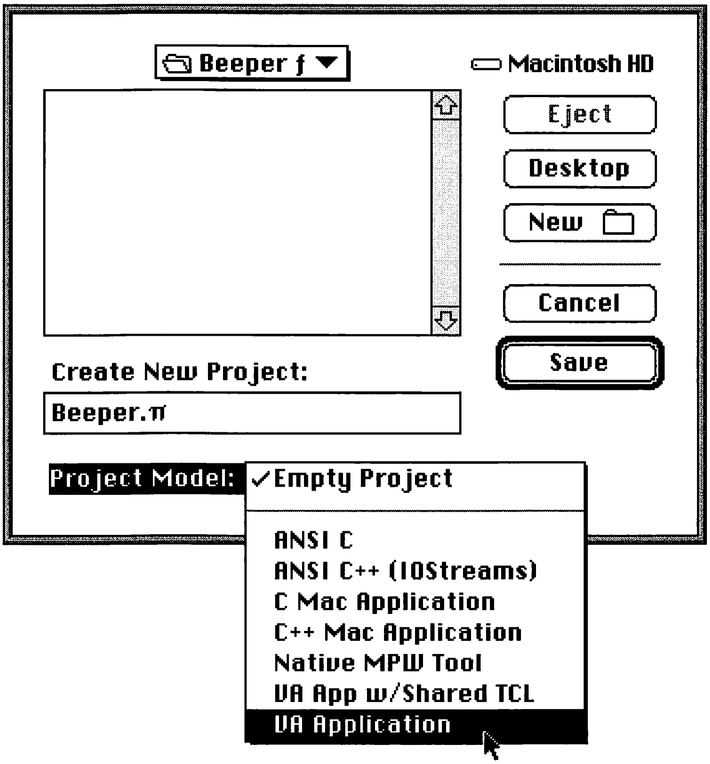


Figure 14-2 Project Model pop-up menu in the File Save dialog box

The Project file `Beeper.π` is created in the folder `Beeper ƒ`, and a window titled Beeper.π is displayed.

With the Beeper project created, you are now ready to build an application with the functionality required for this tutorial. Doing so involves the following steps:

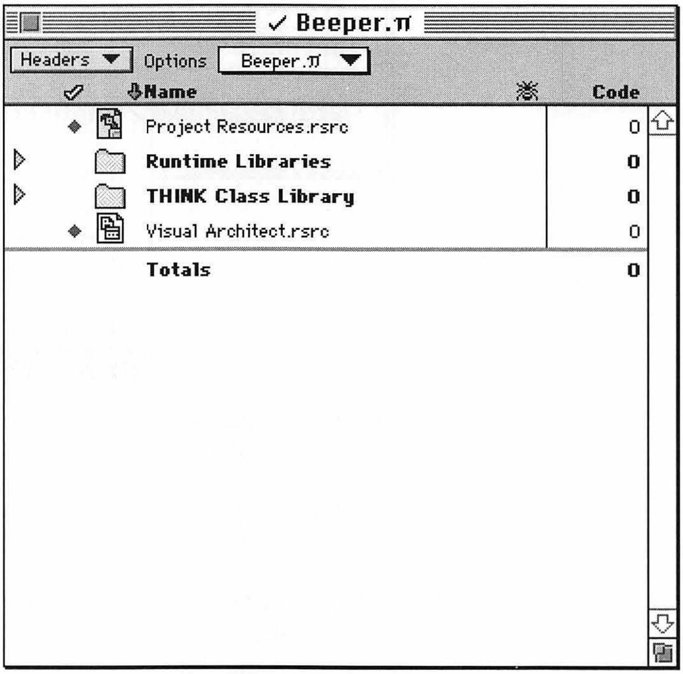
- Designing the user interface
- Generating the code and upgrading the project
- Modifying the generated code
- Updating and running the application

Designing the User Interface

To design Beeper's user interface, you must first launch Visual Architect. Using this tool, you can then create the appropriate dialog box, push buttons, and menus.

Starting Visual Architect

Visual Architect can be launched directly from the Symantec Project Manager. In the Beeper.π Project window, shown in Figure 14-3, double-click the `Visual Architect.rsrc` file to launch Visual Architect with this file.



✓	Name	Code
◆	Project Resources.rsrc	0
▷	**Runtime Libraries**	**0**
▷	**THINK Class Library**	**0**
◆	Visual Architect.rsrc	0
	Totals	**0**

Figure 14-3 Beeper.π Project window

The Views List window is displayed, showing a list of the defined views in `Visual Architect.rsrc` (Figure 14-4).

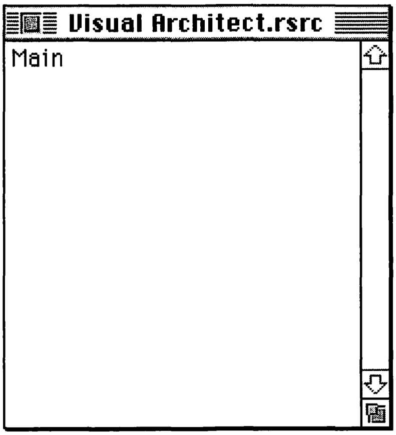
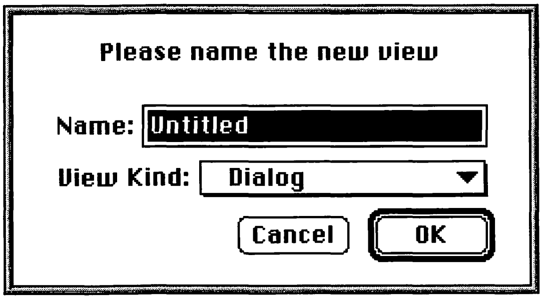


Figure 14-4 Visual Architect.rsrc Views List window

Creating a view

As you can see in Figure 14-4, Main is the only view defined in `Visual Architect.rsrc`. To add a new view:

1. Choose **New View** from the **View** menu to open the **New View** dialog box (Figure 14-5).


Please name the new view

Name: Untitled

View Kind: Dialog

Figure 14-5 New View dialog box

2. Type `BeeperDialog` in the Name textbox, choose **Modal Dialog** from the **View Kind** pop-up menu, and click OK.

   A new window titled BeeperDialog opens. This is the BeeperDialog View Edit window, in which you construct the BeeperDialog view.

Adding pane elements to the dialog box

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

1. Click the **Tools** menu and tear off the Tool palette (Figure 14-6). Put the Tool palette off to the side of the screen.

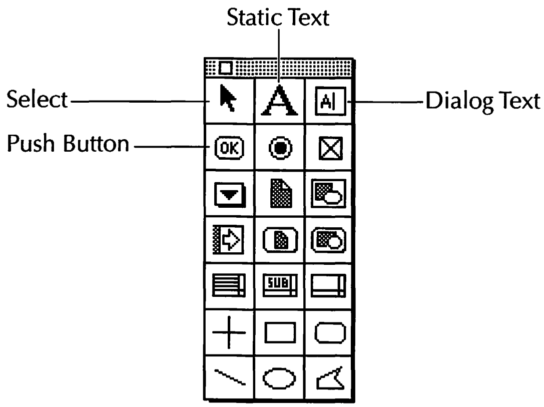


Figure 14-6 Tool palette

Note

Before you start adding panes, you may want to look ahead to the finished dialog box (Figure 14-12).

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 View Edit 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 click elsewhere in the window to end the typing task.

Note

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

Next, add a dialog text pane:

1. Choose the Dialog Text tool from the Tool palette. Note that the cursor changes to a crosshair when it is over the dialog box. Click the cursor to the right of the static text pane you just added to position the dialog text pane. An edit textbox is displayed.

2. Choose **Class** from the **Pane** menu and choose **CIntegerText** from the pop-up menu (Figure 14-7).

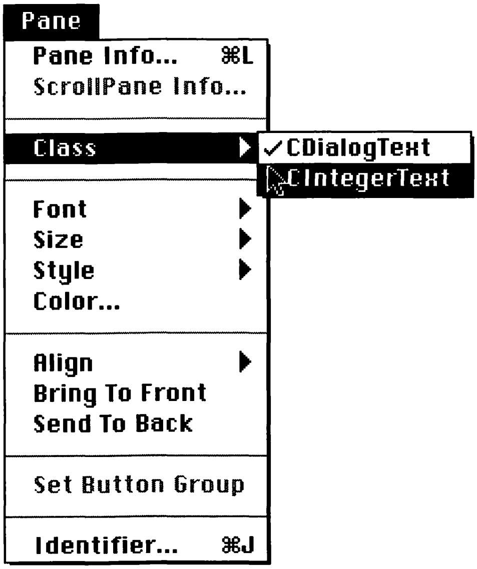


Figure 14-7 Setting the class of the dialog text pane

You can constrain the type of text the user is allowed to enter into the dialog text pane by selecting a special THINK Class Library class. In this example, you constrained the text to integers. The dialog text pane then becomes an instance of the class you selected.

The next item to add is an OK button:

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

   A push button named OK is displayed. 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. Choose the Push Button tool from the Tool palette.
2. Click to the right of the dialog text pane.

   A push button named Cancel is displayed. By default, the second push button added to a view is named Cancel.
3. Type `Beep` to rename it.

Your dialog box should now look like the one displayed in Figure 14-8.

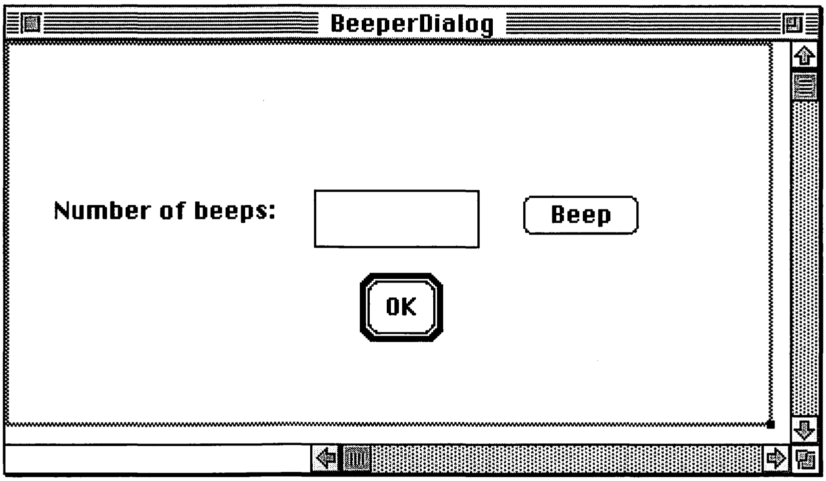


Figure 14-8 BeeperDialog dialog box

Creating a command to execute a 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:

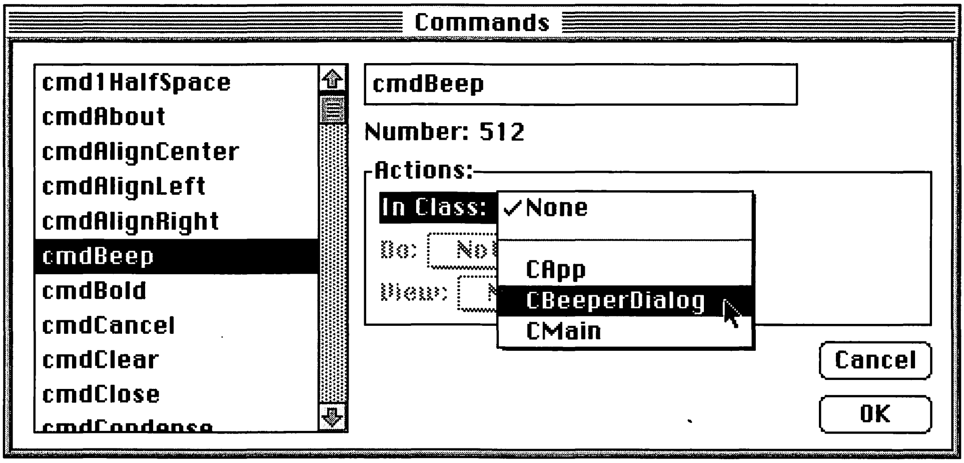
1. Choose **Commands** from the **Edit** menu.

   The **Commands** dialog box opens (see Figure 14-9), which lets you add 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 to indicate that the `cmdBeep` command should be handled by the class CBeeperDialog.



cmdBeep

Number: 512

In Class: CBeeperDialog

Figure 14-9 Using the Commands dialog box

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.

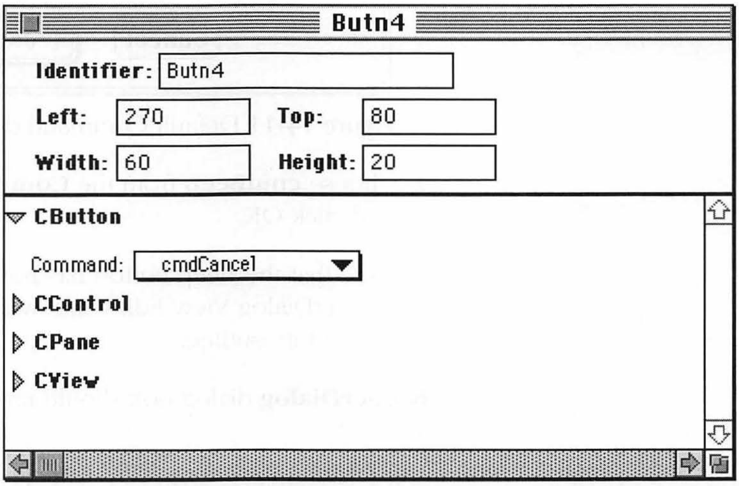
Associating a command with a button

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. You can then manipulate the values of many of the class variables in the selected pane's class as well as in its base classes up through CView.

2. Click the small triangle next to CButton to access the data members of the CButton class (Figure 14-10).



Butn4

Identifier: Butn4

Left: 270 Top: 80

Width: 60 Height: 20

CButton

Command: cmdCancel

CControl

CPane

CView

Figure 14-10 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 **BeeperDialog** 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 to open the **Default Command** dialog box (Figure 14-11). As you can see, `cmdOK` is currently defined as the default command.

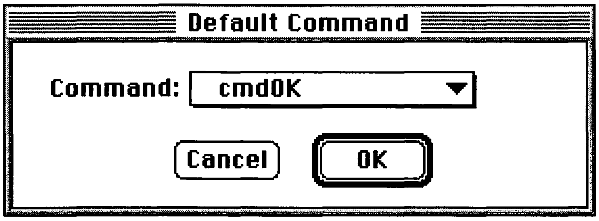


Figure 14-11 Default Command dialog box

2. 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 **BeeperDialog** dialog box should resemble the one shown in Figure 14-12.

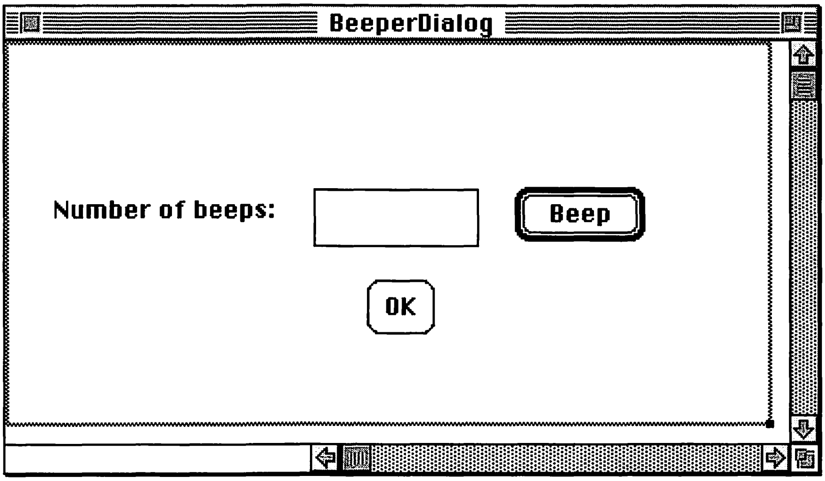


Figure 14-12 Completed BeeperDialog dialog box

3. Close the BeeperDialog View Edit window by clicking its close box. You have now finished constructing the **BeeperDialog** dialog box.

Adding a push button to the Main view

The `Visual Architect.rsrc` file has a default view called Main, as you saw when you first started Visual Architect. To add a push button to this view that opens the **BeeperDialog** 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 pane 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.

   The Main window currently is too small to display a new button without the user having to scroll. You need to increase the size of the window.

3. Drag the size box in the Main View Edit window to 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 14-13.

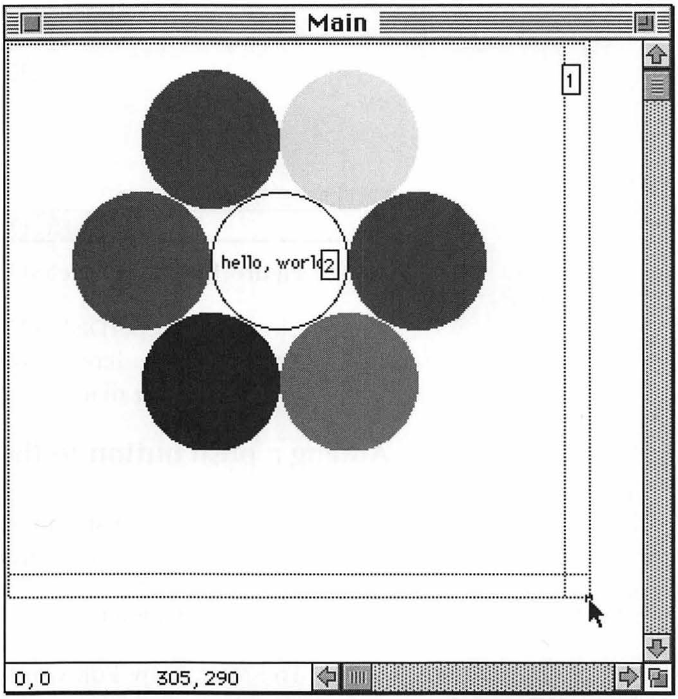


Figure 14-13 Resizing the Main View Edit window's portRect

5. Now add the push button that opens the **BeeperDialog** dialog box. Choose the Push Button tool from the Tool palette.

6. Click just below the circles to position the button and type `Beeper`... to name it, as shown in Figure 14-14.

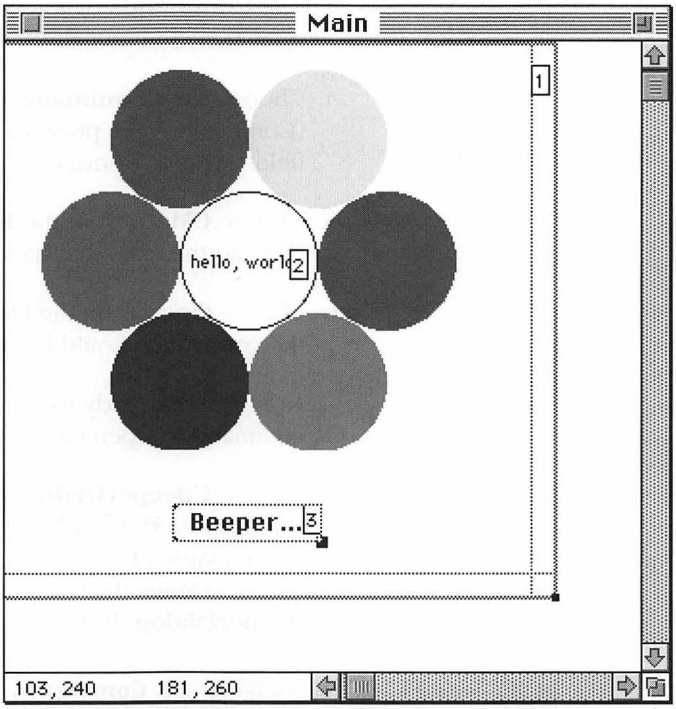


Figure 14-14 Creating the Beeper button

Creating the command to call up the dialog box

Earlier in the tutorial, you created a command to be performed when the user clicked the Beep button in the **BeeperDialog** dialog box. Now you need to create a command to call up the **BeeperDialog** dialog box. This time, however, 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 command.

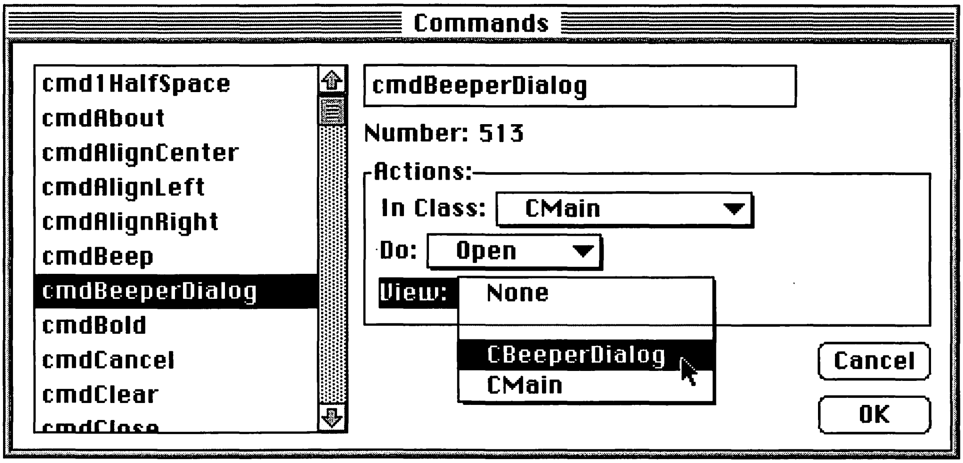
4. Choose **New Command** from the **Edit** menu (Command-K) or press Return to activate the edit text field. Type `cmdBeeperDialog` in the text field.

5. Choose **CMain** from the **In class** pop-up menu to indicate the class that handles the command.

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

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

7. Choose **CBeeperDialog** from the **View** pop-up menu (Figure 14-15), to indicate that the command should open a view of class CBeeperDialog. Visual Architect later generates the code needed to open the **BeeperDialog** dialog box.



cmdBeeperDialog

Number: 513

Actions:

In Class: CMain

Do: Open

View: CBeeperDialog

Figure 14-15 Setting the view that the cmdBeeperDialog command opens

8. Close the **Commands** dialog box 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. To preview the view you just created:

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

   The Main View Edit 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 its close box.

You should now save your work. Choose **Save** from the **File** menu (Command-S).

Generating Code and Updating the 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 **Symantec Project Manager** menu (Figure 14-16). The menu title is the Symantec Project Manager application icon.

Note

The Main window must be selected from `Visual Architect.rsrc` for this to work.

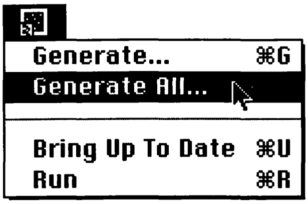
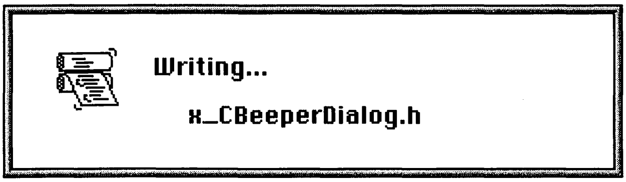


Figure 14-16 Symantec Project Manager menu

Note

Because you are using Visual Architect for the first time to generate code for this project, you need to choose **Generate All**. For subsequent code generation, choose **Generate**. Using the latter command, code is generated only for those classes that have changed since the last code generation.

2. Check the progress of the code generation in the message box that opens (Figure 14-17).



Writing...

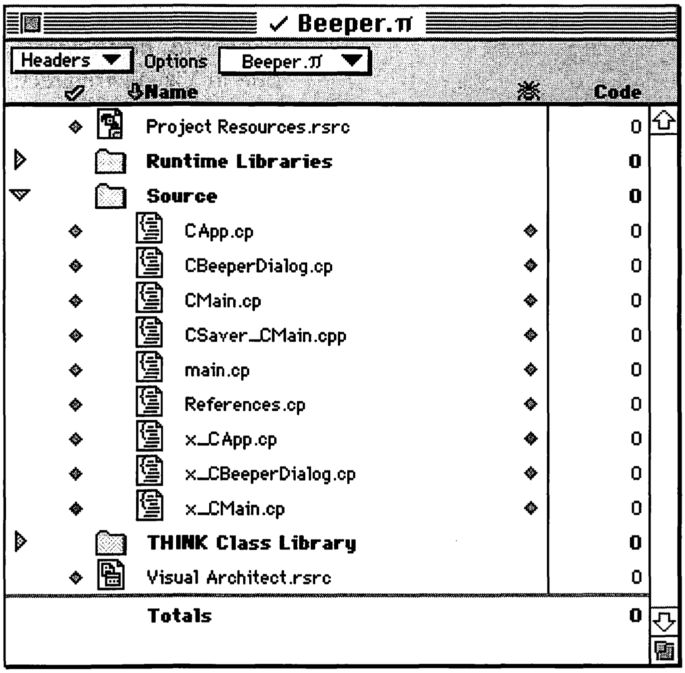
x_CBeeperDialog.h

Figure 14-17 Code Generation message box

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

3. Switch to Symantec Project Manager and open the Source group in the `Beeper.π` Project window if it is not already open. Click the small triangle to the left of the group's name.

   Visual Architect has added the new source files to your project in the Source group (Figure 14-18).



✓	Name		Code
◆	Project Resources.rsrc		0
▷	**Runtime Libraries**		**0**
▽	**Source**		**0**
◆	CApp.cp	◆	0
◆	CBeeperDialog.cp	◆	0
◆	CMain.cp	◆	0
◆	CSaver_CMain.cpp	◆	0
◆	main.cp	◆	0
◆	References.cp	◆	0
◆	x_CApp.cp	◆	0
◆	x_CBeeperDialog.cp	◆	0
◆	x_CMain.cp	◆	0
▷	**THINK Class Library**		**0**
◆	Visual Architect.rsrc		0
	Totals		**0**

Figure 14-18 Beeper.π Project window showing the Source group

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 Project window by double-clicking its name.

Note

The x_ prefix of the x_CBeeperDialog.cp filename identifies it as containing the lower-level class description for the BeeperDialog view. This file has an associated header file x_CBeeperDialog.h. These 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 to the Clipboard.

3. Open the file CBeeperDialog.cp.

Note

The CBeeperDialog.cp file has no prefix; thus it contains the upper-level class description for the BeeperDialog view. This file has an associated header file CBeeperDialog.h. Visual Architect generates these two files only once, so any hand coding in these files 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.

5. 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.

6. 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.

7. 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 the Symantec 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 opens with the Beeper button. Clicking this button makes the **BeeperDialog** dialog box open. You can enter an integer value into the edit text field and click Beep, and your Macintosh beeps that number of times.

Tutorial: Process Monitor

15

In this tutorial, you use Visual Architect to create the user interface for an application you create, then generate code for it. The application contains more user interface elements than does the Beeper application you worked with in Chapter 14. For the Process Monitor application, you need to create two main windows, a dialog box, a subview, push buttons, check boxes, scroll bars, text fields, and menus.

About the Application

The Process Monitor application displays a list of all processes currently running on your Macintosh. When you first launch the application, you are presented with a dialog that lets you select between two views: Process List and Process List and Information.

The first of these shows the list of the processes, while the second view also allows the user to

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

The Process List and Information view also includes a group of check boxes that show the current settings of the `'SIZE'` resource flags for the selected process.

Note that the key piece of information displayed by these views—the list of running processes—is the same. You do not create this component twice; instead you set it up once, then reuse it in the second view. The subview you create during the tutorial thus works as a time-saving device.

Note

Throughout this tutorial, you are asked to give names (identifiers) to various visual elements. The names you type must match, in spelling and case, those given in the tutorial. Otherwise, the pre-edited source files provided for you will not compile properly.

Getting Started

In most of the tutorials in this manual, you start out by creating the project with which you are going to work. In this tutorial, however, you are adding interface elements to an existing project. This section outlines some of the characteristics of the Process Monitor application and explains how to open the application as well as how to specify its top-level characteristics.

Looking at the project

The `PPC Process Monitor.π` project was created with the VA Application project model. The following additions were then made to facilitate your work with the application:

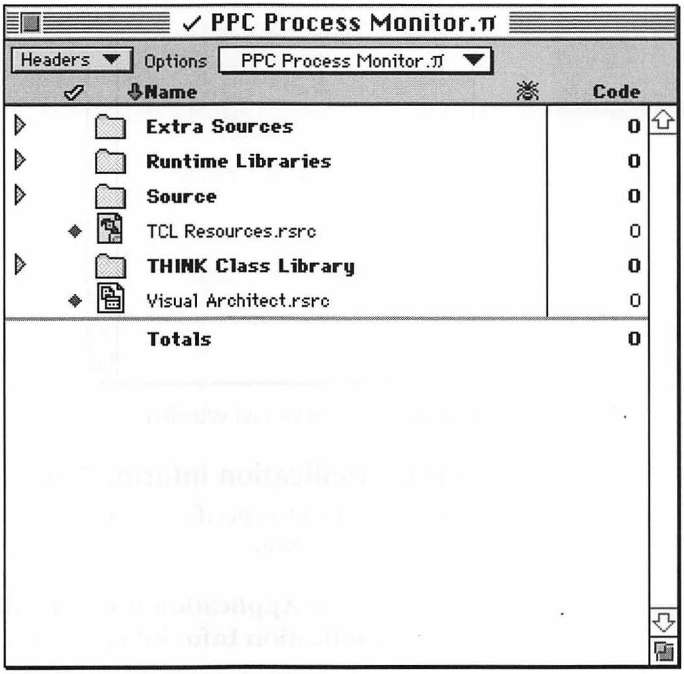
- The project has source files from the folder `Source` that are pre-edited versions of files created by Visual Architect.
- The project has source files from the folder `Extra Sources`.
- The `Visual Architect.rsrc` file has icon resources that you will use later for custom buttons.

Resources for the icons you want to use must be in the form of compiled `'cicn'` and `'ICON'` resources resident in the `Visual Architect.rsrc` file. You can use ResEdit or Resorcerer to cut and paste icons from a resource file into any `Visual Architect.rsrc` file. To save time in the setup of this tutorial, these resources have already been copied into the `Visual Architect.rsrc` file.

Opening the project and launching VA

To get started, you need to open the `PPC Process Monitor.π` project. In Finder, double-click the project's name, which you can find in `Process Monitor ƒ`, inside the `TCL Demos` folder.

At this point, the Symantec Project Manager is launched, and the `PPC Process Monitor.π` Project window opens on the right side of your screen (Figure 15-1).



✓	Name	Code
	Extra Sources	**0**
	Runtime Libraries	**0**
	Source	**0**
◆	TCL Resources.rsrc	0
	THINK Class Library	**0**
◆	Visual Architect.rsrc	0
	Totals	**0**

Figure 15-1 PPC Process Monitor.π Project window

You now need to launch Visual Architect. Double-click the `Visual Architect.rsrc` file at the bottom of the list in the Project window.

A Visual Architect View List window opens (Figure 15-2). As you create new views in the project, their names are displayed in this window. A view can be a dialog box, a window, or a subview. These resources are stored in the `Visual Architect.rsrc` file. Every Visual Architect project you create with the Symantec 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.

The View List window currently contains a sample view called Main (Figure 15-2), which displays the message "Hello, World!" You will change the characteristics of this view as you create the Process Monitor application.

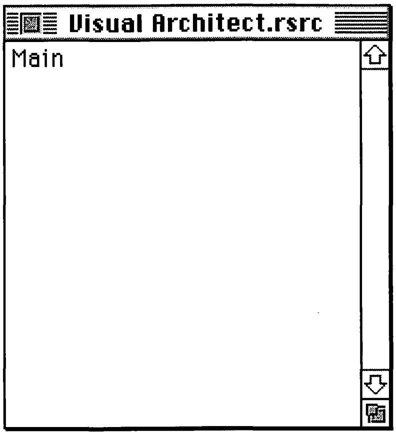
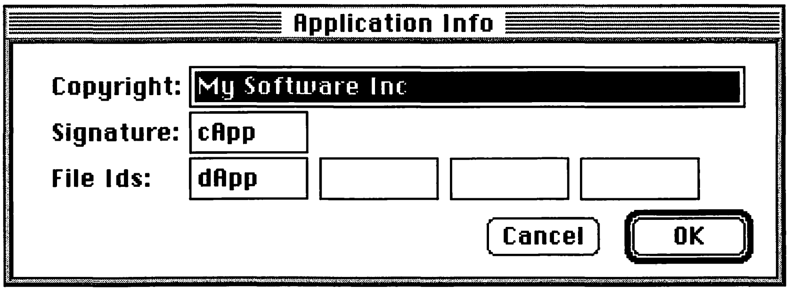


Figure 15-2 View List window

Setting application information

Now, you need to specify the top-level characteristics of the Process Monitor application.

1. Choose **Application** from the **Edit** menu to open the **Application Info** dialog box (Figure 15-3).


Application Info

Copyright: My Software Inc

Signature: cApp

File Ids: dApp

Figure 15-3 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.

Do not change the application information after generating source files for the project. Doing so will result in some of your files being regenerated.

Building the User Interface

The focus of the Process Monitor tutorial is building the interface for the application. In this section, you create the various elements required, such as windows and scroll tables.

Creating and previewing the main window

To create a main window, you start by opening the default Main view and clearing it.

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

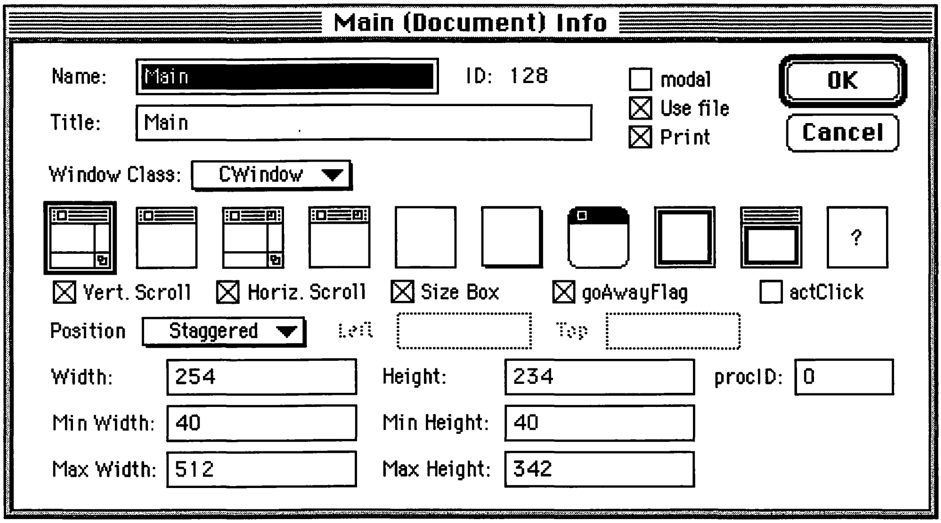
   The Main View Edit window is displayed. You should empty it in preparation for other additions.

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

Now change the characteristics of the main window.

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

   The **View Info** dialog box opens (Figure 15-4).



Main (Document) Info

Name: Main ID: 128

Title: Main

□ modal ☒ Use file ☒ Print

Window Class: CWindow

☒ Vert. Scroll ☒ Horiz. Scroll ☒ Size Box ☒ goAwayFlag □ actClick

Position Staggered Left Top

Width: 254 Height: 234 procID: 0

Min Width: 40 Min Height: 40

Max Width: 512 Max Height: 342

Figure 15-4 Main View Info dialog box

2. In the Name textbox, type `ProcessInfo` (no space) and type `Process Info` in the Title textbox.

3. Set the Use File option off.

4. Click the second window type icon from the left to make the view's window a non-scrollable, non-resizable type (procID 4).

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

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

You can now see how the window will look when the application is run and size it appropriately. Choose **Try Out** from the **View** menu (Command-Y). The window should look similiar to the one in Figure 15-5.

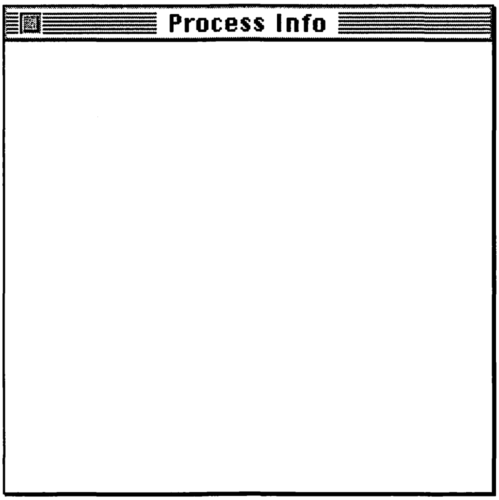


Figure 15-5 Previewing the Process Info view window

As you can see, at this point you have a small, empty window with no scroll bars or size box. Now you need to make the window large enough to accommodate the necessary information:

1. Close the window by either clicking its close box or choosing **Close** from the **File** menu (Command-W).

2. Use the size box in the Main View Edit window to extend the window's size to approximately a six-inch square.

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

Note

You use the size box to increase or decrease the working area of the window. The portRect, on the other hand, demarks the portion of the window that appears in the built application.

Drawing rectangles

Choosing tools is easier if you tear off the Tool palette and place it next to the Main View Edit window. To do so, click the **Tools** menu and drag it away from the menu bar.

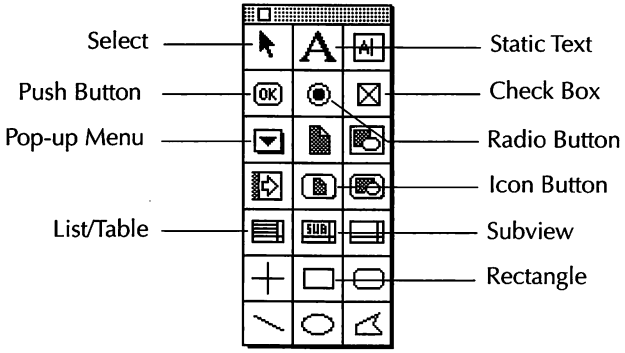


Figure 15-6 Tool palette

Now you are ready to create three rectangles to serve as decorative elements and delineate functional areas in the main window. They can also be made to serve as buttons or used to shield certain active controls from the user.

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

2. Create three rectangles in your Main View Edit window by clicking to place the upper-left corner of each rectangle and then dragging.

3. Size and position the rectangles so the View Edit window resembles the one in Figure 15-7.

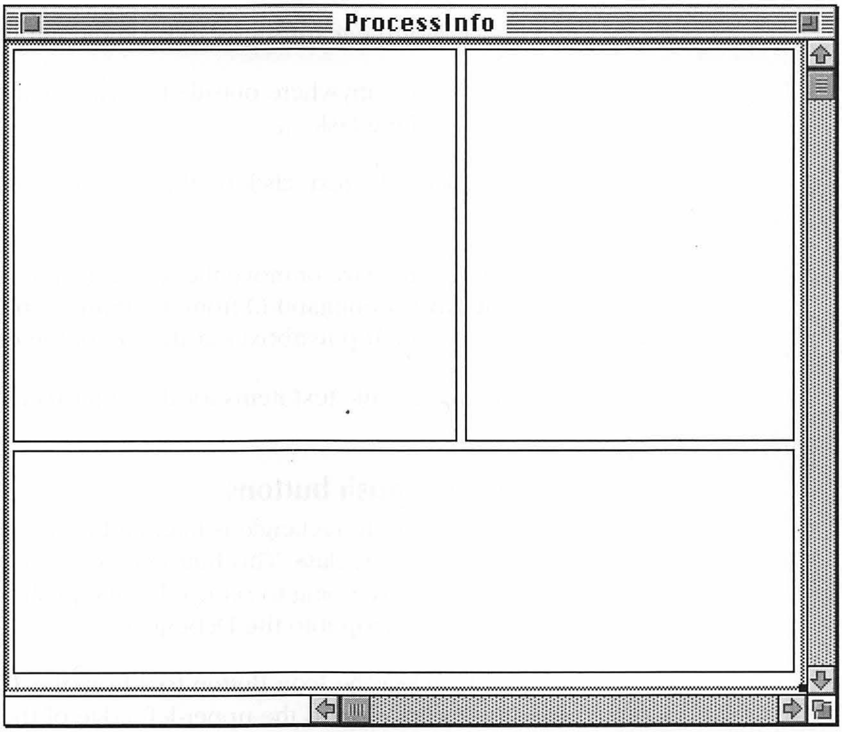


Figure 15-7 Main View Edit window with three functional areas

Creating static text items

Note

> As you work through this part of the tutorial, you may want to look ahead to Figure 15-26, which shows how the finished window appears.

You use static text items to give the rectangles titles. Begin with the upper-right rectangle:

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

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

To edit the text after ending the typing task:

1. Select the text and press Return.
2. Edit the text using standard editing procedures.
3. Click anywhere outside the static text box to end the editing task.

To position the text, click on the static text and drag it to the desired location.

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

You create static text items for the other rectangles later in the process.

Creating push buttons

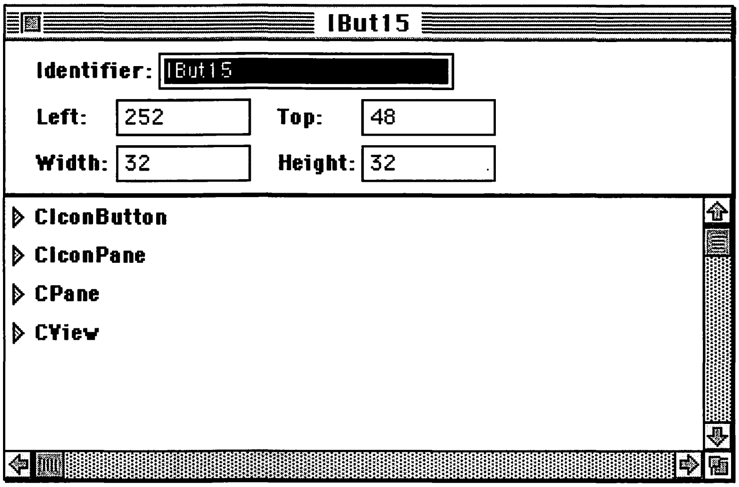
The upper-right rectangle is intended for three push buttons of the `CIconButton` class. This button type supports multiple activation states. You are going to set up the first push button to cause the program to drop into the Debugger.

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. More interesting icons, however, have been copied into the resource file for this purpose.

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

The Pane Info window opens (Figure 15-8). Note that the value in the Identifier field may vary.



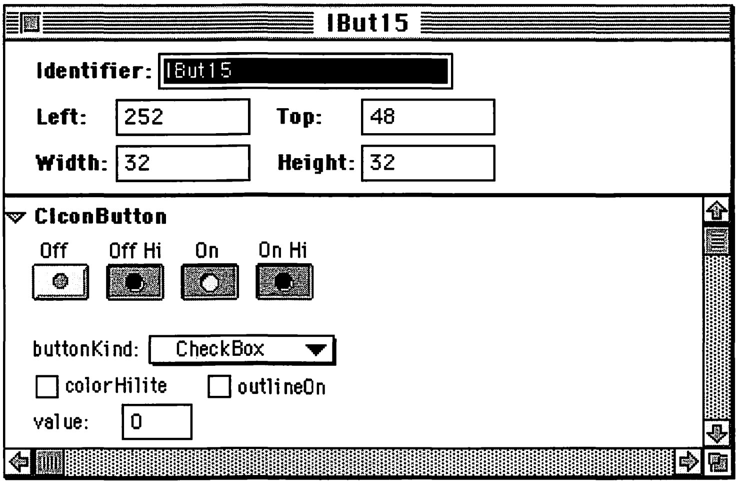
Identifier: IBut15

Left: 252 Top: 48

Width: 32 Height: 32

Figure 15-8 Pane Info window for the Debugger button

3. Open the CIconButton class by clicking the triangle to the left of CIconButton. The CIconButton class is shown (Figure 15-9).



Identifier: IBut15

Left: 252 Top: 48

Width: 32 Height: 32

buttonKind: CheckBox

□ colorHilite □ outlineOn

value: 0

Figure 15-9 Pane Info window showing CIconButton class

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 opens, displaying the icons that are in the `Visual Architect.rsrc` file and thus available for use, as shown in Figure 15-10.

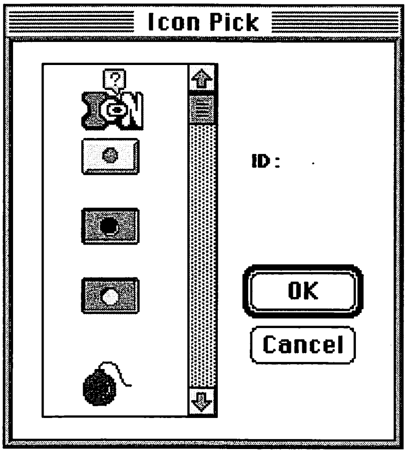


Figure 15-10 Icon Pick dialog box

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. On the Pane Info window, choose **PushButton** from the **buttonKind** pop-up menu.

8. Close the CIconButton class by clicking the triangle to the left of it.

Now it's time to give the button something to do.

Note

> After you finish working with the first push button, you will repeat these steps for the second and third push buttons.

Associating commands with buttons

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

1. On the Pane Info window, open the CIconPane class by clicking the triangle to the left of the CIconPane. Note that the Command field currently is set to `cmdNull`.

   The command to be performed by the Debugger button is not currently displayed on the list. It is a custom command called `cmdEnterDebugger` that invokes the Debugger.

   Later, code will be provided for this command. You can still add the command name to the project, however, so that a framework can be generated for the code.

2. Choose **Other** from the **Command** pop-up menu (the **Other** item is located at the top of the pop-up menu), as shown in Figure 15-11.

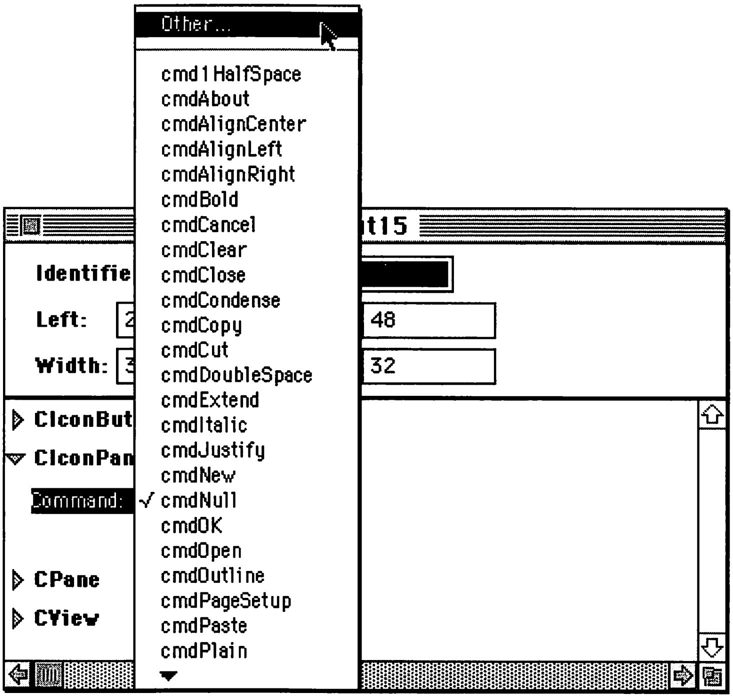
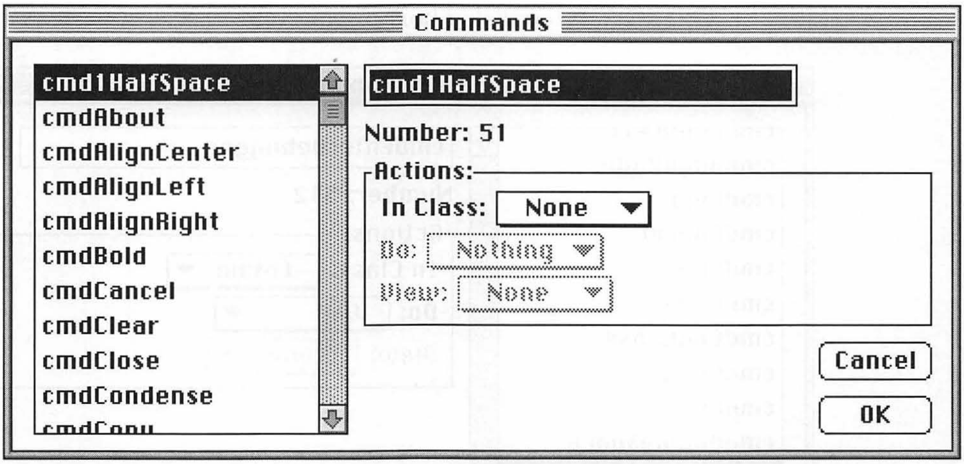


Figure 15-11 Choosing Other from the Command pop-up menu

The **Commands** dialog box opens (Figure 15-12).



Commands

cmd1HalfSpace

Number: 51

Actions:

In Class: None

Do: Nothing

View: None

Figure 15-12 Commands dialog box

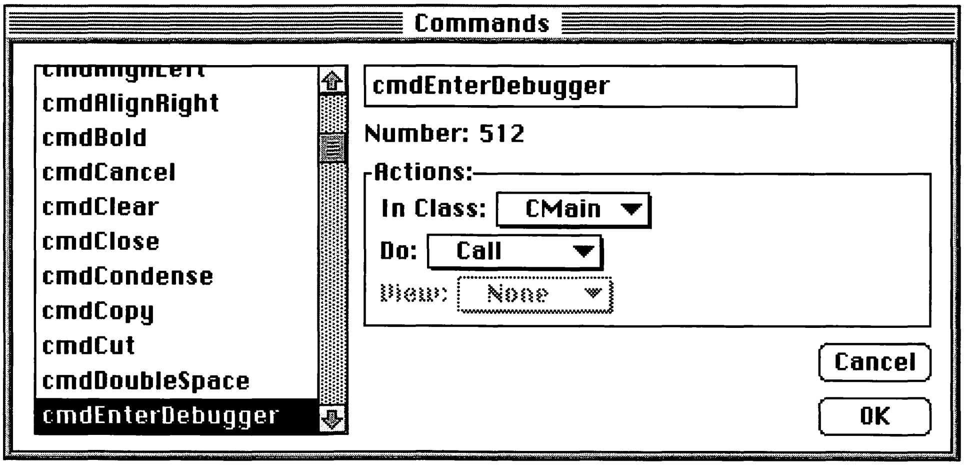
3. Choose **New Command** from the **Edit** menu (Command-K) or press Return to activate the edit text field, then type the name of the new command: `cmdEnterDebugger`.

   You now need to set up 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. When the **Commands** dialog box displays similar choices to those shown in Figure 15-13, click OK to close the dialog box.



cmdAlignLeft
cmdAlignRight
cmdBold
cmdCancel
cmdClear
cmdClose
cmdCondense
cmdCopy
cmdCut
cmdDoubleSpace
cmdEnterDebugger

cmdEnterDebugger

Number: 512

Actions:

In Class: CMain

Do: Call

View: None

Figure 15-13 Commands dialog box with appropriate choices

Back in the Pane Info window, verify that the **Command** pop-up for CIconPane is set to **cmdEnterDebugger**.

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

Assigning identifiers

By default, Visual Architect gives all the user interface elements you create a unique name. Keeping track of these elements, however, is easier if you give them more meaningful names. Name the button you just created DebuggerButton:

1. Choose **Identifier** from the **Pane** menu (Command-J) to open the **Identifier** dialog box.

2. Type DebuggerButton in the Identifier textbox (Figure 15-14).

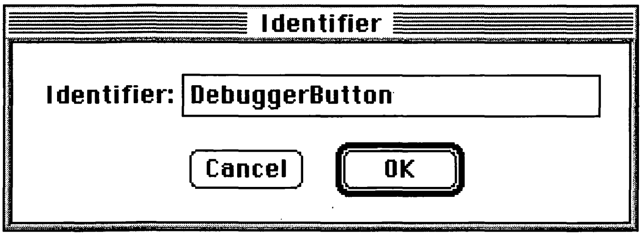


Figure 15-14 Identifier dialog box

3. Click OK to close the **Identifier** dialog box.

Note

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

You should follow these same steps for the rest of the elements you create, with the exception of simple components such as decorative rectangles and static text items. As a result, names in the code that you generate will match the names in the source files provided in the tutorial folder.

Creating the other push buttons

Now you need to repeat the steps you have just performed to create the next two push buttons.

The second push button in the group terminates the currently selected process in the process list. To create this button, repeat the earlier steps in this section, with the following changes:

- "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 follow the same steps again, with these changes:

- "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.

  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.)

- "Assigning identifiers," step 2: Set the identifier to `BringToFrontButton`.

In contrast to a check box or radio button, this CIconButton class has no associated text. You need to create a static text element to be placed next to each of the icon buttons. Using the Static Text tool, create a static text pane next to each of the three buttons, and type the names `Enter Debugger`, `Kill Process`, and `Bring to Front`, respectively.

To try out the buttons you have just created, choose **Try Out** (Command-Y) from the **View** menu. A preview window is displayed as before, but now there are active buttons.

When you are finished previewing, close the window. Either click the window's close box or choose **Close** (Command-W) from the **File** menu.

Positioning objects
You are ready to position the user interface elements you created within the Process Control section of the view. A constraint grid has been activated by default to help you align objects in straight rows and columns. Try to position the buttons to look like the ones shown in Figure 15-15.

1. Verify that **Honor Grid** in the **Options** menu is enabled.
2. Using the Select tool, select each of the three buttons and static objects in turn.
3. Move the object to the appropriate locations.

You have three options for moving objects:

- Move the selected element one grid step in any direction by pressing the appropriate arrow key (up, down, left, or right).
- Drag the object.
- 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 textboxes 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 align them as a group:

1. Select the three buttons by clicking one button, then shift-clicking the other two.
2. Choose **Left** from the **Align** hierarchical menu in the **Pane** menu.

Your view should now look like the one shown in Figure 15-15.

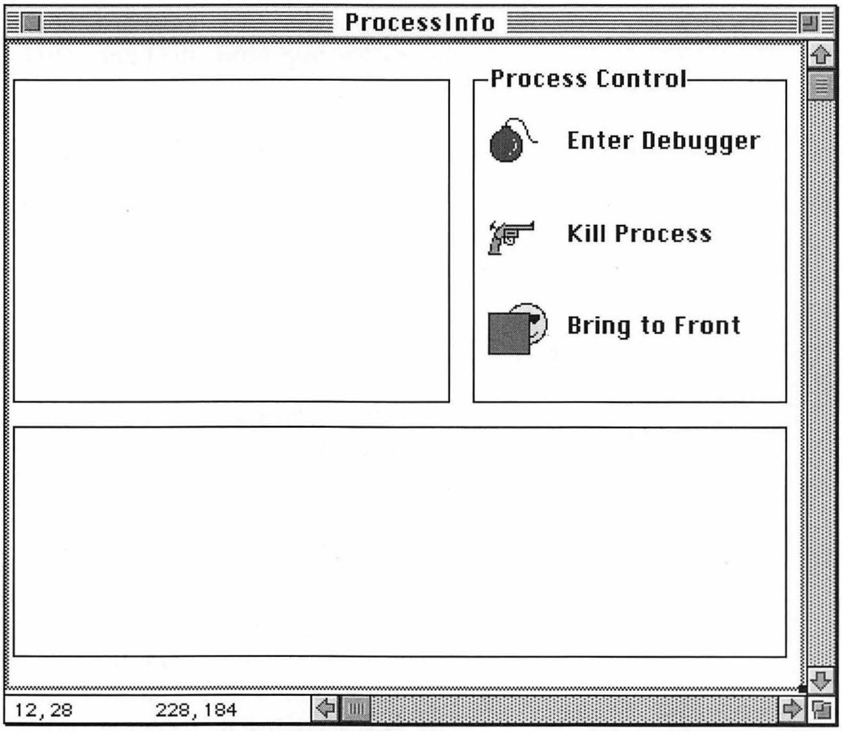


Figure 15-15 Main view with push buttons created

Creating check boxes

In the bottom rectangular section of the main window, you will create 11 checkboxes, each showing the setting of a particular flag in the `'SIZE'` resource of the currently selected process. These checkboxes should not be clickable because these flags cannot be changed by the Process Monitor application.

First provide a title for the bottom rectangle, as you did for the upper-right rectangle. Using the Static Text tool, create a static text object at the top-left corner of the bottom rectangle and name it `Size Resource`.

To create check boxes similar to those pictured in Figure 15-16.

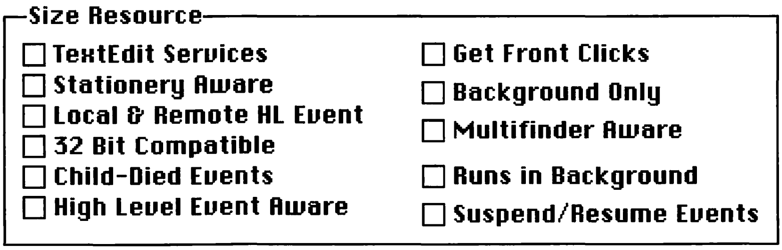
1. Double-click the Check Box tool so the tool remains selected.

Note

Repeat the following three steps for each check box.

2. Click the cursor near the location at which the check box should be positioned.

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



Size Resource

□ TextEdit Services
□ Stationery Aware
□ Local & Remote HL Event
□ 32 Bit Compatible
□ Child-Died Events
□ High Level Event Aware

□ Get Front Clicks
□ Background Only
□ Multifinder Aware
□ Runs in Background
□ Suspend/Resume Events

Figure 15-16 Size Resource check boxes

3. Type the appropriate label for the check box. Use the labels in Table 15-1.

4. Set the identifier for the check box by choosing **Identifier** from the **Pane** menu (Command-J), typing the appropriate identifier from Table 15-1, and clicking OK.

Table 15-1 Check box labels and identifiers

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

Now you need to make sure that the check boxes do not respond to users' clicks. However, by default, the `wantsClicks` attribute (in the CView part of the class) is set to `TRUE`. These particular check boxes are for display only; you do not want a user to be able to click them and change the state of the check box.

One way to prevent the check boxes from receiving clicks is to open the Pane Info window for each of the 11 check box items and change `wantsClicks` to `FALSE`. A quicker way, however, is to place the rectangle surrounding the check boxes on top of them and set it to ignore mouse clicks. To do so:

1. Select the rectangle surrounding the check boxes.

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 sheet of glass over the check boxes, allowing them to be seen but not clicked.

   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 intended for a subview. First, give the display area a title. 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.

Now, create the subview:

1. Select the Subview tool and create a subview just inside the rectangle's border, as shown in Figure 15-17.

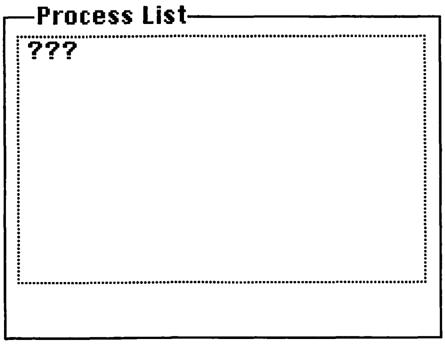


Figure 15-17 Creating the Process List subview

   You can verify that the subview has been created correctly by examining the Pane Info window for the subview.

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

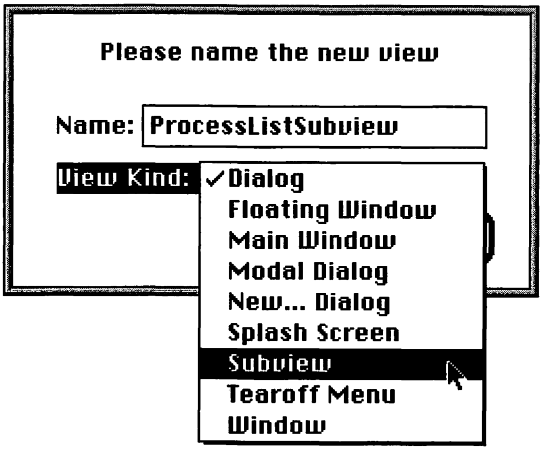
   The Subview pop-up for CSubviewDisplayer contains question marks (???) because no subview has yet been instantiated. A subview is an entirely new view, so you must first create the view.

3. Close the Pane Info window.

To create the new view, which will become the subview:

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

2. Name the subview `ProcessListSubview` and choose **Subview** from the **View Kind** pop-up menu (Figure 15-18).



Please name the new view

Name: ProcessListSubview

View Kind:

Figure 15-18 Creating a subview using the New View dialog box

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

   A new View Edit 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:

1. Choose **ProcessInfo** from the **Windows** menu.
2. Select the subview (the pane with the three question marks) in the Process List area.
3. Choose **Identifier** from the **Pane** menu (Command-J), name the subview `ProcListSubview`, and click OK.
4. Once again, choose **Pane Info** from the **Pane** menu (Command-L) to open the Pane Info window and open the CSubViewDisplayer class.
5. Even though the **Subview** pop-up already shows **ProcessListSubview**, choose **ProcessListSubview** from the **Subview** pop-up menu.

   Notice how the three question marks change to `ProcessListSubview`.
6. Close the Pane Info window by clicking in its close box or pressing Command-W.

The subview is now properly incorporated into the Main view.

Creating a scroll table

This subview will contain a list of processes in a scrolling table. Below the table is a pop-up menu, to allow a user to choose how to display the processes.

To continue the process, first return to the subview's own View Edit window, and then add the scrolling table to the subview:

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

2. Select the List/Table tool and click in the left side of the Subview window. Drag diagonally to create a scroll table pane, as shown in Figure 15-19. Make sure to leave some space at the bottom for the pop-up menu.

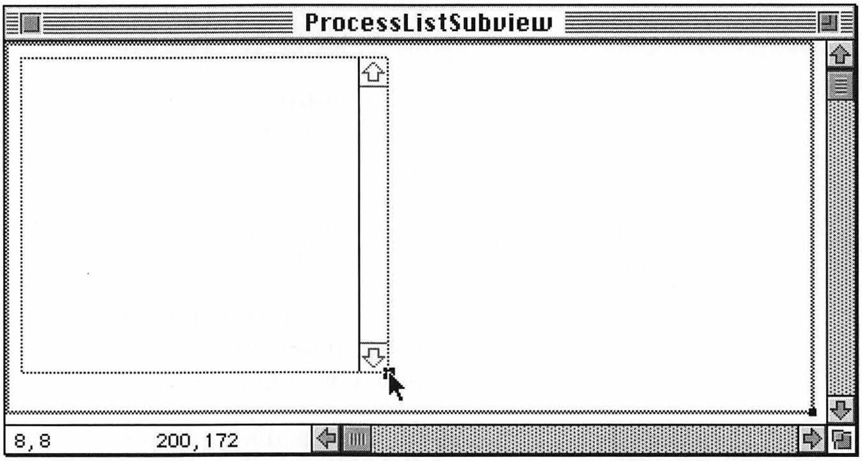


Figure 15-19 Creating a scroll table in the ProcessListSubview view

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

   When the scroll table pane is selected, the **Scrollpane Info** command in the **Pane** menu becomes enabled. Although you do not need to change any of the values in CScrollPane now, you can verify that the **Scrollpane** item is enabled.

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

   Notice how this CScrollPane has the vertical scroll enabled and horizontal scroll disabled.

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

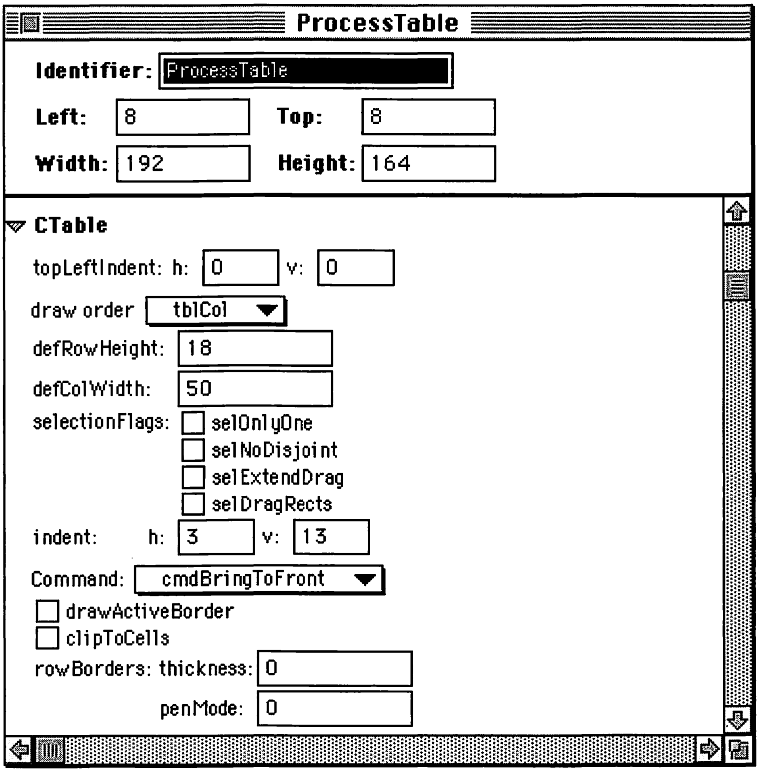
Setting the table command

The **Pane Info** window for the scroll table shows you the information on the data members for the CTable class contained within the scroll pane, not to the scroll pane.

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 (Figure 15-20).



ProcessTable

Identifier: ProcessTable

Left: 8 Top: 8

Width: 192 Height: 164

CTable

topLeftIndent: h: 0 v: 0

draw order tblCol

defRowHeight: 18

defColWidth: 50

selectionFlags: □ selOnlyOne
□ selNoDisjoint
□ selExtendDrag
□ selDragRects

indent: h: 3 v: 13

Command: cmdBringToFront

□ drawActiveBorder

□ clipToCells

rowBorders: thickness: 0

penMode: 0

Figure 15-20 Pane Info window for the ProcessTable scroll pane

3. Close the Pane Info window.

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

Creating a derived class

To give the table pane the necessary functionality to display a list of processes, you need to make it a derived class of CArrayPane. To do so:

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 (Figure 15-21).

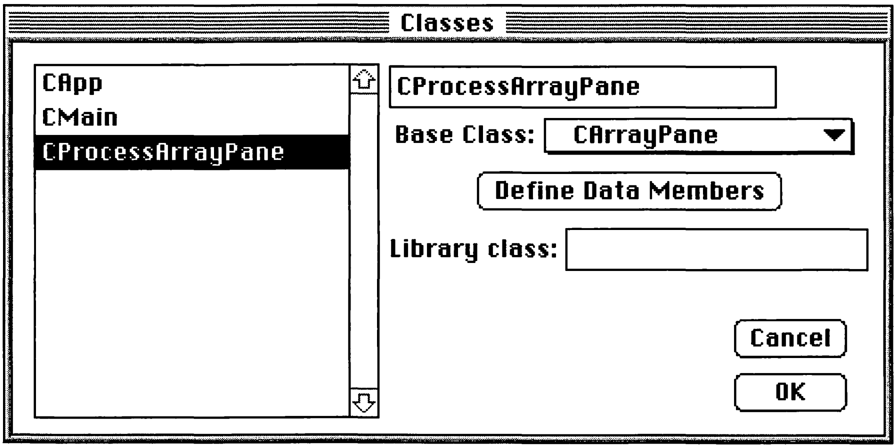


Figure 15-21 Classes dialog box

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

   You now need to set the scroll table pane to be an instance of the CProcessArrayPane class.

4. Choose **CProcessArrayPane** from the **Class** submenu in the **Pane** menu (Figure 15-22).

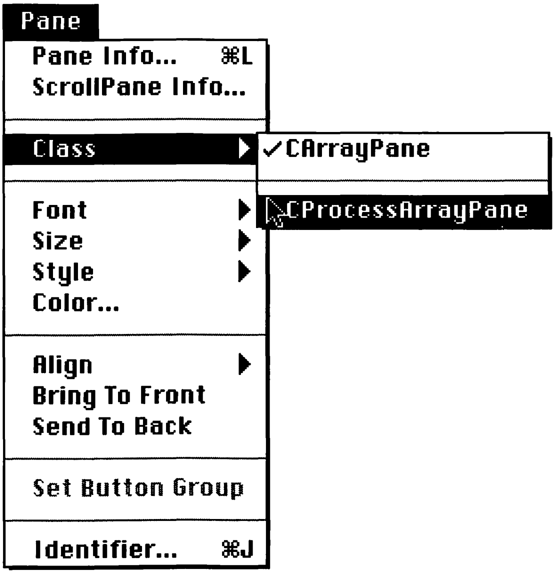


Figure 15-22 Setting the class of the scroll table pane

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—for example, to handle certain commands from a pop-up menu. To do this, you must create a new subview class derived from CPanorama:

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

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 the name `CProcListSubviewPanorama`.

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

4. Click somewhere outside the scroll pane on the ProcessListSubview view.

5. Choose **CProcListSubviewPanorama** from the **Class** submenu in the **Pane** menu.

Now, the entire subview is a CProcListSubviewPanorama into which you can add functionality.

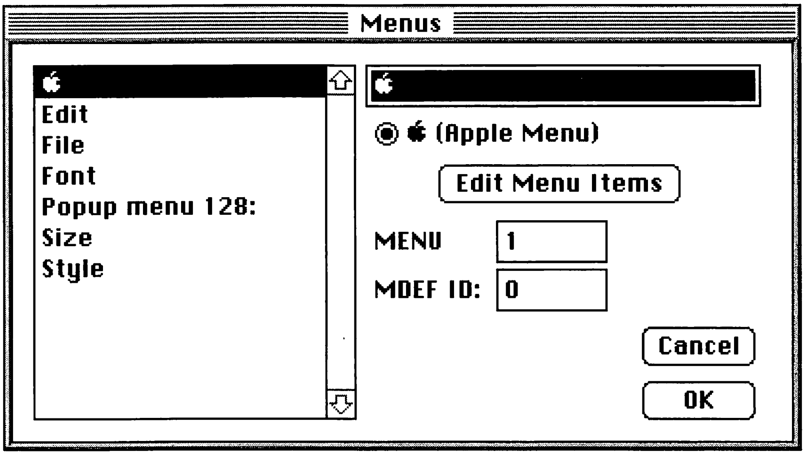
Creating a pop-up menu

To 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 displays default values. You need to enter your own values in the menu.

2. Choose **Menus** from the **Edit** menu to bring up the **Menus** dialog box (Figure 15-23).



● (Apple Menu)

MENU: 1

MDEF ID: 0

Figure 15-23 Menus dialog box

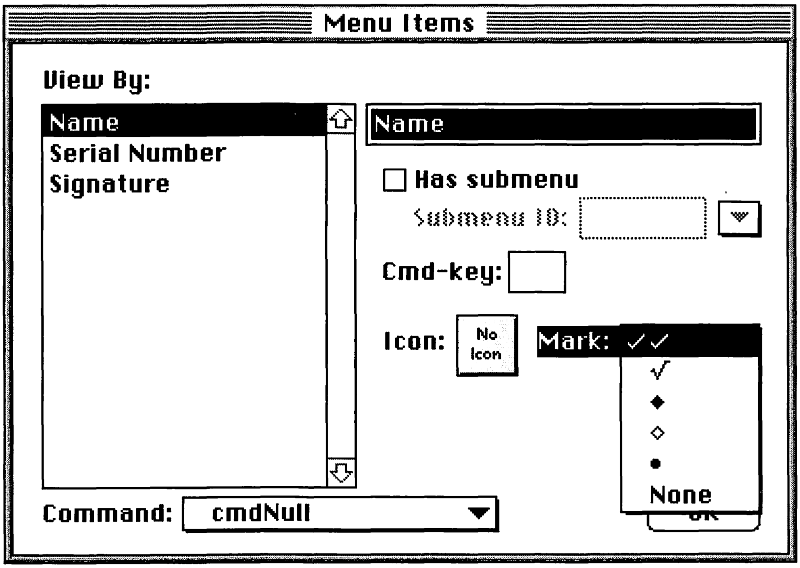
Note that the normal default menus (for example, **File** and **Edit**) already exist, as does the pop-up menu you just added, generically titled Popup Menu. You should now name this menu, as well as the items within it.

3. Click the Popup Menu item and type `View By:` into the textbox as the name of the menu.

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

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

6. Select Name from the menu item list and choose the checkmark item in the **Mark** pop-up menu (Figure 15-24).



Menu Items

View By:

Name
Serial Number
Signature

Name

□ Has submenu

Submenu ID:

Cmd-key:

Icon: No Icon

Mark: ✓✓

Command: cmdNull

Figure 15-24 Menu Items 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 pop-up menu choices, the commands must be created:

1. Click the Name item to select it.

2. Choose **Other** from the **Command** pop-up menu to open the **Commands** dialog box.

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.

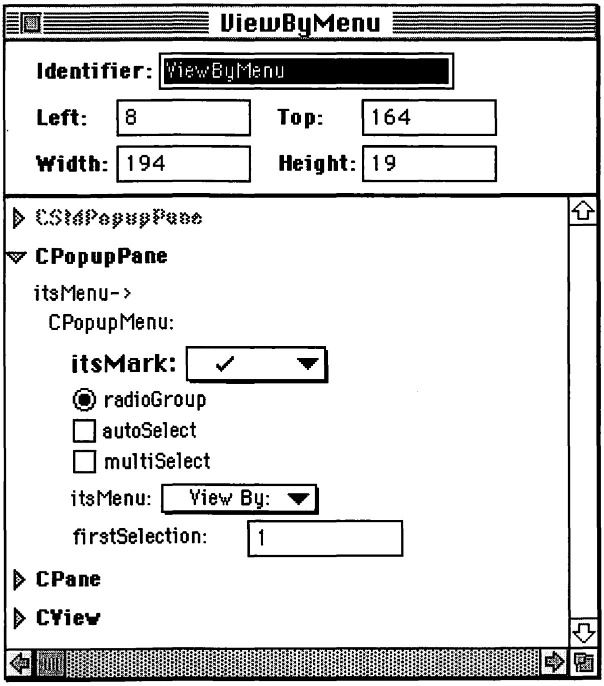
Now the CProcListSubviewPanorama can handle any of the three menu selections.

Finishing the pop-up menu

You are virtually finished with the pop-up menu. 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.

2. Click the radioGroup radio button (Figure 15-25) and close the CPopupPaneClass.



ViewByMenu

Identifier: ViewByMenu

Left: 8 Top: 164

Width: 194 Height: 19

▷ CStdPopupPane

▽ CPopupPane

itsMenu->

CPopupMenu:

itsMark: ✓

● radioGroup

□ autoSelect

□ multiSelect

itsMenu: View By:

firstSelection: 1

▷ CPane

▷ CView

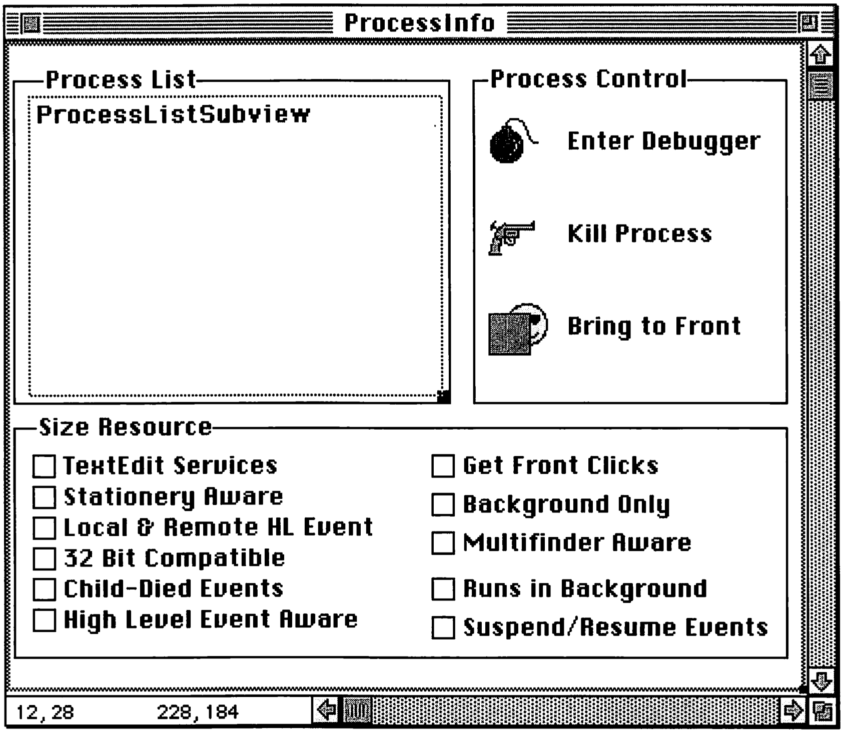
Figure 15-25 Clicking radioGroup in the Pane Info window

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

4. Close the ProcessListSubview View Edit window.

Trying out the completed main window

Your Main view should now appear as shown in Figure 15-26.



ProcessInfo

Process List

ProcessListSubview

Process Control

Enter Debugger

Kill Process

Bring to Front

Size Resource

□ TextEdit Services
□ Stationery Aware
□ Local & Remote HL Event
□ 32 Bit Compatible
□ Child-Died Events
□ High Level Event Aware

□ Get Front Clicks
□ Background Only
□ Multifinder Aware
□ Runs in Background
□ Suspend/Resume Events

12, 28 228, 184

Figure 15-26 Completed Main view

To preview the appearance of the Main view in the running application:

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

2. Verify that the controls function properly. One such test is shown in Figure 15-27.

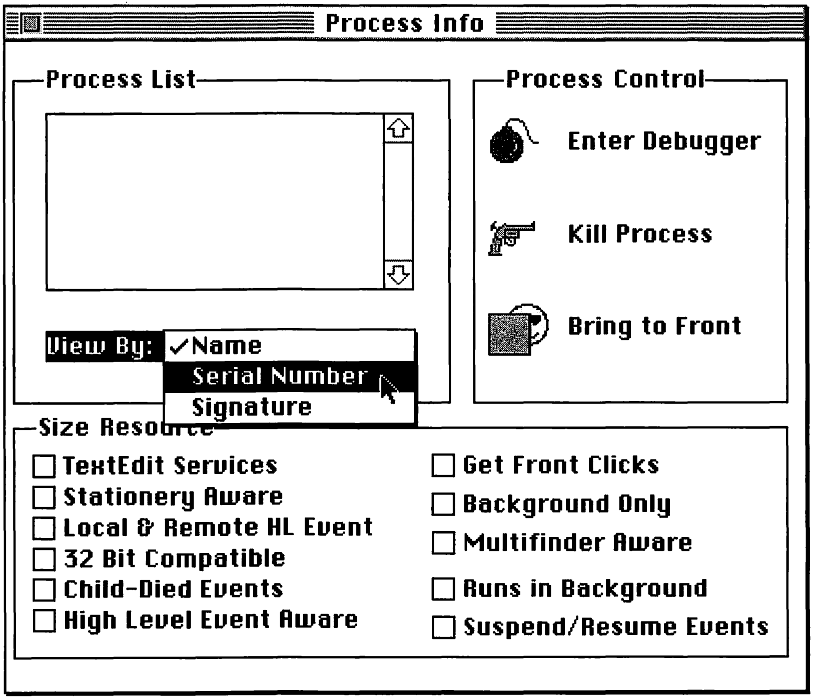


Figure 15-27 Previewing the Main view

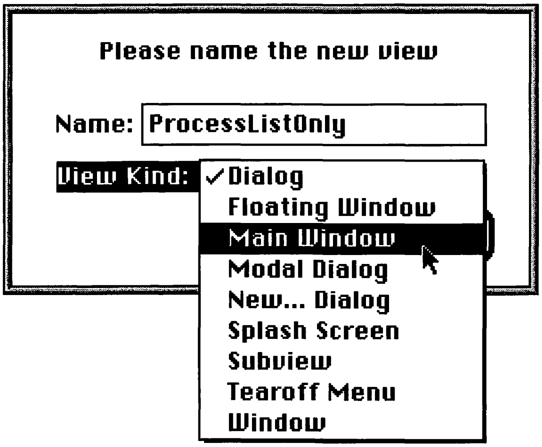
3. Click on the close box to close the preview window.

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. Type `ProcessListOnly` as the view name and choose **Main Window** from the **View Kind** pop-up menu (Figure 15-28).



Please name the new view

Name: ProcessListOnly

View Kind:

Figure 15-28 Creating a Main view

3. Click OK to close the **New View** dialog box to open the ProcessListOnly View Edit window.

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

5. Set the Use File option off.

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

7. 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). Click OK.

This view is going to contain the same subview as the main window. As before, you should create a subview in the window.

1. Select the Subview tool and create a subview just inside the gray-outlined rectangle within the window (Figure 15-29).

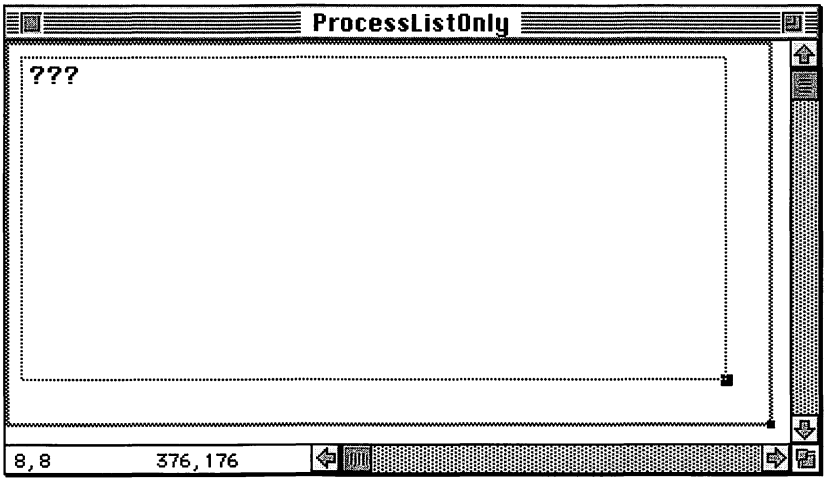


Figure 15-29 Creating the subview in the ProcessListOnly view

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

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

   Because the subview handles its own commands, you do not need to assign command handlers for this window.

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

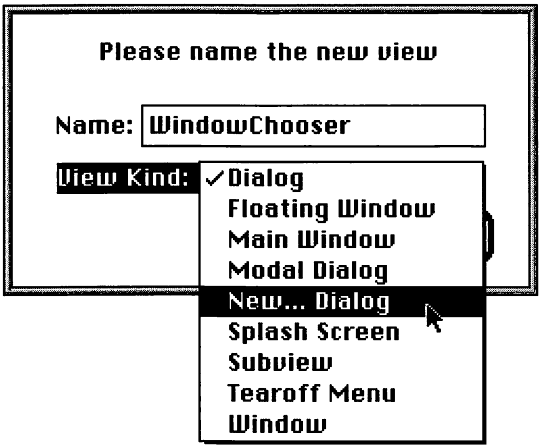
4. Close the ProcessListOnly View Edit window.

Creating the New… Dialog

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

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

2. Type `WindowChooser` as the view's name and choose **New... Dialog** from the **View Kind** pop-up menu (Figure 15-30).



Please name the new view

Name: WindowChooser

View Kind: New... Dialog

Figure 15-30 Creating a New... Dialog view

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

The WindowChooser View Edit window opens. 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 a view to create.

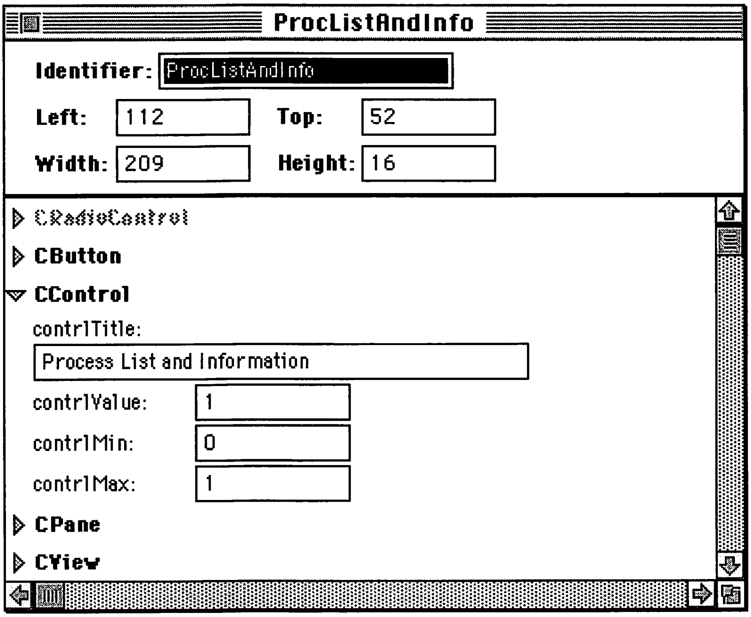
1. With the Static Text tool, create a static text item at the top of the dialog and type `Create a window showing:`.

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 15-31, so this button will be selected by default, and close the Pane Info window.



ProcListAndInfo

Identifier: ProcListAndInfo

Left: 112 Top: 52

Width: 209 Height: 16

▷ CRadioControl

▷ CButton

▽ CControl

contrlTitle: Process List and Information

contrlValue: 1

contrlMin: 0

contrlMax: 1

▷ CPane

▷ CView

Figure 15-31 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.

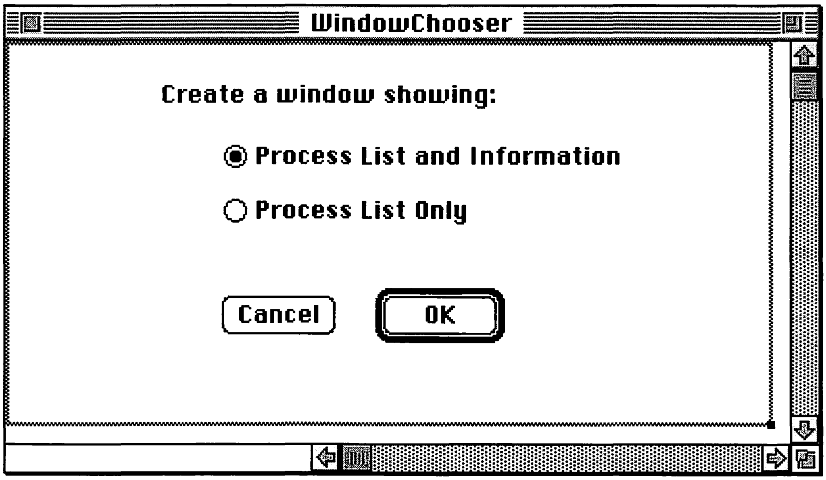
   By default, the first button you add to a view is named OK and has `cmdOK` as its command.

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 View Edit window should resemble that shown in Figure 15-32.



Create a window showing:

● Process List and Information

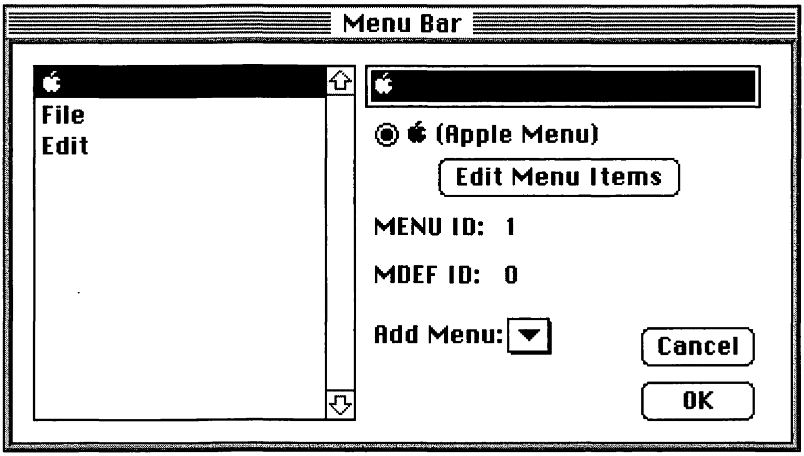
○ Process List Only

Figure 15-32 The New... Dialog View Edit window

Editing menus

Before saving the project, you have one final user interface task: customizing the menu bar for the application. This tutorial did not attempt to build a complete menu; therefore, you need to remove menu choices that are not currently supported.

1. Choose **Menu Bar** from the **Edit** menu to open the **Menu Bar** dialog box (Figure 15-33).

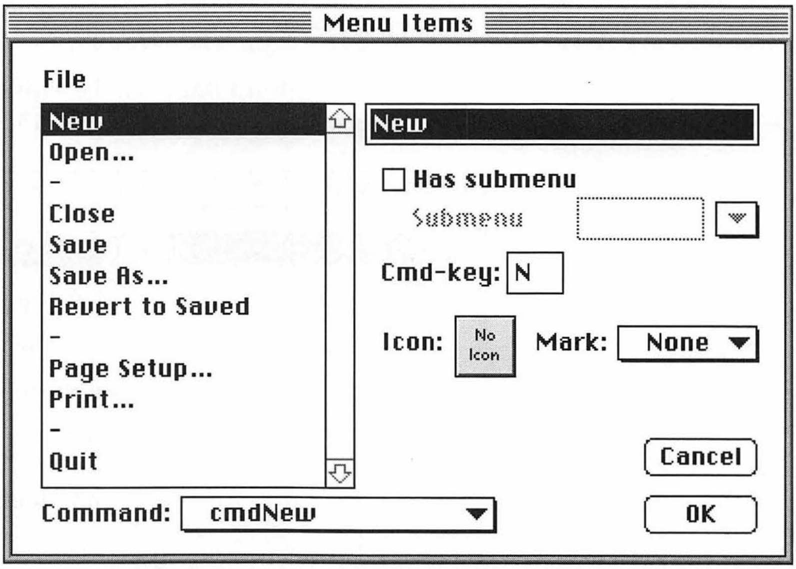


MENU ID: 1

MDEF ID: 0

Figure 15-33 Menu Bar dialog box

2. Select the **File** menu from the list and click the Edit Menu Items button to open the **Menu Items** dialog box (Figure 15-34).



New

□ Has submenu

Cmd-key: N

Mark: None

Command: cmdNew

Figure 15-34 Menu Items dialog box

3. Delete all menu items except New, Close, and Quit by selecting them from the list and pressing Delete. Leave 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.

8. Choose **Save** from the **File** menu (Command-S) to save the `Visual Architect.rsrc` file.

Generating Code for Your Application

With the user interface for the application built, you are ready to generate the code that enables it to run. This code falls into two categories: support of the application's function (the display and manipulation of running processes) and support of the user interface. Visual Architect generates the latter type of code; typically, you would still need to write the functional code. For this tutorial, however, the functional 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 **Symantec Project Manager** menu. The title of the menu is the Symantec Project Manager application icon (Figure 15-35).

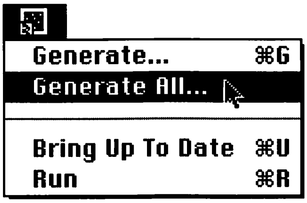
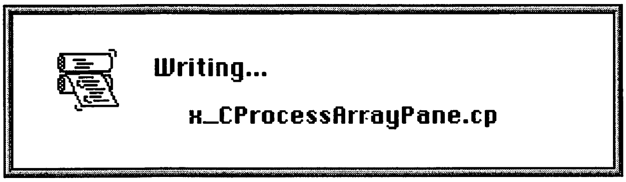


Figure 15-35 Symantec Project Manager menu

A message box is displayed showing the progress of the code generation (Figure 15-36).


Writing...

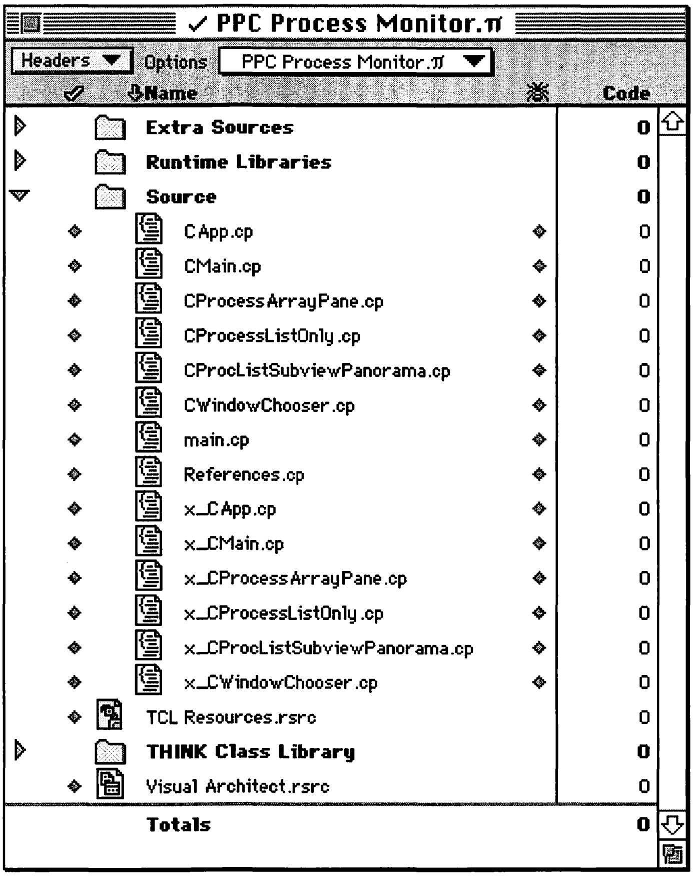
x_CProcessArrayPane.cp

Figure 15-36 Code Generation Progress message 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 called `Source` in your `Process Monitor` folder contains the generated code.

2. Switch to the Symantec Project Manager application and open the Source group 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 Source group (Figure 15-37).



	✓	Name		Code
▶		**Extra Sources**		**0**
▶		**Runtime Libraries**		**0**
▼		**Source**		**0**
	◆	CApp.cp	◆	0
	◆	CMain.cp	◆	0
	◆	CProcessArrayPane.cp	◆	0
	◆	CProcessListOnly.cp	◆	0
	◆	CProcListSubviewPanorama.cp	◆	0
	◆	CWindowChooser.cp	◆	0
	◆	main.cp	◆	0
	◆	References.cp	◆	0
	◆	x_CApp.cp	◆	0
	◆	x_CMain.cp	◆	0
	◆	x_CProcessArrayPane.cp	◆	0
	◆	x_CProcessListOnly.cp	◆	0
	◆	x_CProcListSubviewPanorama.cp	◆	0
	◆	x_CWindowChooser.cp	◆	0
	◆	TCL Resources.rsrc		0
▶		**THINK Class Library**		**0**
	◆	Visual Architect.rsrc		0
		Totals		**0**

Figure 15-37 PPC Process Monitor.π Project window

Customizing code

At this point, Visual Architect has generated the user interface code. This typically is the time at which you would add your functional code. As mentioned earlier, however, the functional part already has been created.

To understand 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 as follows:

```
// ••••• 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 do not have any visual elements, so they cannot be specified by Visual Architect. These files have already been added to the project.

Running the application

To run the Process Monitor application:

1. Choose **Bring Up to Date** from the **Project** menu.
2. Choose **Run** from the **Project** menu.

If you receive compilation errors, you have probably misspelled or forgotten to set one of the identifiers. Launch Visual Architect to correct the mistake, then choose **Run** from Visual Architect's **Symantec Project Manager** menu. Files are regenerated automatically, as necessary.

Once the program is running, you are free to experiment with it. However, keep in mind the following:

- Processes with the Runs in Background option set off cannot be killed until those processes are brought to the foreground. The application will not process the Quit event until it executes its event loop. Try killing Visual Architect to test this rule.
- Some processes in the list may not respond to clicking the Bring to Front button. Look at the Background Only bit in the Size Resource section. If it is set on, then the selected process does not have a user interface and cannot be queued as the foreground application.
- If you have not installed a low-level debugger such as MacsBug, be sure you run the tutorial under the Symantec Debugger before clicking the Enter Debugger button.

Symantec C++

Symantec Project Manager Reference

Part Four

24 The Debugger Windows

25 The Debugger Menus

26 The Windows Menu

The File Menu 16

This reference chapter provides a detailed explanation of all commands in the Symantec Project Manager **File** menu. The project models and options available from the **New Project** dialog box are described, along with the options in the **File Open** dialog box. The final sections of the chapter cover creating and modifying project models.

Commands in the File Menu

You use the Symantec Project Manager **File** menu to create new source code files and to open existing ones. You also use this menu to save and print your source code and to create new project files as you need them. Figure 16-1 shows the **File** menu.

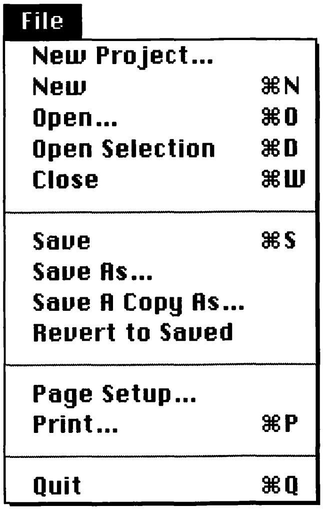


Figure 16-1 File menu

These commands primarily are used to perform the following functions:

- Open or create projects, Editor windows, and files
- Save files and close windows
- Print

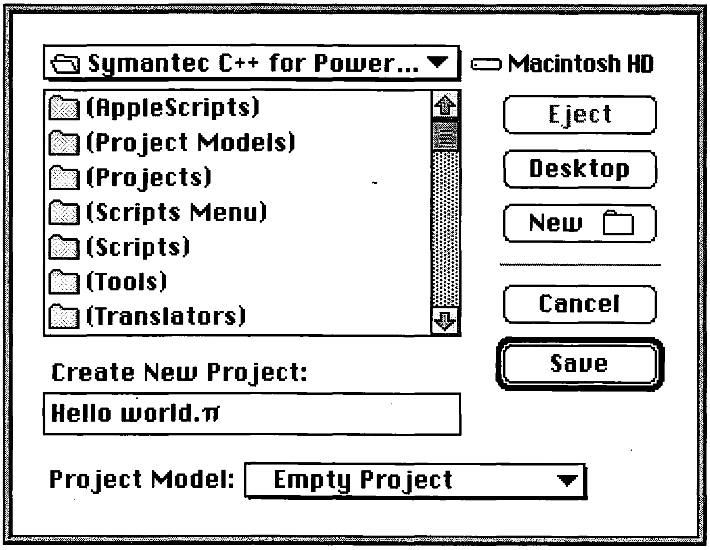
This chapter discusses the **File** menu commands by function in the order listed above.

Opening projects, Editor windows, and files

The first four commands in the menu let you create new projects and empty Editor windows, as well as open files.

New Project

Opens the **New Project** dialog box (Figure 16-2), from which you can create a project.



Create New Project: Hello world.π

Project Model: Empty Project

Figure 16-2 New Project dialog box

To create a new project, you set options in this dialog box and click Save. First type a name for the project and choose a project model from the pop-up menu. You also have the option to create a new folder for the project.

Project Model pop-up menu

Use the **Project Model** pop-up menu to choose a project model for a project.

Project models are templates from which new projects are created. They define the libraries, resources, and source code files that the project initially contains, as well as the project's options settings. Symantec provides models for many common project types. You can create your own project models and display them in the **Project Model** pop-up menu.

Note

> Once a project is created from a project model, you can add files, remove files, and change settings as needed. For details, see the section "Modifying Project Models," later in this chapter.

The project models displayed in Figure 16-3 are included with Symantec C++ for Power Macintosh.

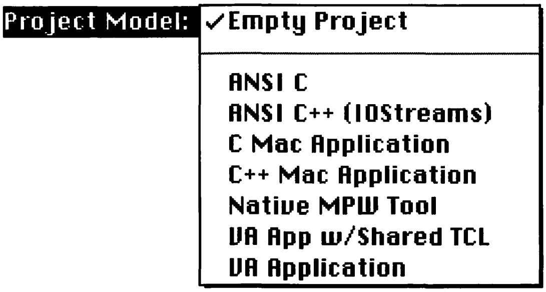


Figure 16-3 Project Model pop-up menu

Empty Project. Creates an empty Project file with no source files. This model is built into the Symantec Project Manager and is always available. The Empty Project is the only project model that you cannot change.

ANSI C. Builds an application in C that makes use of the console package for input and display of text instead of calling the Toolbox directly.

The ANSI library included with Symantec C++ for Power Macintosh lets you create programs that perform input and output using only functions in the ANSI libraries. Calls to functions such as `printf` display text in a window and calls to `scanf` read input from the keyboard. Being able to create applications that do not call the Macintosh Toolbox directly is helpful if you are porting a program from another platform or working with examples from books designed for other operating systems.

The console package also includes some functions not found in the standard ANSI library. These include commands that open a dialog box to allow the entry of command line arguments and `cecho2printer` and `cecho2file`, which echo the output of a given console window to a printer or file respectively.

ANSI C++ (IOStreams). Builds an application in C++ if you do not want to call the Toolbox directly. The model is the same as the ANSI C model, except that it includes the libraries required to support programming in C++ and to use the IOStreams package.

C Mac Application. Builds an application in C that calls the Macintosh Toolbox directly. Choose this if you already have some Toolbox-based code written in C that you want to include in your project.

C++ Mac Application. Builds an application in C++ that calls the Macintosh Toolbox directly. Choose this if you want to use a Toolbox-based class library other than the THINK Class Library.

Native MPW Tool. Builds a tool to use with the Macintosh Programmer's Workshop. You may also use MPW tools from the Symantec Program Manager with ToolServer. For more information on ToolServer, see the section "Using ToolServer," in Chapter 8, "Advanced Topics."

VA Project Models. Builds an application using the THINK Class Library and Visual Architect. These are the most powerful of the project models and the best choice for starting a new application, unless you want to use a different class library or have existing code handle Toolbox functions.

A project built with either of the VA project models will contain all the source code for the Think Class Library and all the resources the class library requires. A project built from either one will be ready to interface with Visual Architect.

VA App w/Shared TCL. Includes the THINK Class Library as a shared library; all projects built from this model share a copy of the TCL's object code and debugging information. Also, after the first time you compile the TCL's library, you do not have to recompile it each time you start a new project. Therefore, use this model if you are developing more than one project based on the THINK Class Library and would like to save disk space and compilation time.

The disadvantage of using this model is that the resulting application consists of two files, the application itself and the TCL's shared library. This may not be suitable for some applications because users could receive multiple (possibly differing) copies of the TCL library from different products.

VA Application. Includes the source code to the TCL as part of the project. When built, the application consists of a single application file. Therefore, use this model when you are working on a single project based on the THINK Class Library, as well as for the final versions of projects that you will be distributing.

Create New Project textbox
In this textbox, enter the name of a project with the extension .π (press Option-P to type π).

New Project Folder button
Click this button to create a new folder for a project.

New ⌘N

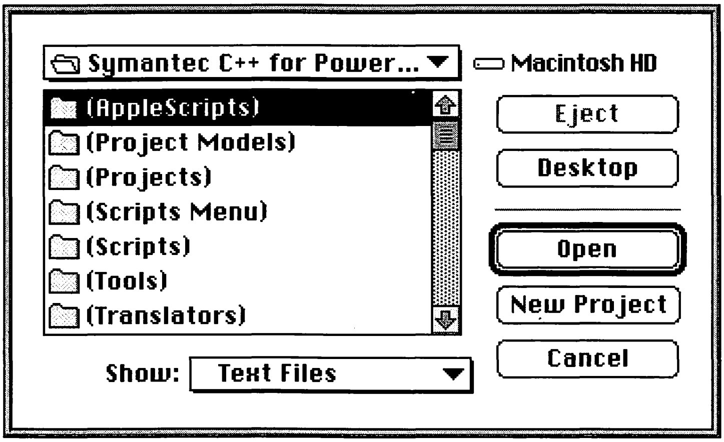
Opens an empty and untitled Editor window. You must save the contents of the new window before you can add it to a project. Once the file has been saved, you can add it to the active project by choosing **Add "*filename*"** or **Add Files** from the **Project** menu.

Note

> For the translators to work on contents of the file, you must give it an appropriate extension (such as ".c" for C files). The extensions for each translator are listed on the Extensions Mapping page of the Project Manager's **Project Options** dialog box. For further information, see the section "Project Options Page," in Chapter 18, "The Project Menu."

Open ⌘O

Opens a **File Open** dialog box (Figure 16-4), which you can use to open text and project files.



Show: Text Files

Figure 16-4 File Open dialog box

The **File Open** dialog box is similar to a standard **File Open** dialog box with the addition of two controls: the New Project button and a pop-up menu to control the kind of files displayed in the dialog box.

Note

> Any project opened with the **Open** command becomes the active project.

The Project Manager lets you have more than one project open at a time. However, only one project at a time is considered to be active. You can make any open project—or any project that you have opened since you opened the Project Manager—the active project by choosing its name from the **Switch Main Project** submenu of the **Project** menu. If you choose a project that is not currently open from the **Switch Main Project** submenu, the Project Manager opens that project and makes it the active project.

Show pop-up menu

The **Show** pop-up menu, located at the bottom of the **File Open** dialog box, is used to set a file filter for displaying files (Figure 16-5).

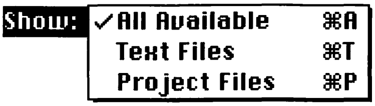


Figure 16-5 Show pop-up menu

You have three alternatives:

All Available. Displays folders, text files, Symantec Project Manager project files, and Think Project Manager project files.

Text Files. Displays only folders and text files.

Project Files. Displays only folders and Symantec Project Manager project files.

New Project button

Click this button to open a new project. This has the same effect as choosing **New Project** from the **File** menu.

Open Selection ⌘D

Open In Editor ⌥⌘D

Opens the currently selected file(s) if the frontmost window is the Project window. If the frontmost window is an Editor window, selecting this command opens the currently selected file and extends the selection to the right to include the file's extension (such as "`.h`"), if appropriate. You can use this command to open a header file by selecting the name from an `#include` statement.

If you hold down the Option key while the **File** menu is selected, this command becomes **Open In Editor**. If you select **Open In Editor** when a Class Browser window is frontmost, an Editor window opens that contains the source file currently displayed in the Class Browser's source pane.

Saving files and closing windows

These commands let you save files, revert to the last-saved version, and close windows.

Save ⌘S

Save All ⌥⌘S

Saves the contents of the frontmost Editor window to disk. If the file has not yet been saved to disk, a standard **File Save** dialog box opens for naming and placing the file.

If you hold down the Option key with the **File** menu selected, this command toggles to **Save All**. Choosing the **Save All** command saves the contents of all open windows that have unsaved modifications. This includes any text in the Worksheet window (if open) and any changes made to source code in Class Browser windows.

Save As

Saves a copy of the contents of the frontmost Editor window with a new name. If the file has been added to the project, the version of the file with the new name replaces the older version in the Project file.

Note

If you don't want the new version of the file to replace the older one in the project, use the **Save A Copy As** command instead of **Save As** to save the file.

Save A Copy As

Saves a copy of a file without changing the project. If the frontmost window contains text, a copy of the text file is made. If the frontmost window is a Project window, a copy of the Project file is made, minus any object code from the original. This is a good way to make a temporary backup of a file. A standard **File Save** dialog box opens to let you name and place the file.

Revert to Saved

Replaces the contents of the frontmost Editor window with the most recently saved version of the file.

Close ⌘W
Close All ⌥⌘W

Closes the frontmost window. You can perform the same function by clicking the close box on the left side of the window's title bar.

If you hold down the Option key with the **File** menu selected, this item toggles to **Close All**, which closes all open windows.

If you have modified text in an Editor window since the last save and have checked the **Confirm Saves** option on the Editor Options General Settings page of the **Project Manager Preferences** dialog box, you are prompted to save your changes. Otherwise, any changes are automatically saved.

Quit ⌘Q

Closes the Symantec Project Manager and all open windows. If any of these windows contain unsaved changes, you are prompted to save them.

Printing

These commands cover page setup and printing.

Page Setup

Opens the standard **Page Setup** dialog box for your printer. See the manuals that came with your Macintosh for more details.

Print ⌘P

Prints the contents of the frontmost window. You can print the following types of windows: Editor windows, the active pane of a Class Browser window, Project windows, Search Results windows, Build Errors windows, and Link Errors windows.

Modifying Project Models

You can modify any of the project models included with Symantec C++ for Power Macintosh except the Empty Project. You can change, add, or remove any file or change any setting, once a project is created. You can change one kind of project to another in this way. If you want to change the basic settings of a project model that Symantec provides, you should make a copy of the project model, make the necessary modifications, and add the new project model to the models folder.

To change a project's initial options, open the project named `@1` in the folder with the name of the project model you want to modify. This folder is located in the `(Project Models)` folder, which is located in the same folder as the Symantec Project Manager application. You might want to do this if you repeatedly change options each time you create a new project from a given project model.

For example, you might add named option sets for debugging and nondebugging versions of applications. You can create two new models by modifying the model you use to start projects with so each model contains one of the two option sets.

When the Project Manager creates a new project from a project model, it copies the project file named `@1` from the project model's folder into the new project's folder and gives it the name the user enters in the **New Project** dialog box. The Project Manager then copies all files and folders (including their contents) in the project model's folder to the location of the new project.

Creating Custom Project Models

You can add your own project models to the **Project Model** pop-up menu. This is useful if you have a standard Project file from which you always start new projects. If you turn it into a project model, you can create copies of it without ever having to leave the Project Manager.

Note

The project list in the **Project Model** pop-up menu is sorted alphabetically. If you want a model to appear at the top of the list, add a space at the beginning of the project's folder's name.

To turn an existing project into a project model, you need to complete tasks in both the Project Manager and the Finder.

Note

You should complete these steps with a duplicate of the project.

The following steps are performed in the Project Manager:

1. Open the project in the Project Manager.
2. From the Project file, remove any source code files you do not want to include in the project model by choosing **Remove** from the **Project** menu.
3. Add any source code files that you want to add.
4. Choose **Remove Objects** from the **Build** menu.
5. Quit the Project Manager.

The following steps are performed in the Finder:

1. From the project folder, delete any source code files that you do not want to include in the project model.
2. Change the primary Project file's name to `@1`.

3. Rename the project's folder with the name you would like displayed in the **Project Model** pop-up menu ("My Project Model" for example).

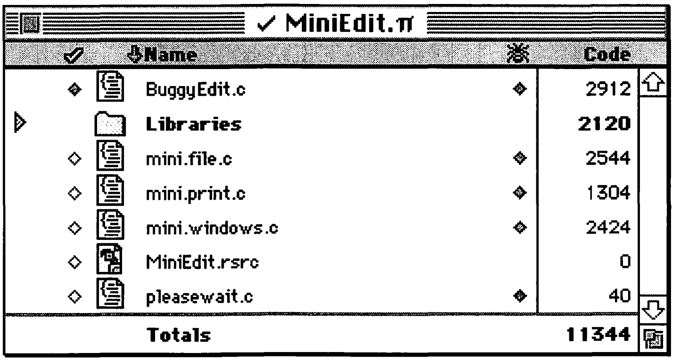
4. Move the project's folder and its contents into the folder named `(Project Models)`. The `(Project Models)` folder is located in the same folder as the Symantec Project Manager (that is, the Symantec Project Manager's program file).

The Project Window 17

This reference chapter describes both how to display information in the Project window and what project information can be displayed. The information the Project window can display includes make and debug status, file location and last modification date, code and data size, and project organization.

Introducing the Project Window

The Project window shows the contents of your Project file as well as the status of each component. Any time you open a project, whether through the **File** menu or **Project** menu, you open a Project window. Figure 17-1 shows the default Project window for a project.



✓ MiniEdit.π

✓		Name		Code
	◆	BuggyEdit.c	◆	2912
▷		**Libraries**		**2120**
	◇	mini.file.c	◆	2544
	◇	mini.print.c	◆	1304
	◇	mini.windows.c	◆	2424
	◇	MiniEdit.rsrc		0
	◇	pleasewait.c	◆	40
		Totals		**11344**

Figure 17-1 Project window

You use the **Project** menu and the **Windows** menu to configure the components displayed in the Project window. The **Windows** menu controls general window characteristics and the **Project** menu controls the arrangement of files. Selecting the **Show Toolbar** command from the **Windows** menu, for example, adds the **Header** and **Options** pop-up menus to the Project window toolbar.

In addition, you can control what and how the Project window displays information by setting options in the Project Window page of the **Project Options** dialog box. For information about Project window options, see Chapter 18, "The Project Menu."

The following sections describe all the kinds of information that can be displayed in the Project window. To display every information type, set all options on the Project Window page of the **Project Options** dialog box and set the **Show Toolbar** command on.

Pop-up menus on the toolbar

If you choose the **Show Toolbar** command from the **Windows** menu, the Project window displays two pop-up menus: **Headers** and **Options**. These two menus are described next.

Headers pop-up menu

The **Headers** menu lists the include files for the source files in the project (Figure 17-2). To view a header file, choose its name from the pop-up menu. A text Editor window is opened containing the selected file. See section "Window features," in Chapter 19, "The Editor Window" for more information on the **Headers** pop-up menu.

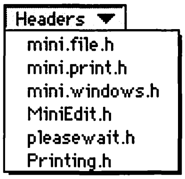


Figure 17-2 Headers pop-up menu

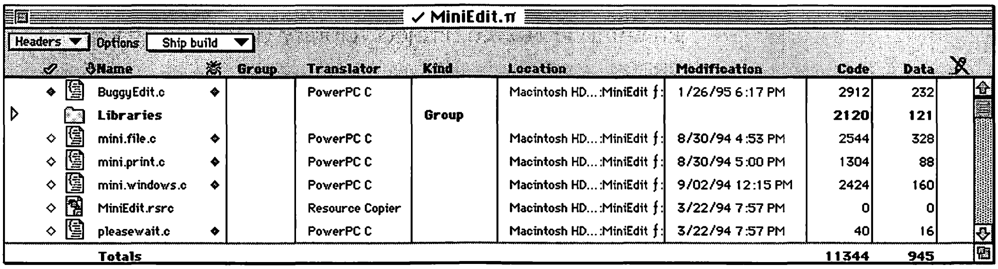
Options pop-up menu

The **Options** pop-up menu lists the option sets established for a project—for example, a development build options set or a ship build options set. To make an options set active, choose its name from the **Options** pop-up menu (Figure 17-3). For more information on options sets, see Chapter 18, "The Project Menu," and Chapter 8, "Advanced Topics."

Figure 17-3 Options pop-up menu

Project window column headings

Each column heading provides a specific kind of information regarding the project (Figure 17-4). This section describes the column headings in order, from left to right.



✓ MiniEdit.π

Headers ▼ Options Ship build ▼

	✓	Name	🐞	Group	Translator	Kind	Location	Modification	Code	Data
	◆	BuggyEdit.c	◆		PowerPC C		Macintosh HD…:MiniEdit ƒ:	1/26/95 6:17 PM	2912	232
▷		**Libraries**				**Group**			**2120**	**121**
	◇	mini.file.c	◆		PowerPC C		Macintosh HD…:MiniEdit ƒ:	8/30/94 4:53 PM	2544	328
	◇	mini.print.c	◆		PowerPC C		Macintosh HD…:MiniEdit ƒ:	8/30/94 5:00 PM	1304	88
	◇	mini.windows.c	◆		PowerPC C		Macintosh HD…:MiniEdit ƒ:	9/02/94 12:15 PM	2424	160
	◇	MiniEdit.rsrc			Resource Copier		Macintosh HD…:MiniEdit ƒ:	3/22/94 7:57 PM	0	0
	◇	pleasewait.c	◆		PowerPC C		Macintosh HD…:MiniEdit ƒ:	3/22/94 7:57 PM	40	16
		Totals							**11344**	**945**

Figure 17-4 Project window column headings

Make status

The Make status column indicates whether a source file needs to be rebuilt. A filled diamond indicates that the project entry is not up-to-date; an empty diamond indicates that the project entry is up-to-date.

Icon

The Icon column displays the Finder icon for each source file or the folder icon for each group.

Name

The Name column displays the name of each file or group in the project. The name is always displayed.

Debug status

The Debug status column shows whether the debug status for each source file is on or off. A filled diamond (under the bug icon) indicates that debugging information is generated for the file; an empty diamond indicates that debugging information is not generated for the file.

Group

The Group column displays the project group hierarchy.

The Group column shows the immediate group owner of each source file. This option is useful when the Group Hierarchy option on the Project Window page of the **Project Options** dialog box is set off.

Translator
The Translator column shows the name of the translator that is used for each source file.

Kind
The Kind column shows the kind of each entry: Group, Source, Precompile Source, Project, Library (hard import), Library (soft import).

Location
The Location column shows the full path of each source file.

Modification
The Modification column shows the last modification date of each source file.

Code
The Code column shows the code size of each file in bytes.

Data
The Data column shows each file's data contribution in bytes.

Projector status
The Projector status column shows the projector checkout status of each source file. If the pencil icon has a line through it, the source file is read-only; no line, the file is modifiable; and a dashed line, the user has selected the **Modify Read Only** command from the **Revision** menu. That command allows you to change the file, but you cannot check the changed file back in.

Selecting Project Entries

Any of the column headings can be used as a criterion for sorting the files or groups listed in the window. You can choose, for example, to sort by file name (by selecting the Name heading) or by modification date (by selecting the Modification heading). Depending on the field, the sort would display the project entries in alphabetic or numeric order.

To select a heading as a sort criterion, click it. Once the heading is selected, a directional arrow is displayed adjacent to the heading or icon (Figure 17-5). Note that the project files listed in Figure 17-4 have been sorted by name. The directional arrow next to the Name column heading shows that the alphabetic sort ordered the filenames from A to Z.

Click the arrow next to a selected column heading to reverse the sort (from ascending to descending, for example).

Figure 17-5 Name heading with directional arrow

Drag-and-Drop Operations

In the Project window, you can use a number of drag-and-drop operations. You can, for example, add, remove, arrange, or copy project entries and groups by moving entries in, or between, Project windows.

Adding project entries from the Finder

To add project entries and groups from the Finder, select them in the Finder and drag their outlines into a Project window. You can only add files that have the correct file extensions (or are shared libraries or projects) and that are not already in the project.

If any of the selected files cannot be added, the entire drag is refused (the window does not draw the drag highlight). If you do not want to do type checking (for example, you may want to add some documentation files), hold down the Command key as you drag items into the Project window. This has the same effect as selecting **All Files** from the **Show** pop-up menu in the **Add Files** dialog box.

If you drag a folder from the Finder, a group is created with the name of the folder and its contents are added to the group. By dragging folders, you can create a group hierarchy that reflects your source code hierarchy.

Removing project entries

You can remove project entries and groups. For example, you can select a number of project entries and groups and drag them to the trash icon on the desktop.

Rearranging project entries

You can rearrange project entries and groups. For example, you can select a number of project entries and groups and drag them to other group locations in the Project window. The group destination is highlighted to show where the dragged items will go. Note that you cannot reorder items within a group by dragging because items are always sorted.

Copying project entries

To copy project entries and groups to other projects, first select them, then drag them to another Project window. This adds these project entries and groups at the drop location (if you dropped on a group entry). Note that the project entries and groups are not removed from the project from which you dragged them.

The Project Menu

18

This chapter describes how you control project behavior. This includes options for building and running an application, for configuring applications and libraries, for linking, for compiling, and for project display.

Commands in the Project Menu

The commands in the **Project** menu are used to perform the following functions:

- Switch projects
- Add and remove files from a project
- Work with the Debugger
- Set project-level options

Figure 18-1 shows the **Project** menu. This chapter discusses the **Project** menu commands by function in the order listed above. Because of the complexity of the **Options** command, establishing project-level options is covered last.

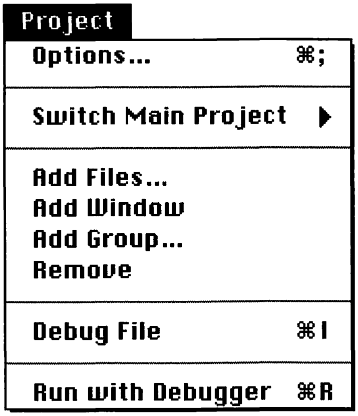


Figure 18-1 Project menu

Switching projects

One command lets you switch to any open project or any project that you just closed.

Switch Main Project

Opens a submenu containing the names of all projects that have been opened during the current session, even if they are closed (Figure 18-2). This menu also contains projects that have aliases in the `(Projects)` folder in the Symantec tree. You use this submenu to choose a different project as the target of a build command.

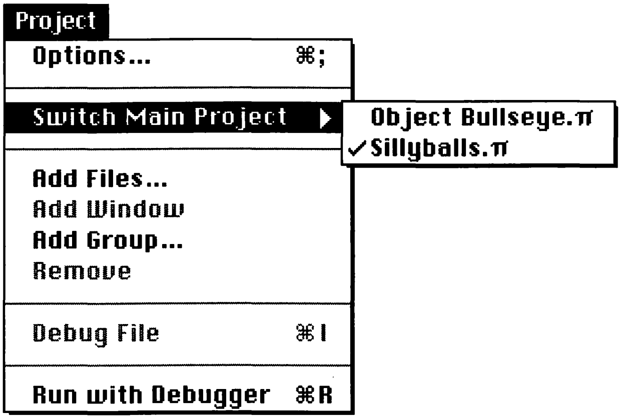


Figure 18-2 Switch Main Project submenu

Adding and removing files from a project

The following four menu commands let you add and delete files from a project.

Add Files

Opens a dialog box that lists all extension-mapped files. You use this dialog box to add files to the Project window, in the selected group (if any). You can add any type of file to a project; only those files that map to translators will be built. For example, you can add word processing files without tacking on extensions. To show all files, select **All Files** from the **Show** pop-up menu.

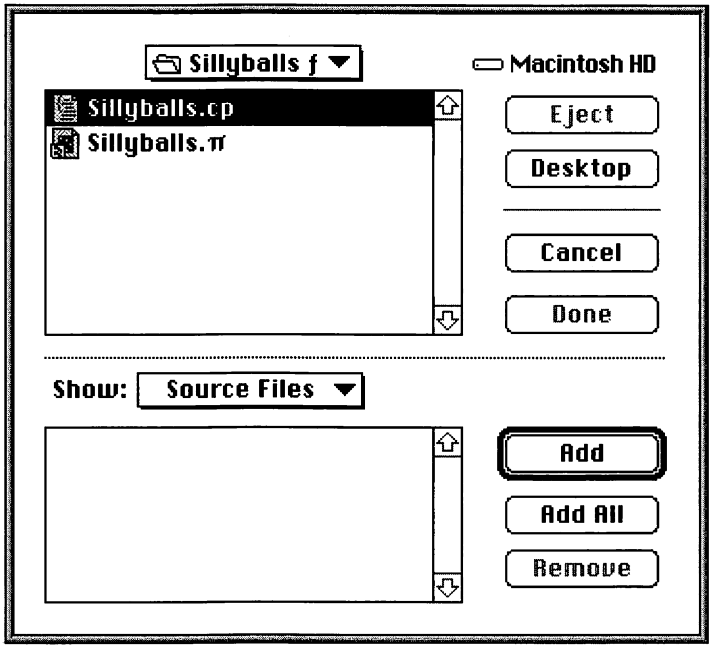


Figure 18-3 Add File dialog box

Navigate to a file that you want to add, click to select it, then click Add. When you have added all the files you want, click Done.

Add *filename*

Adds the file that is open in the current Editor window to the main project.

Add Group

Creates a new group in the project within the currently selected group. Group names must be unique within the same enclosing group.

Remove *filename*

Removes selected files or groups from the project. Multiple files or groups can be selected.

Working with the Debugger

The next two commands describe how to launch the Debugger and how to run an application with the Debugger on.

Debug File ⌘I

Launches the Debugger, if necessary, and opens the selected file for debugging. In the Debugger, the Main window opens with the same line selected that you selected earlier in the Editor window.

Run with Debugger
Run ⌘R

Builds a version of an application without linker optimizations and runs it or sends it to the Debugger. If changes have been made since the last time you ran the project, you are prompted to update the project. You can toggle between **Run** and **Run with Debugger** mode by holding down the Option key while clicking this command.

Setting project-level options

The most complex command in the **Project** menu is **Options**. Choosing this command opens a multipage dialog box.

Option ⌘;

Opens the **Project Options** dialog box, in which you configure the options settings for the current project (or if no project is current, a project that you select). Options are project specific, unlike the preferences you set using the **Project Manager Preferences** dialog box. Options settings remain bound to a project even when you close and later reopen the project. They control general project behavior, including build and run settings.

Note

> The preferences in the **Project Manager Preferences** dialog box affect all projects.

The **Project Options** dialog box has eight options pages:

- Project Options
- Project Type
- Linker
- Extensions Mapping
- Project Window
- PowerPC C Compiler Options
- PowerPC C++ Compiler Options
- Symantec Rez Compiler Options

Each options page is represented by an icon in the scrolling list at the left side of the dialog box. By default, the **Project Options** dialog box opens to the Project Options page. Clicking a page's icon selects the page. Some options pages, such as Project Type, have multiple subpages. You can move freely between options pages as you configure options. Table 18-1 shows the keyboard substitutes for moving in the icon list.

Table 18-1 Keyboard substitutes for page icons

Press this	To select
Command-Option-Up Arrow	Preceding icon
Command-Option-Down Arrow	Following icon
Command-Option-Page Up	Preceding icon
Command-Option-Page Down	Following icon
Command-Option-Home	First icon
Command-Option-End	Last icon

Each options page contains Cancel, Save, and Factory Settings buttons. Cancel performs its standard Macintosh operation. Clicking Save saves the options settings for the current editing session. These options settings are then available for selection—for example, in another editing session or from the Project window. The Factory Settings button resets all options to their default settings.

Other common elements on all options pages include the Help area and the **Options** pop-up menu.

Help

The Help area provides information for each option displayed. To view the information for an option, click the option.

Note

> To receive help on an option without changing its value, move the cursor away from the option before releasing the mouse.

Options pop-up menu

The Symantec Project Manager allows you to define multiple options sets for each project. These sets can be used to group various option configurations. For example, you can define options sets for the different stages of a project's development (such as development build, beta build, and ship build).

By default, each project has one options set that is created when the project is created; this options set has the same name as the project. Note that any changes made on any of the five pages of the **Project Options** dialog box are made in relation to the options set currently selected in this pop-up menu.

Clicking Options opens the **Options** pop-up menu.

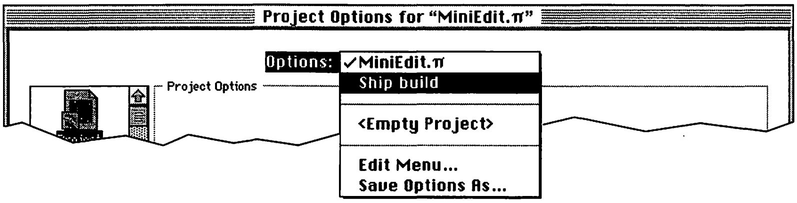


Figure 18-4 Options pop-up menu

You use the pop-up menu to create and select options sets. As you define new options sets, their names are added to the project.

Note

> Any changes you make to options sets through the **Options** pop-up menu affect only the current editing session. To save those changes, click Save. To switch options sets for the project for later use, use the **Options** pop-up menu in the Project window.

<Empty Project>

When a project is created with the Empty Project template, its default options set contains the options defined in the <Empty Project> options set. You can change the settings in this options set by choosing **<Empty Project>** from the **Options** pop-up menu, configuring the options pages appropriately, and clicking Save.

Edit Menu

Opens the **Edit Options Menu** dialog box, in which you can select an options set to delete or rename (Figure 18-5).

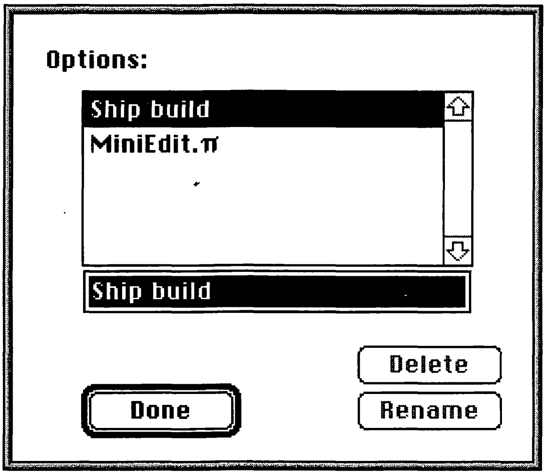


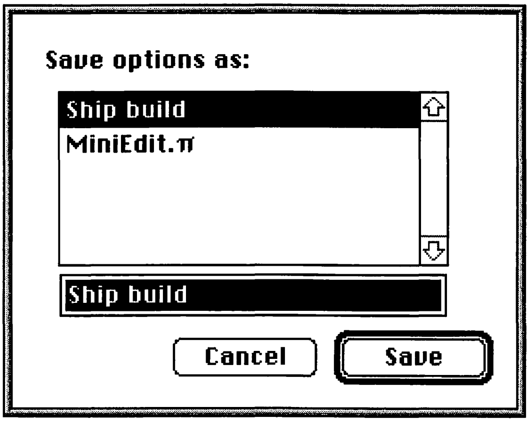
Figure 18-5 Edit Options Menu dialog box

To delete an options set, select its name from the list and click Delete. This removes the options set for the current editing session only. To delete the options set permanently, click Save after you have deleted the options set and closed the dialog box. Note that you cannot delete the options set currently selected in the Project window.

To rename an options set, select its name from the list, type the new name in the textbox, and click Rename. This renames the options set for the current session. To permanently save the new name, click Save after you have renamed the option and closed the dialog box.

Save Options As

Opens an **Options Save** dialog box, in which you can create a new options set from an existing one. This is the last command on the **Options** pop-up menu (Figure 18-6).



Save options as:

Ship build

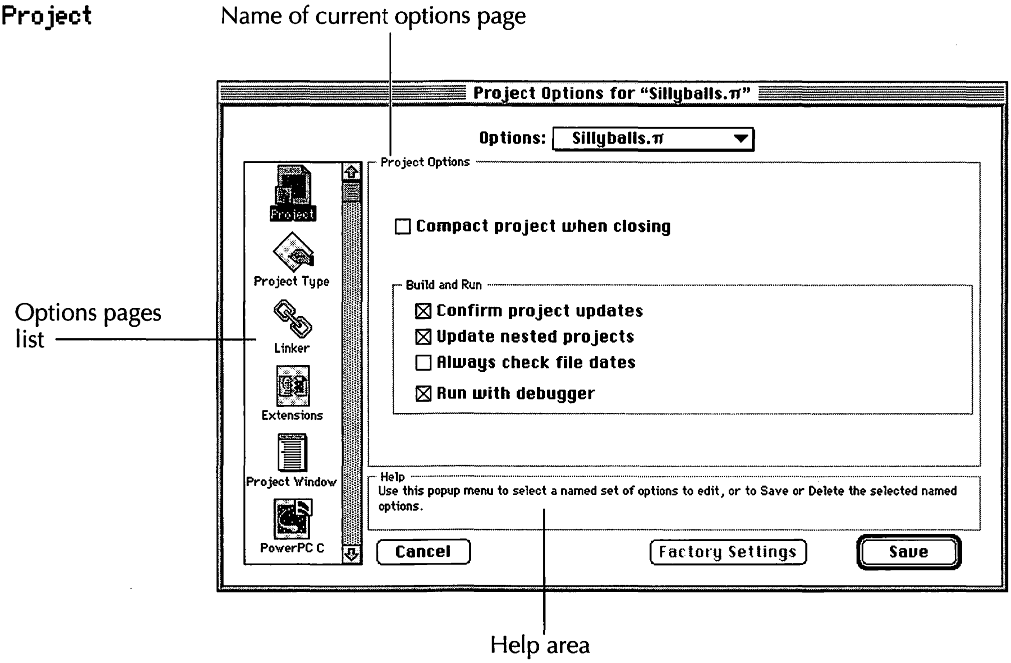
Figure 18-6 Options Save dialog box

The scrolling pane lists the names of current options sets. You select an options set to copy, then type a new name in the textbox. Figure 18-6 shows the creation of a ship build setting. Remember to click Save on the options page after you close the dialog box to permanently save the options set you just created.

Project Options Page

Project

Clicking the Project Options page icon displays a set of options for building and running a project (see Figure 18-7).



Project Options for "Sillyballs.π"

Options: Sillyballs.π

Project Options

☐ Compact project when closing

Build and Run

☒ Confirm project updates
☒ Update nested projects
☐ Always check file dates
☒ Run with debugger

Help

Use this popup menu to select a named set of options to edit, or to Save or Delete the selected named options.

Figure 18-7 Project options page

Compact Project When Closing

Setting on the option Compact Project When Closing causes the Project Manager to compact the project whenever you close it.

Build and Run settings

There are four options in this group.

Confirm Project Updates

When you try to run or build a project, you are prompted to make any necessary updates. If this option is not set on, the project is updated without a confirmation prompt.

Update Nested Projects

Whenever the project is brought up-to-date, the Project Manager builds the target from all included projects.

Always Check File Dates
Whenever you try to run or build a project, the Project Manager checks the modification date for each file in the project. Outdated files are flagged for compilation.

Run with Debugger
When you select **Run**, the Project Manager launches the application through the Symantec Debugger.

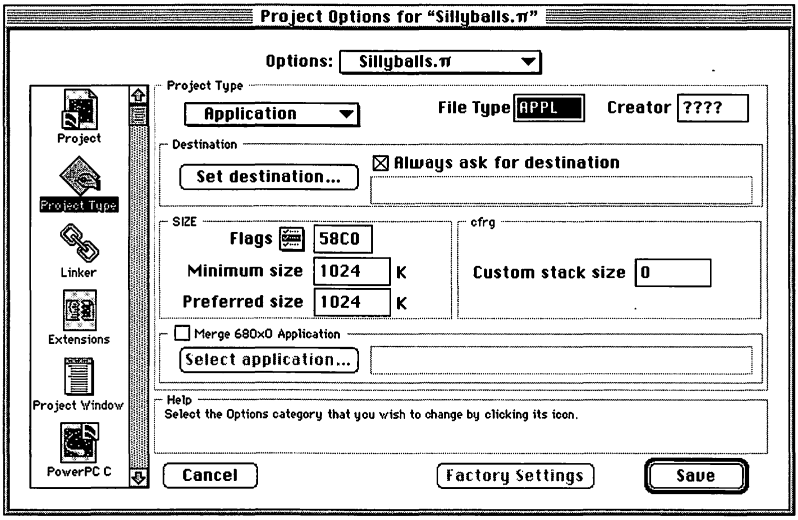
Project Type Page

Project Type

Clicking the Project Type page icon displays a set of options for configuring various aspects of the target application or library produced from a project. The Project Type page has three subpages, one for each entry in the **Project Type** pop-up menu.

Application subpage

Choosing **Application** from the **Project Type** pop-up menu specifies that the target is an application. The Application subpage lets you configure options for that application (Figure 18-8).



Project Options for "Sillyballs.π"

Options: Sillyballs.π

Project Type: Application

File Type: APPL

Creator: ????

Destination

☒ Always ask for destination

SIZE

Flags: 58C0

Minimum size: 1024 K

Preferred size: 1024 K

cfrg

Custom stack size: 0

□ Merge 680x0 Application

Help
Select the Options category that you wish to change by clicking its icon.

Figure 18-8 Application subpage of the Project Type page

Application identifiers
These textboxes specify the file type and creator of the application file you are creating.

File Type. Displays the file type ID for the application file. By default, the ID is set to `'APPL'`.

Creator. Displays the creator ID for the application file. Type the four-character ID in the textbox.

Destination

These options govern whether your target application has a default location associated with it.

Set Destination. Opens a standard **File Save** dialog box, in which you define the default location for the target file application. Once you specify the target's filename and folder, its name is displayed in the textbox below the Always Ask for Destination check box.

Always Ask for Destination. Causes the project system to prompt you for the destination of the application file each time the application is built.

SIZE

The SIZE options let you set the bits of the SIZE resource, which specify the system services your application uses or is compatible with (Figure 18-9). You can either use the pop-up menu to set the bits individually or type a hexadecimal value in the textbox.

Flags. Opens the **SIZE Flags** pop-up menu. Each entry in the menu is a flag, or bit, that can be set on or off. For more information, see the SIZE resource page in the *THINK Reference.*

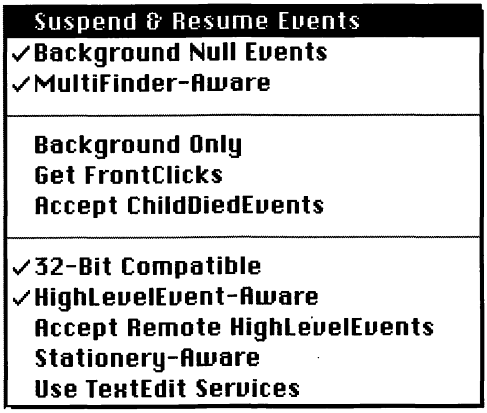


Figure 18-9 SIZE Flags pop-up menu

The first three flags indicate the compatibility of the application with System 7 and MultiFinder.

Suspend & Resume Events

Specifies that the application gets these events as it shifts from the foreground to the background layers, in addition to the Activate and Deactivate events it normally receives.

Background Null Events

Specifies that the application gets regular Null events when it is in the background. Otherwise, the application gets only Update events. Note that the Debugger can only debug applications that have this flag set.

MultiFinder-Aware

Indicates that the Finder expects you to conform to the guidelines for shifting from the foreground to the background layers. The application gets Suspend and Resume events as it shifts from foreground to background, but it does not get Activate/Deactivate events. If you check this flag, you should also check the Suspend & Resume flag.

The next three flags indicate those extra features of System 7 and MultiFinder that an application uses.

Background Only

Specifies that the application runs only in the background. Set this flag if the application has no interface and cannot run in the foreground. Note, however, that because the Debugger cannot debug background-only applications, you should turn off this flag while you are debugging this kind of application.

Get FrontClicks

Specifies that the application receives the Mouse-Down and Mouse-Up events that bring the application to the foreground.

Accept ChildDiedEvents

Specifies that the application is notified when an application that it launched quit or crashed.

The last five flags indicate to the Finder which System 7 features an application supports.

32-Bit Compatible

Sets the application to run under the 32-bit version of Memory Manager. Do not set this flag unless you have tested the application on a 32-bit system.

HighLevelEvent-Aware

Sets the application to receive all high-level events, including Apple events, when it calls `WaitNextEvent`.

Accept Remote HighLevelEvents

Specifies that the application receives high-level events, including Apple events, from your computer and other computers on your network. If this flag is not set, the application receives high-level events from your computer only.

Stationary-Aware

Indicates that the Finder expects the application to handle stationary documents. If this flag is not set, the Finder handles stationary documents for you by duplicating the document and asking the user for a name for the duplicate.

Use TextEdit Services

Indicates that your application can use the inline text services that TextEdit provides for multi-byte script systems (such as those for Japan and China). See *Inside Macintosh VI* for more information about these services.

The remaining options on the Application subpage of the Project Type page include the following check boxes:

Minimum Size. Lets you define the minimum memory partition to allocate to an application.

Preferred Size. Lets you define the application's default memory partition.

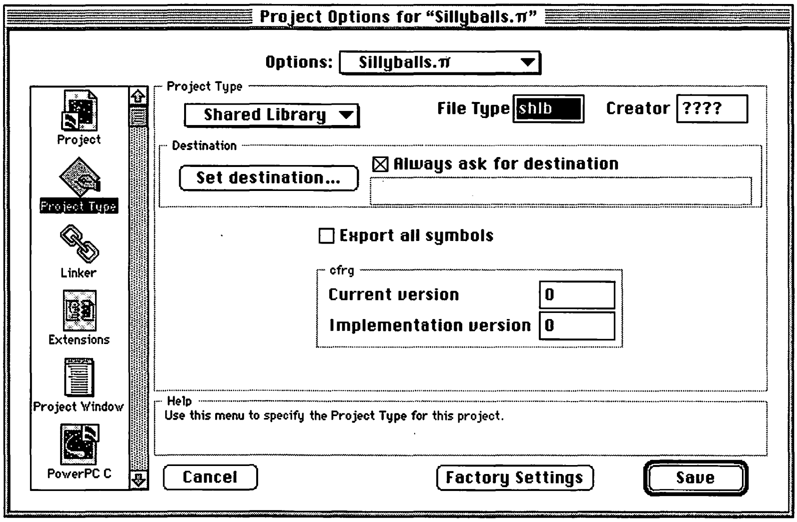
cfrg resource setting
You use this setting to specify the application's cfrg resource settings. This resource type describes characteristics of code fragments on the Power Macintosh. See *Inside Macintosh: PowerPC System Software* for more information.

Custom Stack Size. Allows you to change the stack size in bytes for an application. A value of 0 indicates that the default size should be used. The default size varies depending on the machine configuration; see *Inside Macintosh: PowerPC System Software* for more information on default stack sizes.

Merge 680x0 Application
To create a "fat app" that contains both PowerPC and 680x0 code, check this option and then click Select Application. A standard **Open** dialog box opens, in which you specify the 68K application whose code is to be merged into the target application. Once you specify an application, its name is shown to the right of the button.

Shared Library subpage

Choosing **Shared Library** from the **Project Type** pop-up menu specifies that the project target is to be a shared library. This subpage (see Figure 18-10) lets you configure options for a shared library.



Project Options for “Sillyballs.π”

Options: Sillyballs.π

Project Type: Shared Library

File Type shlb

Creator ????

☒ Always ask for destination

□ Export all symbols

Current version 0

Implementation version 0

Help
Use this menu to specify the Project Type for this project.

Figure 18-10 Shared Library subpage of the Project Type page

Shared library identifiers
These textboxes specify the file type and creator of the shared libraries you are creating.

File Type. Displays the file type ID for the shared library file. By default, the ID is set to `'shlb'`.

Creator. Displays the creator ID for the shared library file. Type the four-character ID into the textbox.

Destination
These options govern whether the target library has a default location associated with it.

Set Destination. Opens a standard **File Save** dialog box in which you can define the default location for the target library. Once you specify the library's filename and its folder, the library's name is shown in the textbox below the Always Ask for Destination check box.

Always Ask for Destination. Causes the project system to always prompt you for the destination of the library each time the shared library is built.

Export All Symbols
Setting this option on includes all nonstatic symbols in the export section of the shared library. Otherwise, only those symbols explicitly compiled as exports are included (see the `#pragma lib_export` in the *Symantec C++ Compiler Guide*).

cfrg resource setting
You use this option to specify the shared library's cfrg resource settings. This resource type describes characteristics of code fragments on the Power Macintosh. See *Inside Macintosh: PowerPC System Software* for more information.

Current Version. Lets you specify the current version of the shared library. For information about this topic, see *Inside Macintosh: PowerPC System Software.*

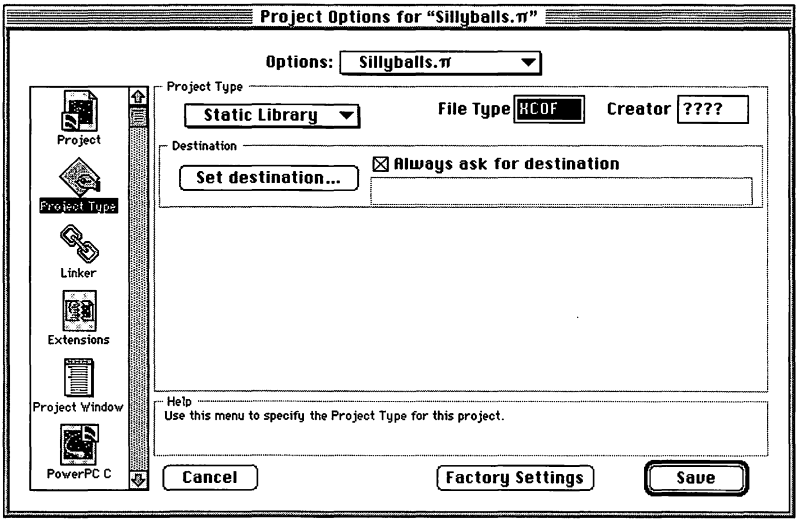
Implementation Version. Lets you specify the shared library's oldest supported implementation version.

Static Library subpage

Choosing **Static Library** from the **Project Type** pop-up menu specifies that the project target is to be a static library. The subpage that opens (Figure 18-11) lets you configure options for a static library.

Note

To create static libraries, you must use PPCLINK & MakePEF.



Project Options for “Sillyballs.π”

Options: Sillyballs.π

Project Type: Static Library

File Type XCOF

Creator ????

Destination

☒ Always ask for destination

Help

Use this menu to specify the Project Type for this project.

Figure 18-11 Static Library subpage of the Project Type page

Static library identifiers
These textboxes specify the file type and creator of the static libraries you are creating.

File Type. Defines the file type ID for the static library file. By default, the ID is set to 'XCOF'.

Creator. Defines the creator ID for the static library file. Type the four-character ID into the textbox.

Destination
These options govern whether the target library has a default location associated with it.

Set Destination. Opens a standard **File Save** dialog box in which you can define the default location for the target library. Once you specify the target library's filename and folder, its name is displayed in the textbox below the Always Ask for Destination check box.

Always Ask for Destination. Causes the project system to always prompt you for the destination of the library each time the static library is built.

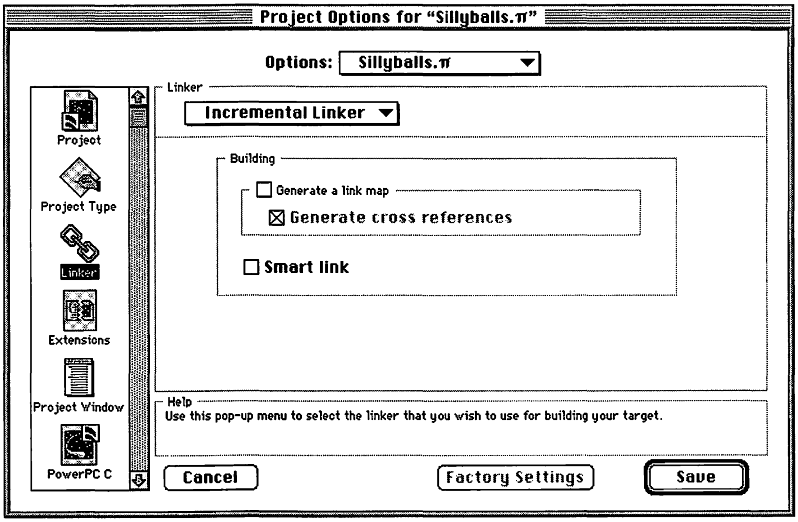
Linker

Linker Page

Clicking the Linker page icon displays a set of options for configuring how the Symantec Project Manager builds your target. The Linker page has two subpages, one for each entry in the **Linker** pop-up menu.

Incremental Linker subpage

This subpage lets you choose options for the Incremental Linker (Figure 18-12). Choosing **Incremental Linker** from the **Linker** pop-up menu specifies that the Symantec Incremental Linker is to be used to link your application's code.



Project Options for "Sillyballs.π"

Options: Sillyballs.π

Linker: Incremental Linker

Building

☐ Generate a link map

☒ Generate cross references

☐ Smart link

Help

Use this pop-up menu to select the linker that you wish to use for building your target.

Figure 18-12 Incremental Linker subpage of the Linker page

Generate a Link Map

Setting Generate a Link Map causes the Symantec Incremental Linker, during a project build, to generate a link map for the target application.

Generate Cross References. Causes the linker to put cross-reference information in the link map. Generate a Link Map must be set before changing the setting on this option. You can then turn Generate a Link Map off if you want.

Smart Link

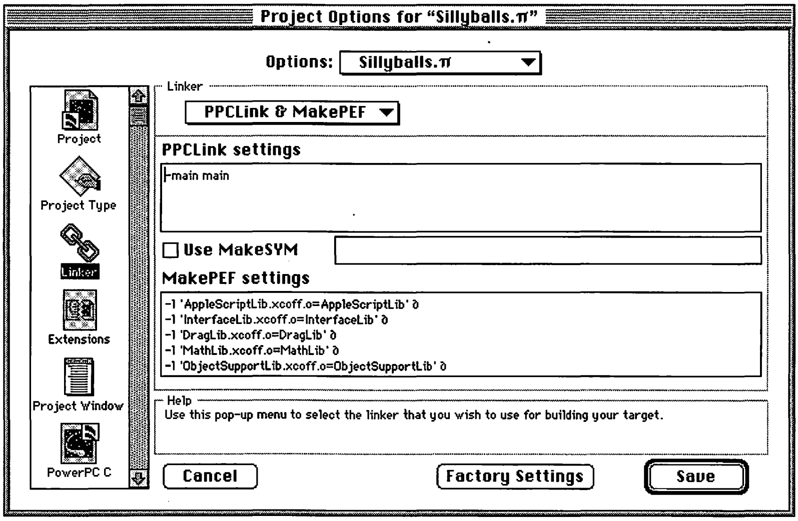
Setting Smart Link causes the Symantec Incremental Linker, during a project build, to remove unreferenced code from the target application.

Note

> Each of the options Generate a Link Map, Generate Cross References, and Smart Link can be turned on independently of the other options.

PPCLink & MakePEF subpage

Choosing **PPCLink & MakePEF** from the **Linker** pop-up menu specifies that PPCLink & MakePEF are to be used to link your application's code (Figure 18-13).



Options: Sillyballs.π

Linker: PPCLink & MakePEF

PPCLink settings

```
-main main
```

□ Use MakeSYM

MakePEF settings

```
-l 'AppleScriptLib.xcoff.o=AppleScriptLib' ∂
-l 'InterfaceLib.xcoff.o=InterfaceLib' ∂
-l 'DragLib.xcoff.o=DragLib' ∂
-l 'MathLib.xcoff.o=MathLib' ∂
-l 'ObjectSupportLib.xcoff.o=ObjectSupportLib' ∂
```

Help: Use this pop-up menu to select the linker that you wish to use for building your target.

Figure 18-13 PPCLink & MakePEF subpage of the Linker page

Note

> To use these tools, you need to have placed an alias for Toolserver into your (`Tools`) folder at installation. (See the section "Using ToolServer" in Chapter 8, "Advanced Topics.")

PPCLink Settings

Use this text field to type the PPCLink options you want to send PPCLink.

Use MakeSYM

Setting this option on causes the linker to create a `.SYM` file for the PPCLink output file.

Use the textbox to the right of the Use MakeSYM check box to type the options that you want to provide MakeSYM.

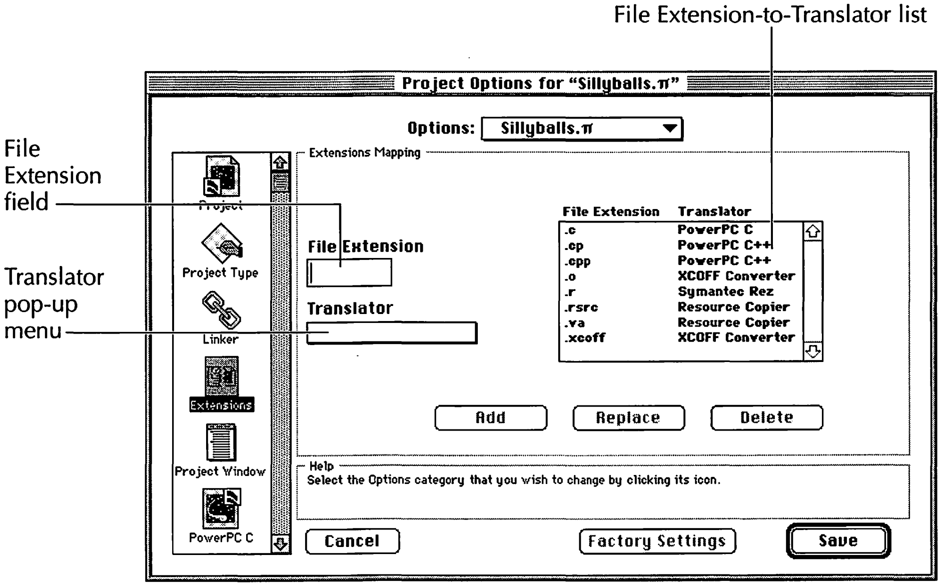
MakePEF Settings

Use this edit area to type in the MakePEF options you want to send MakePEF.

Extensions Mapping Page

Extensions

Clicking the Extensions icon provides access to the extensions mapping options, including a list of recognized file extensions (Figure 18-14).



File Extension-to-Translator list

File Extension field

Translator pop-up menu

File Extension	Translator
.c	PowerPC C
.cp	PowerPC C++
.cpp	PowerPC C++
.o	XCOFF Converter
.r	Symantec Rez
.rsrc	Resource Copier
.va	Resource Copier
.xcoff	XCOFF Converter

Help
Select the Options category that you wish to change by clicking its icon.

Figure 18-14 Extensions Mapping page

When you build a project, individual project entries are processed differently at build time according to their type. Each project maintains a table that determines how each type of project entry is handled when the project is built. Using this page, you can redefine default extensions mappings, or define new extensions and the tools used to translate them.

File Extension
Use this field to type a file extension to add to the extensions list.

Translator
Use this menu to associate a translator with a file extension that you are adding.

File Extension/Translator list
This scrolling list contains all the recognized file extensions. As long as your project files have any of these extensions, the project will compile. You can add to the File Extension/Translator list.

Add, Replace, and Delete buttons
These buttons let you add, replace, or delete associations between file extensions and translators.

Add. Adds to the File Extension/Translator list the association between a file extension and translator that you have chosen.

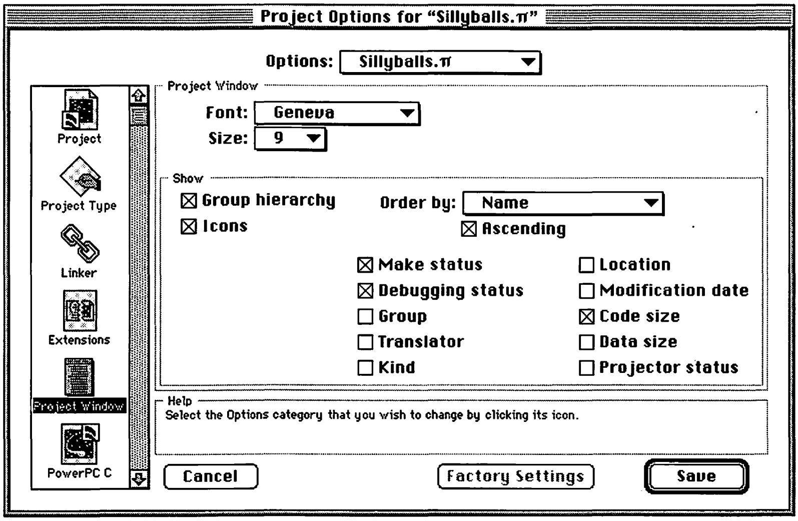
Replace. Deletes and adds an association between a file extension and a translator in one step. Before clicking this button, you must choose the association to delete and specify the one to add.

Delete. Deletes from the File Extension/Translator list the association you have chosen.

Project Window Page

Project Window

Clicking the Project Window page icon displays options that control the contents of the Project window and how those contents are displayed (Figure 18-15).



Project Options for "Sillyballs.π"

Options: Sillyballs.π

Project Window

Font: Geneva

Size: 9

Show

☒ Group hierarchy

Order by: Name

☒ Icons

☒ Ascending

☒ Make status

☒ Debugging status

□ Group

□ Translator

□ Kind

□ Location

□ Modification date

☒ Code size

□ Data size

□ Projector status

Help

Select the Options category that you wish to change by clicking its icon.

Figure 18-15 Project Window page

Typeface options

These options identify the font and type size that will be used.

Font

Clicking the arrow opens a pop-up menu for changing the font.

Size

Clicking the arrow opens a pop-up menu for changing font size.

Show options

These options establish those columns that appear in the application's Project window and how the entries are displayed in them. Most of these options represent column headings on the window.

Group Hierarchy

Setting this option on includes a column for displaying the project group hierarchy in the Project window.

Icons
Setting this option on includes an icons column for displaying the Finder icon for each source file or the folder icon for each group.

Order by
This pop-up menu determines the order in which files are displayed in the Project window. Your selection in this menu serves as the sort criterion.

Ascending
Setting this option on causes the items selected for display through the **Order by** pop-up menu to be displayed in ascending rather than descending order.

For example, in Figure 18-15, Name is selected in the **Order by** pop-up menu as the sort criterion for displaying project files. Because this is an alphabetic attribute, setting the Ascending option on sorts the project files from A to Z; otherwise, the files would be sorted from Z to A. For a numeric attribute, an ascending sort would display items from the smallest to the largest numbers.

Make Status
Setting the Make Status option on includes a column used to indicate whether a source file needs to be rebuilt. In the Project window, a filled diamond next to the file's name indicates that the project entry is not up-to-date; an empty diamond indicates the project entry is up-to-date.

Debugging Status
Setting this option on displays the bug column in the Project window. In the window, a filled diamond (under the bug icon) indicates that debugging information is generated for the file; an empty diamond indicates that debugging information is not generated for the file.

Group
Setting this option on shows the immediate group owner of each source file in the window. This option is useful when the Group Hierarchy option is set off.

Translator
Setting this option on includes a Translator column in the window, which is used to show the name of the translator for each source file.

Kind
Setting this option on includes a Kind column in the window, which is used to show the kind of each entry: Group, Source, Precompile Source, Project, Library (hard import), and Library (soft import).

Location
Setting this option on includes a Location column in the window, which is used to show the full path of each source file.

Modification Date
Setting this option on includes a Modification column in the window, which is used to show the last modification date of each source file.

Code Size
The Code column, if you include it in the window, shows the code size of each file in bytes.

Data Size
The Data column, if you include it in the window, shows each file's data contribution in bytes.

Projector Status
The Projector Status column, if you include it in the window, shows the projector checkout status of each source file.

PowerPC C Options Page

PowerPC C

Click the PowerPC C icon to select the PowerPC C options page. There are six basic sets of PowerPC C options: language settings, compiler settings, code optimization, debugging, warning messages, and prefix settings. See the *Symantec C++ Compiler Guide* for further information regarding each of these sets of options.

PowerPC C++ Options Page

PowerPC C++

Click the PowerPC C++ icon to select the PowerPC C++ options page. There are six basic sets of PowerPC C++ options: language settings, compiler settings, code optimization, debugging, warning messages, and prefix settings. See the *Symantec C++ Compiler Guide* for further information regarding each of these sets of options.

Symantec Rez Options Page

Symantec Rez

The Symantec Rez options page lets you configure the Symantec Rez resource compiler. See the *Symantec C++ Compiler Guide* for further information.

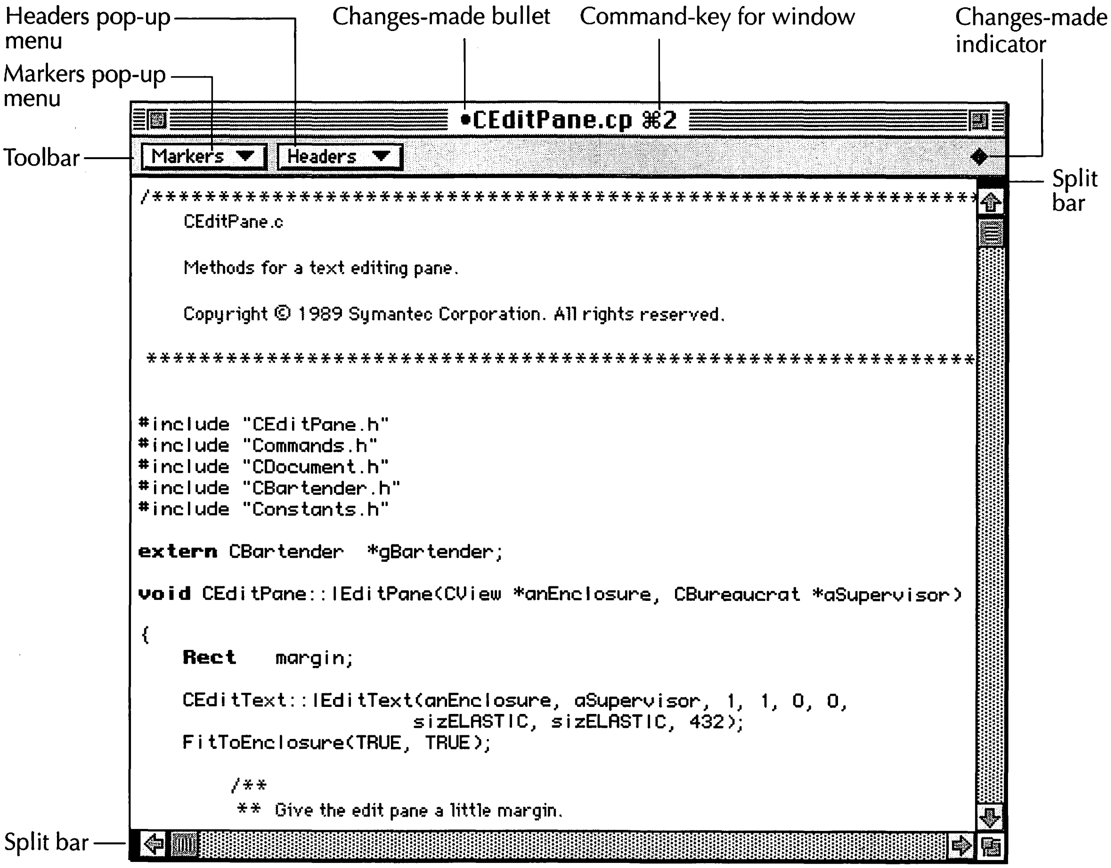
The Editor Window ◆ 19

This reference chapter provides a detailed description of the Editor window. The Symantec Editor offers integrated full-featured editing. You can set preferences for the Editor window using the **Edit** menu, described in Chapter 26, "The Windows Menu." Search commands are located in the Editor's **Search** menu, which is described in Chapter 21, "The Search Menu."

Introducing the Editor Window

The Editor window has all the functionality of the THINK Editor's editing window, plus new features such as syntax highlighting, auto-indenting, delimiter matching, **Markers** and **Headers** pop-up menus, and the ability to split a window into subpanes. An Editor window opens when you open a source or other text file.

Figure 19-1 shows an example of an Editor window.



```
/*****************************************************************
	CEditPane.c

	Methods for a text editing pane.

	Copyright © 1989 Symantec Corporation. All rights reserved.

 *****************************************************************


#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)

{
	Rect	margin;

	CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
						sizELASTIC, sizELASTIC, 432);
	FitToEnclosure(TRUE, TRUE);

		/**
		 ** Give the edit pane a little margin.
```

Figure 19-1 Editor window

The features of the Editor window can be divided into window features and text features. Examples of window features include closing windows, thumb scrolling, and editor independence. Examples of text features include syntax highlighting and auto-indenting. Window features are discussed first in this chapter.

Window features

There are ten window features of the Editor window.

Changes-made, Command-key, and file path indicators

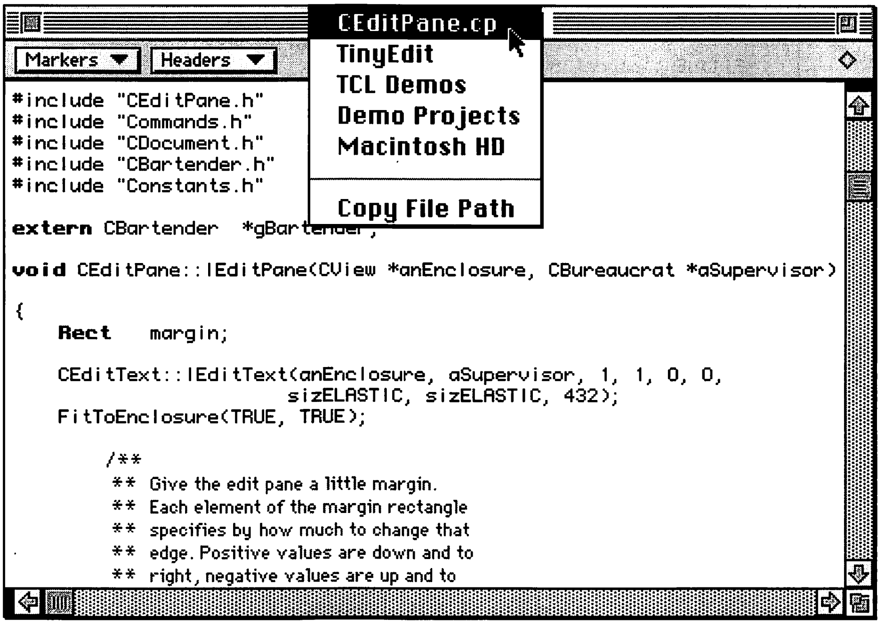
In the window's title box, you can choose to display the following:

- A bullet (•) before the filename if the contents of the file have been changed
- The Command-key used to bring the window to the front (using the key combination has the same effect as selecting the Editor window's name from the **Windows** menu)
- The file's path

To enable these features, use the **Preferences** command from the **Edit** menu. For details, see Chapter 20, "The Edit Menu."

File Path pop-up menu

The **File Path** pop-up menu shows the path for the file (similar to the Finder), as shown in Figure 19-2.



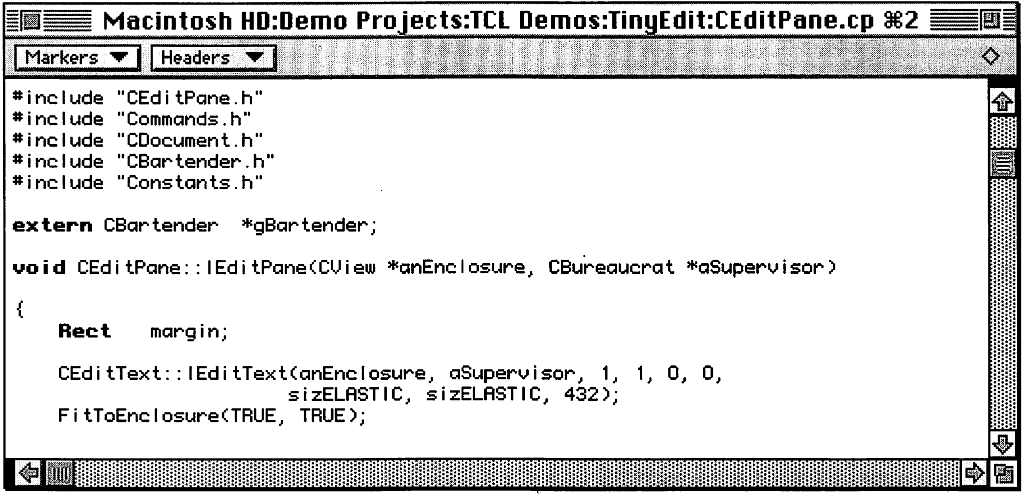
```
#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender   *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)

{
    Rect   margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                         sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little margin.
         ** Each element of the margin rectangle
         ** specifies by how much to change that
         ** edge. Positive values are down and to
         ** right, negative values are up and to
```

Figure 19-2 File Path pop-up menu

You access the **File Path** pop-up menu by Command-clicking the title in the title bar. Selecting the bottom menu item, **Copy File Path**, copies the full path to the Clipboard.

If the preference Show Full Path is set in the Editor Options General Settings page of the **Project Manager Preferences** dialog box, the file path is displayed in the title bar, as shown in Figure 19-3. For details, see Chapter 20, "The Edit Menu."

File path in title bar


Macintosh HD:Demo Projects:TCL Demos:TinyEdit:CEditPane.cp ⌘2
Markers ▼
Headers ▼

```
#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)
{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                         sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);
```

Figure 19-3 File path in title bar

Scroll rate control

You have some control over the rate at which scrolling occurs in a window. Holding down the Option key while scrolling doubles the scroll rate. Holding down the Command key while scrolling triples the scroll rate. Holding down both the Command and Option keys while scrolling multiplies the scroll rate by six. This also works when auto-scrolling, when you are dragging to extend a selection.

Live thumb scrolling

To prevent jerky scrolling, the Editor smoothes the scrolling motion as you drag the thumb. Text scrolls continuously, rather than snapping into place when you release the mouse button. To disable the smoothing, hold down the Command key; to disable the live scrolling, hold down the Option key before clicking on the thumb.

To return to the point from which you started live thumb scrolling, hold down the Option key and release the mouse button.

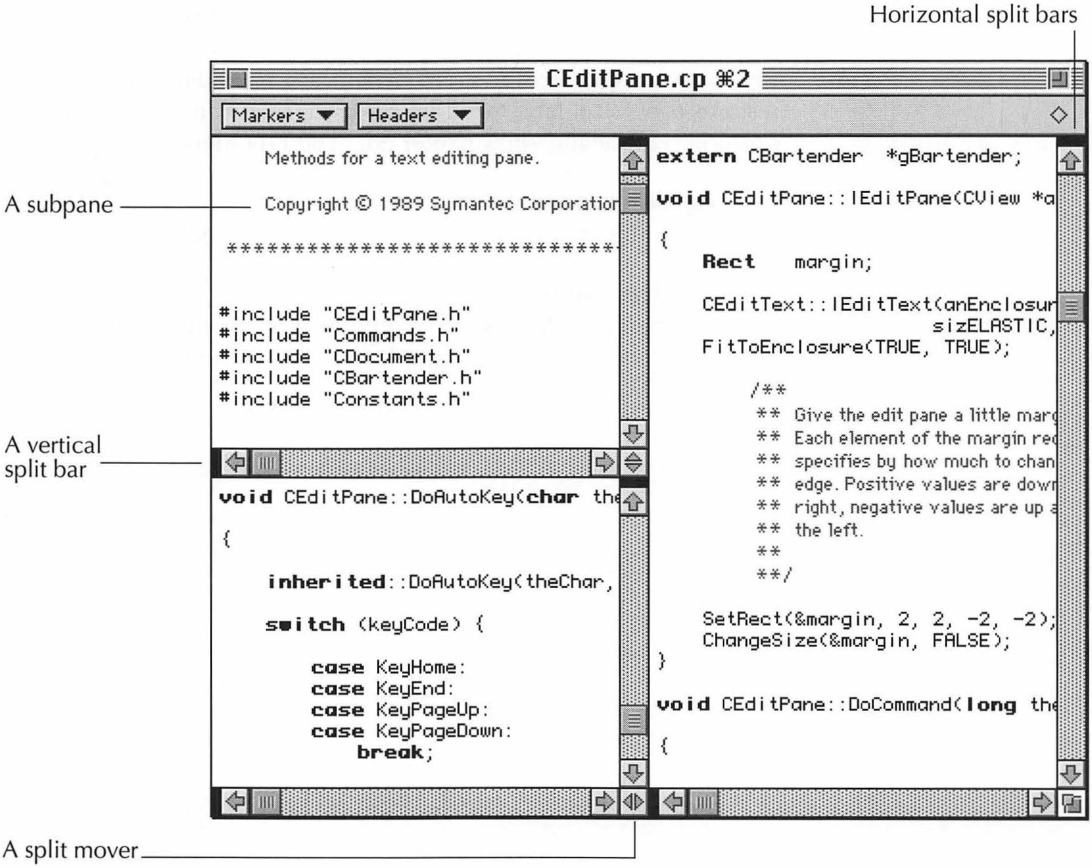
Saving window features
If the preferences Honor Saved View Settings and Honor Saved Font Settings are set in the Editor Options General Settings page of the **Project Manager Preferences** dialog box, the Editor saves and restores font, size, tabs, window position, selection, and scroll position. For details, see Chapter 20, "The Edit Menu."

Closing all windows
Option-clicking in a close box closes all open windows.

Editor independence
You can open an Editor window without first opening a project, and you can edit files while a project is building in the background.

Split windows

An Editor window can be split to view different parts of a source file, as shown in Figure 19-4.



Horizontal split bars

A subpane

A vertical split bar

A split mover

```
    Methods for a text editing pane.

    Copyright © 1989 Symantec Corporatio

 ****************************


#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"
```

```
void CEditPane::DoAutoKey(char th

{

    inherited::DoAutoKey(theChar,

    switch (keyCode) {

        case KeyHome:
        case KeyEnd:
        case KeyPageUp:
        case KeyPageDown:
            break;
```

```
extern CBartender   *gBartender;

void CEditPane::IEditPane(CView *a

{
    Rect    margin;

    CEditText::IEditText(anEnclosur
                         sizELASTIC,
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little marg
         ** Each element of the margin rec
         ** specifies by how much to chan
         ** edge. Positive values are dowr
         ** right, negative values are up a
         ** the left.
         **
         **/

    SetRect(&margin, 2, 2, -2, -2);
    ChangeSize(&margin, FALSE);
}

void CEditPane::DoCommand(long th

{
```

Figure 19-4 A split window

The subpanes can be scrolled independently to bring different portions of the code into view.

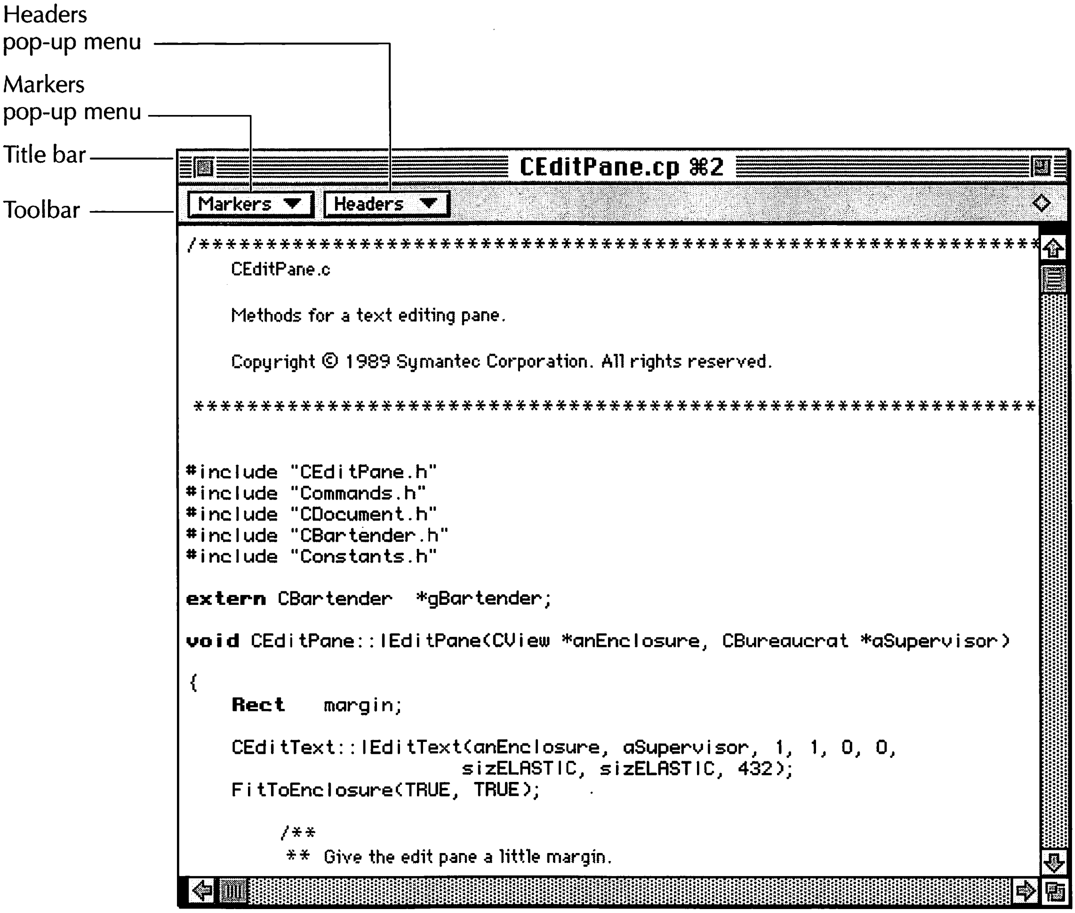
You split a window using a split bar. This is the black rectangle located between the arrow on the horizontal or vertical scroll bar and the edge of the pane. Clicking and dragging a split bar splits a window. Double-clicking a split bar splits a window into two equal subpanes.

You move a split point using a so-called split mover. This is the double triangle located next to the split bar on horizontal and vertical scroll bars. Clicking and dragging on the split mover for a split point moves the split point. Dragging the split point all the way to the edge cancels the split, as does double-clicking the split mover.

Event suites
You can create and record scripts automatically using AppleScript. To learn how to use this feature, see Chapter 8, "Advanced Topics."

Window toolbar

The toolbar at the top of the Editor window contains the **Markers** and **Headers** pop-up menus (Figure 19-5). You can hide the toolbar using **Show Toolbar** in the **Windows** menu.

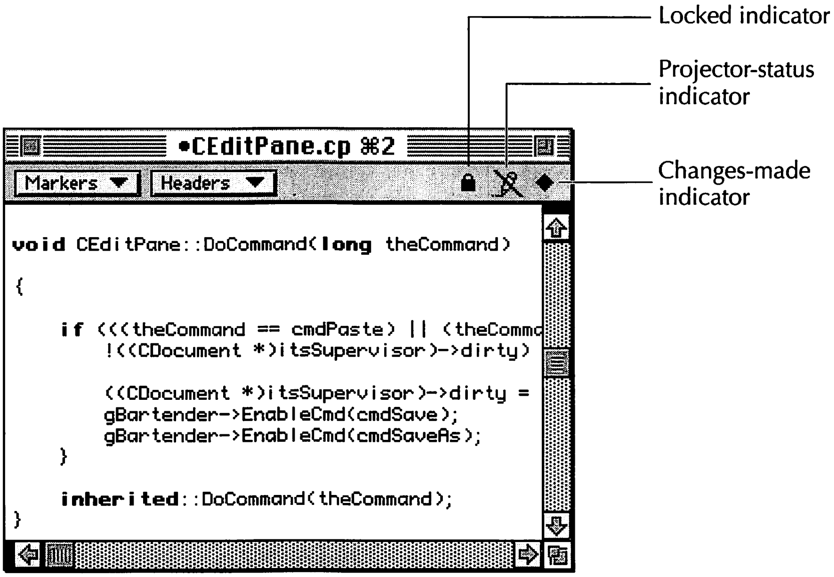


```
/*******************************************************************
    CEditPane.c

    Methods for a text editing pane.

    Copyright © 1989 Symantec Corporation. All rights reserved.

 ******************************************************************


#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)

{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                         sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little margin.
```

Figure 19-5 Editor window's toolbar

The toolbar also contains a diamond displayed to the right. This diamond is filled if the window's contents have changed since the file was saved. Note that you can choose to hide the changes-made bullet (see Figure 19-1), but the diamond indicator is always displayed.

If the file is locked, a lock icon is displayed in the toolbar. If the file is under source control with Projector/SourceServer, the Projector status is shown in the toolbar. These items are shown in Figure 19-6.



```
void CEditPane::DoCommand(long theCommand)
{

    if (((theCommand == cmdPaste) || (theComm
        !((CDocument *)itsSupervisor)->dirty)

        ((CDocument *)itsSupervisor)->dirty =
        gBartender->EnableCmd(cmdSave);
        gBartender->EnableCmd(cmdSaveAs);
    }

    inherited::DoCommand(theCommand);
}
```

Figure 19-6 Changes-made, locked, and projector-status indicators in the window toolbar

See the electronic supplementary information for details on Projector/Source Server control. Also see Chapter 8, "Advanced Topics."

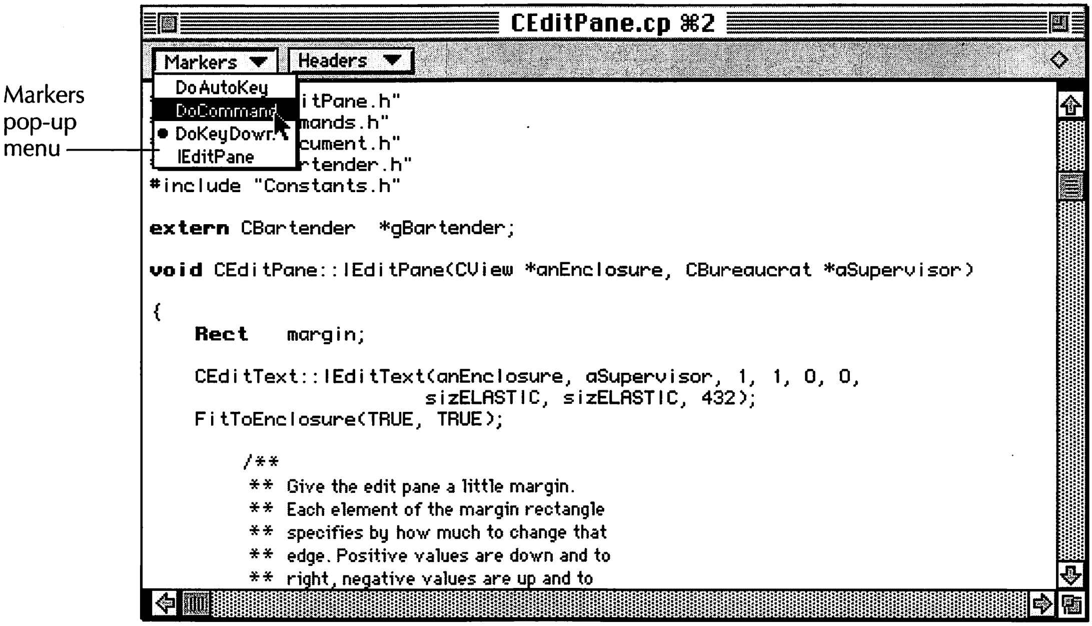
Markers and Headers pop-up menus

The **Markers** pop-up menu displays user-defined markers and symbols parsed from C/C++ source code. The **Headers** pop-up menu lists the headers for your source file.

Markers pop-up menu. The **Markers** pop-up menu lists instances of the following kinds of items:

- Functions
- Class/`struct` definitions
- `#pragma` marks
- `enum` declarations
- `typedef` declarations
- User markers

To see a list of these items in a file, click the **Markers** pop-up menu, as shown in Figure 19-7.

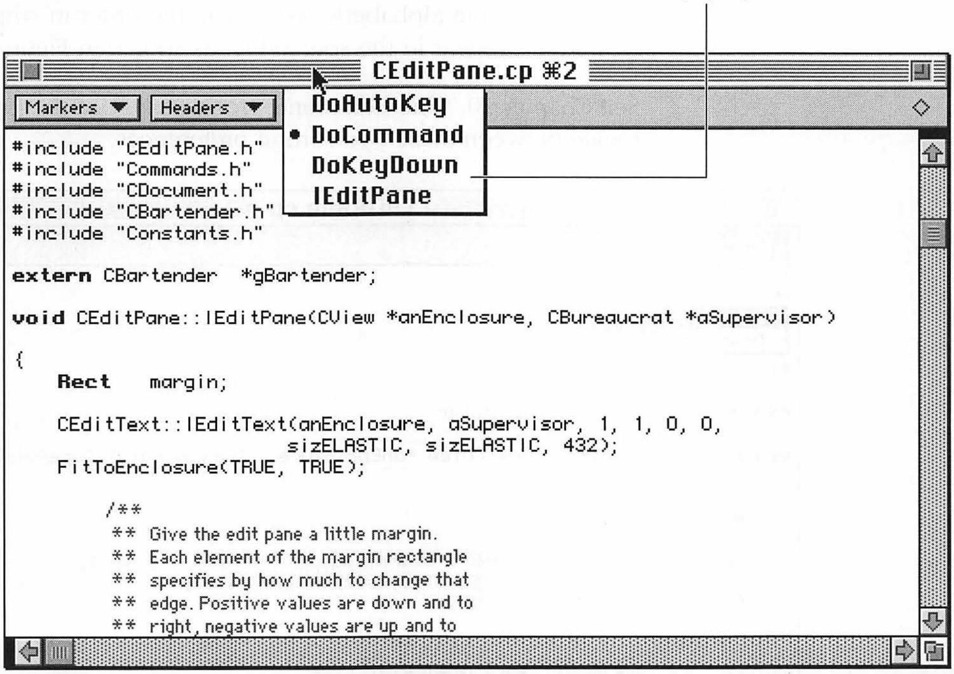


```
             itPane.h"
             mands.h"
             cument.h"
             rtender.h"
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)

{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                         sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);

        /**
         **  Give the edit pane a little margin.
         **  Each element of the margin rectangle
         **  specifies by how much to change that
         **  edge. Positive values are down and to
         **  right, negative values are up and to
```

Figure 19-7 Markers pop-up menu accessed from the toolbar

You can also access that list by holding down the Command key and clicking the title bar (not the title itself), as shown in Figure 19-8.

Markers pop-up menu



```
#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)

{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                         sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little margin.
         ** Each element of the margin rectangle
         ** specifies by how much to change that
         ** edge. Positive values are down and to
         ** right, negative values are up and to
```

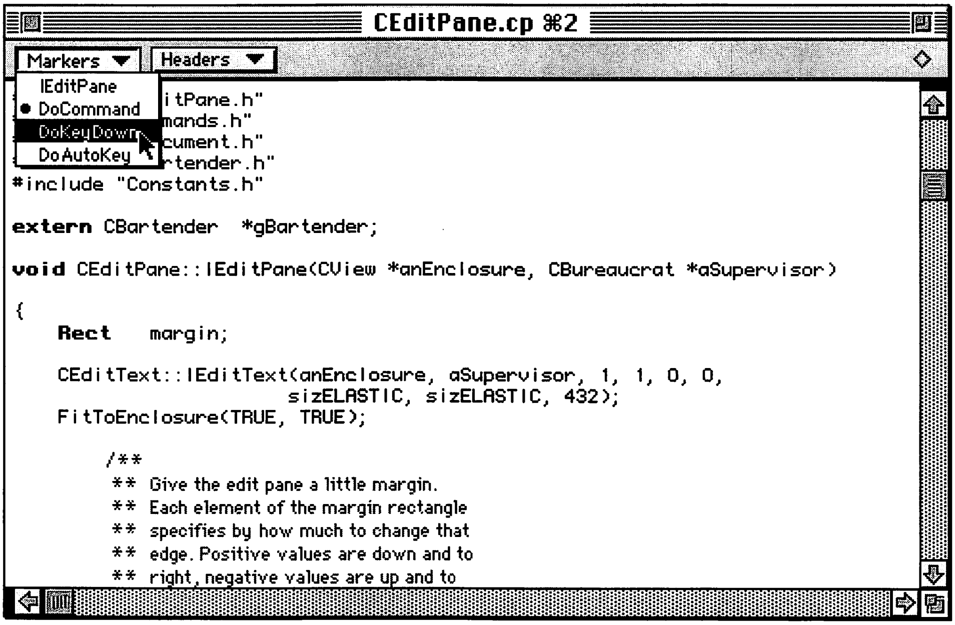
Figure 19-8 Markers pop-up menu accessed from the title bar

Accessing the list in this way is useful if you prefer to work with the toolbar hidden.

The item preceding (and closest to) the current insertion point is marked on the pop-up menu with a bullet (•). Selecting an item in the **Markers** pop-up menu scrolls to the marker or to the declaration of the item.

The marker or item is also selected if the Change Selection option on the Editor Options Marker Pop-up page of the **Project Manager Preferences** dialog box is set. On this same page of the dialog box, you can indicate whether you want items to appear in the **Markers** pop-up menu in alphabetic order or in the order in which their definitions appear in the source file, as shown in Figure 19-9.

See Chapter 20, "The Edit Menu," for details. Use the Option key to toggle between these two sorting preferences.



```
itPane.h"
ands.h"
ument.h"
tender.h"
#include "Constants.h"

extern CBartender  *gBartender;

void CEditPane::IEditPane(CView *anEnclosure, CBureaucrat *aSupervisor)
{
    Rect    margin;

    CEditText::IEditText(anEnclosure, aSupervisor, 1, 1, 0, 0,
                         sizELASTIC, sizELASTIC, 432);
    FitToEnclosure(TRUE, TRUE);

        /**
         ** Give the edit pane a little margin.
         ** Each element of the margin rectangle
         ** specifies by how much to change that
         ** edge. Positive values are down and to
         ** right, negative values are up and to
```

Figure 19-9 Markers pop-up menu with sorted list

The items displayed in the **Markers** pop-up menu can be drawn in different styles or tagged with different characters. For more information, see the description of the Editor Options Marker Pop-up page of the **Project Manager Preferences** dialog box in Chapter 20, "The Edit Menu."

Note that the **Go To Marker** dialog box, which is accessible from the **Search** menu, provides the same list of markers as the **Markers** pop-up menu. By displaying this dialog box, you can navigate to a marker without using the mouse. For details, see Chapter 21, "The Search Menu."

Headers pop-up menu. The **Headers** pop-up menu works the same way as the **Markers** pop-up menu. You can access the **Headers** pop-up menu either by clicking Headers on the toolbar or by holding down the Option key and clicking the title bar (not the title itself). Holding down the Shift key and clicking on Headers sorts the headers by the order in which they were included in the source file.

Holding down the Control key and clicking on Headers displays all headers included by the source file, including those used to build the precompiled header (if any, and if the header files can be found). Holding down the Control and Shift keys and clicking on Headers displays the expanded list in the order in which the headers were included.

Note

> Headers do not appear until a file is compiled. If a file is not compiled, the words "Not compiled yet" are displayed. If the file is compiled but no headers exist, the word "None" is displayed.

Text features

The four text features of the Editor window are described in this section.

Text-entry features

In the Editor window, you have access to all of the standard text-entry features of other Macintosh editors, as well as to some special features. Double-clicking and dragging anywhere in an Editor window selects the first and last whole words that you drag through, as well as all text in between. Triple-clicking and dragging selects entire lines.

Table 19-1 describes deletion options in the Editor window.

Table 19-1 Deletion options in the Editor window

Press this	To delete
Delete	The character to the left of the insertion point
Option-Delete	To the beginning of a word
Command-Delete	To the beginning of a line

Table 19-2 describes options for moving the insertion point in text.

Table 19-2 Moving the insertion point in the Editor window

Press this	To move the insertion point
Up Arrow	Up one line
Down Arrow	Down one line
Left Arrow	Left one character
Right Arrow	Right one character
Option-Up Arrow	To the top of the page
Option-Down Arrow	To the bottom of the page
Option-Left Arrow	Left one word
Option-Right Arrow	Right one word
Command-Up Arrow	To the beginning of the file
Command-Down Arrow	To the end of the file
Command-Left Arrow	To the beginning of the line
Command-Right Arrow	To the end of the line

If you have an Apple Extended keyboard, the keys above the arrow keys can be used as described Table 19-3.

Table 19-3 Using the editing keys

Press this	To do this
Forward Delete	Delete the character to the right of the insertion point
Home	Scroll to the beginning of the file
End	Scroll to the end of the file
Page Up	Scroll to the previous screen
Page Down	Scroll to the next screen
Option-Forward Delete	Delete to the end of a word
Option-Home	Scroll all the way to the left
Option-End	Scroll all the way to the right
Option-Page Up	Scroll to the left
Option-Page Down	Scroll to the right
Command-Forward Delete	Delete to the end of a line

Note

The Home, End, Page Up, and Page Down keys just scroll the file. They do not move the insertion point.

Syntax highlighting

The Editor lets you highlight five categories of language elements in different colors and styles. The default colors and styles are as follows:

- Comments: gray, plain
- Keywords: black, bold
- Preprocessor directives: blue, plain
- String literals: red, plain
- Character constants: red, plain

These defaults can be overridden by setting options in the Editor Options Syntax Formatting page of the **Project Manager Preferences** dialog box. See Chapter 20, "The Edit Menu."

The Editor supports syntax highlighting in the following languages:

- C
- C++
- AppleScript
- MPW Shell Script
- Pascal
- Symantec Rez

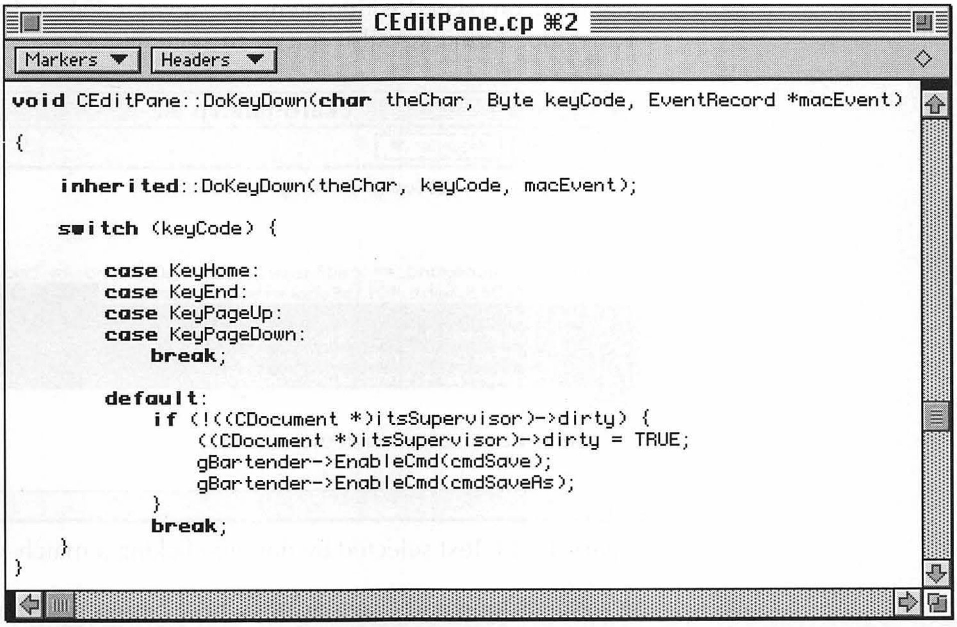
The Editor determines the language to use by looking at the filename extension. You can set the mapping between filename extensions and languages in the Extensions Mapping page of the **Project Options** dialog box. See Chapter 18, "The Project Menu," for details. The default mappings are summarized in Table 19-4.

Table 19-4 Default mapped languages

Filename Extension	Mapped Language
`.c`	C
`.cp, .h, .cpp, .tem`	C++
`.se`	AppleScript
`.ss, .ts`	MPW Shell Script
`.p`	Pascal
`.r`	Symantec Rez

Auto-indenting

The Editor auto-indents a line to the tab setting of the preceding line. In addition, the Editor can add an extra level of indenting after you type an opening code-block delimiter, then press Return. For example, this would be a left brace { in C or C++ or `begin` in Pascal. This is shown for C++ in Figure 19-10.



```
void CEditPane::DoKeyDown(char theChar, Byte keyCode, EventRecord *macEvent)
{
    inherited::DoKeyDown(theChar, keyCode, macEvent);

    switch (keyCode) {

        case KeyHome:
        case KeyEnd:
        case KeyPageUp:
        case KeyPageDown:
            break;

        default:
            if (!((CDocument *)itsSupervisor)->dirty) {
                ((CDocument *)itsSupervisor)->dirty = TRUE;
                gBartender->EnableCmd(cmdSave);
                gBartender->EnableCmd(cmdSaveAs);
            }
            break;
    }
}
```

Figure 19-10 C/C++ auto-indenting

Block auto-indenting is available on a language-specific basis for all languages listed in the section “Syntax highlighting” earlier in this chapter. Plain auto-indenting works on all text files.

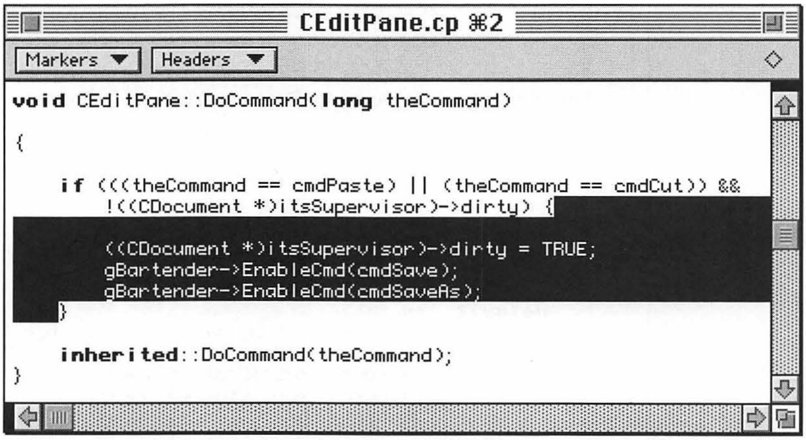
The Editor can also automatically outdent a level after typing a return after the closing code-block delimiter, such as a right brace } in C or C++ or `end` in Pascal (that is, it outdents the line containing the closing delimiter).

If this feature is not appropriate for your coding style, you can turn it off in the Editor Options General Settings page of the **Project Manager Preferences** dialog box. For details, see Chapter 20, “The Edit Menu.” For a specific line, you can override indenting or outdenting modes by using Option-Return.

Delimiter matching

Matching is provided for matched delimiters, including parentheses ( ), brackets [ ], or braces { } and for string, character constant, or comment delimiters, including single quote ', double quote ", or slash /.

Double-clicking either of a pair of matched delimiters selects the text that is encompassed within them, as shown in Figure 19-11. Holding down the Option key also selects the delimiters.

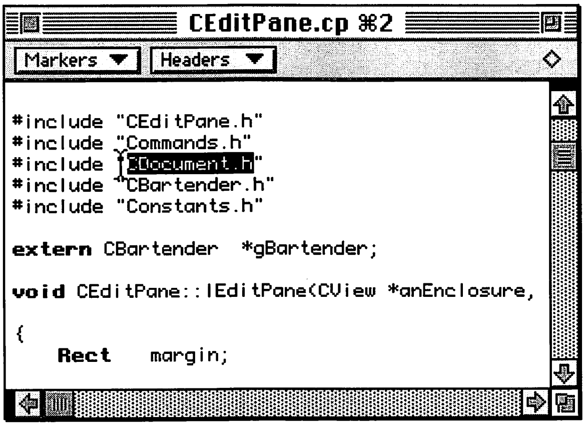


```
void CEditPane::DoCommand(long theCommand)

{

    if (((theCommand == cmdPaste) || (theCommand == cmdCut)) &&
        !((CDocument *)itsSupervisor)->dirty) {

        ((CDocument *)itsSupervisor)->dirty = TRUE;
        gBartender->EnableCmd(cmdSave);
        gBartender->EnableCmd(cmdSaveAs);
    }

    inherited::DoCommand(theCommand);
}
```

Figure 19-11 Text selected by double-clicking a matched delimiter

Double-clicking and dragging anywhere between two matching delimiters selects all the text between those delimiters.

Double-clicking a string, character constant, or comment delimiter selects forward to the next instance of such a delimiter, as shown in Figure 19-12.



```
#include "CEditPane.h"
#include "Commands.h"
#include "CDocument.h"
#include "CBartender.h"
#include "Constants.h"

extern CBartender   *gBartender;

void CEditPane::IEditPane(CView *anEnclosure,

{
    Rect    margin;
```

Figure 19-12 Text selected forward by double-clicking a string delimiter

By default, the selection includes only the text. Holding down the Option key while double-clicking the delimiter also selects the delimiter.

The Edit Menu

20

This reference chapter provides detailed descriptions of the commands on the Symantec Project Manager **Edit** menu. The four pages of the **Project Manager Preferences** dialog box are also covered. See Chapter 19, "The Editor Window," and Chapter 21, "The Search Menu," for further features available for editing text files.

Commands in the Edit Menu

The **Edit** menu contains the standard Macintosh editing commands (**Cut**, **Copy**, **Paste**), as well as other commands for manipulating text in source files. The **Edit** menu also contains the **Preferences** command, which lets you set preferences for the Symantec Project Manager and the text editor. The commands in the **Edit** menu are shown in Figure 20-1.

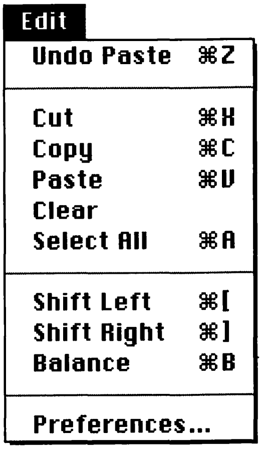


Figure 20-1 Edit menu

The commands in the **Edit** menu are used to perform the following functions:

- Edit and manipulate text in source files
- Set preferences

This chapter discusses the **Edit** menu commands by function in the order listed in Figure 20-1.

Editing and manipulating text

Most of the commands in the **Edit** menu are used to work with or select text in a source pane.

Undo ⌘Z

Reverses the most recent edit operation. The name of this command changes to reflect the operation you are undoing. After a paste, for example, this command changes to **Undo Paste**. Once you have undone something, the name of this command changes to **Redo**.

If you have not performed a command that can be undone, this command is disabled.

Cut ⌘X

Removes selected text and places it on the Clipboard. The command replaces the current contents, if any, of the Clipboard.

Copy ⌘C

Copies selected text and places it on the Clipboard. This command replaces the current contents, if any, of the Clipboard.

Paste ⌘V

Copies the contents of the Clipboard into the file being edited at the insertion point. If text is currently selected, it is replaced by the Clipboard contents.

Clear

Clears the selected text, but does not place it on the Clipboard. Pressing the Clear key has the same effect as choosing the **Clear** command. You can also use the Delete key to clear selected text.

Select All ⌘A

Selects all the text in the current Editor window, if the Editor window is frontmost. The command selects all project entries and groups, if the Project window is frontmost.

Shift Left ⌘[

Shifts the selected range of lines to the left. If a line begins with a tab, the tab is deleted. If the line begins with spaces, enough spaces to equal a tab are deleted. The command has no effect on a line that does not begin with spaces or a tab.

Shift Right ⌘]

Shifts the selected range of lines to the right. The command inserts a tab at the start of each line.

Balance ⌘B

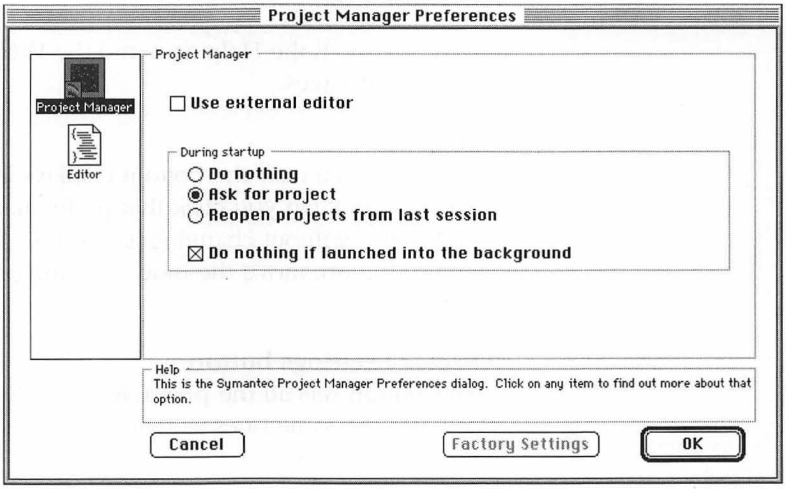
Extends the currently selected text in both directions until opening and closing parentheses `()`, square brackets `[]`, or braces `{}` are reached. Successive invocations select larger sequences of text.

Note

> Balance is a textual operation. The search includes comments and strings. For example, if you have a lone brace, square bracket, or parenthesis in a comment, **Balance** tries to find a match for it.

Preferences

Opens the **Project Manager Preferences** dialog box, in which you set options for how the Symantec Project Manager and the text editor behave (Figure 20-2).



Project Manager Preferences

Project Manager

☐ Use external editor

During startup

○ Do nothing
● Ask for project
○ Reopen projects from last session

☒ Do nothing if launched into the background

Help

This is the Symantec Project Manager Preferences dialog. Click on any item to find out more about that option.

Figure 20-2 Project Manager Preferences dialog box

There are four pages in the **Project Manager Preferences** dialog box, one Project Manager page and three Editor Options pages. The four pages are:

- Project Manager, for specifying those projects that be opened when you launch the Symantec Project Manager
- Editor Options General Settings, for specifying general editing characteristics, such as fonts and indenting
- Editor Options Syntax Formatting, for specifying the colors, fonts, and styles to be used in syntax highlighting
- Editor Options Marker Pop-up, for specifying the items to be shown in the Editor window's **Markers** pop-up menu

By default, the **Preferences** dialog box opens to the Project Manager page.

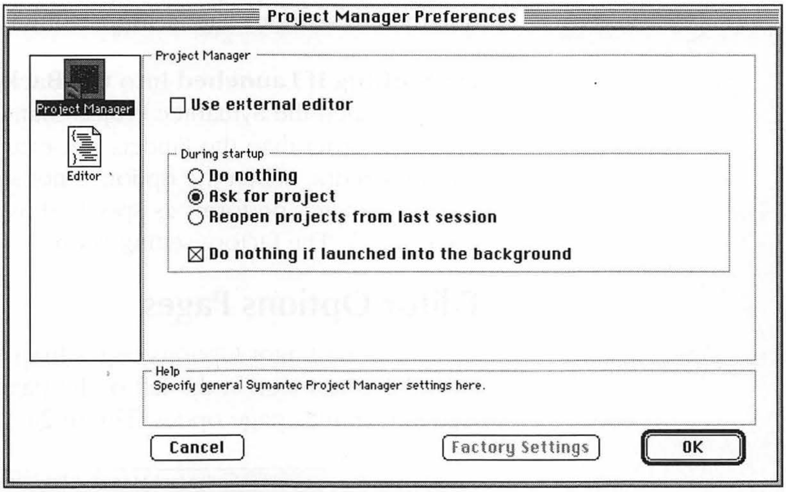
Note that both the Help area and the Factory Settings button appear on all four pages.

Help area
The Help area near the bottom displays information about a preference when you click that preference. To receive help on a preference without changing its setting, click it without releasing the mouse button, move the mouse off the preference, then release the button.

Factory Settings button
This button sets all the preferences on the page to their factory settings, the same ones as when you first installed the Symantec Project Manager.

Project Manager Page

Use this page (Figure 20-3) to enable an external editor and to set launch preferences. This section describes the options on this page.



Project Manager

□ Use external editor

During startup

○ Do nothing
● Ask for project
○ Reopen projects from last session

☒ Do nothing if launched into the background

Help
Specify general Symantec Project Manager settings here.

Figure 20-3 Project Manager page

Use External Editor

This preference lets you use an external (third-party) editor in place of the Symantec editor. An alias named "Editor," which refers to the external editor application, must exist in the `(Tools)` folder. The factory setting is off.

Note

> Third-party editors for the THINK Project Manager need to be upgraded to take full advantage of the Symantec Project Manager. Check with the company that wrote your editor for more information.

During Startup preferences

You can choose the following four launch preferences for start-up: Do Nothing, Ask for Project, Reopen Projects from Last Session, or Do Nothing If Launched into the Background.

Ask for Project. Prompts you to open a project when you start the Symantec Project Manager. The factory setting is off.

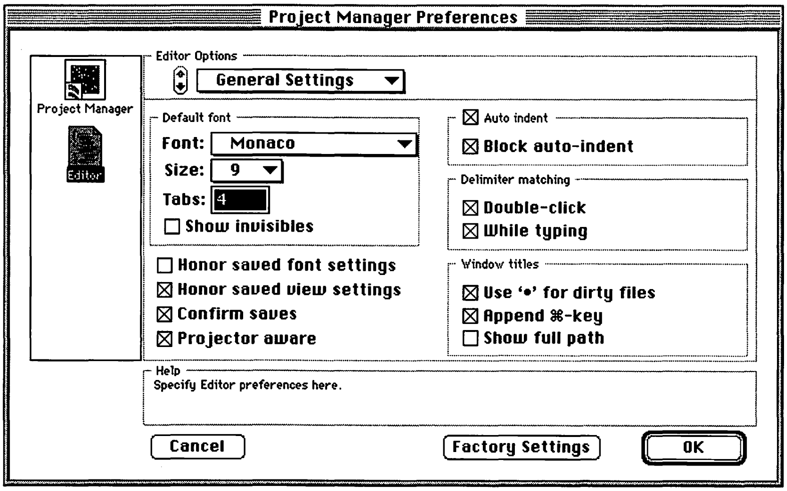
Do Nothing. Does not prompt you to open a project when you start the Symantec Project Manager. The factory setting is on.

Reopen Projects from Last Session. Opens all projects, when you start the Symantec Project Manager, that were open at the time you quit the last session. The factory setting is off.

Do Nothing If Launched into the Background. Does not open projects when the Symantec Project Manager is launched from programs other than the Finder—for example, if it is launched from an AppleScript. When the option is not selected, the Symantec Project Manager performs as specified by the During Startup preferences. The factory setting is on.

Editor Options Pages

To access the Editor Options pages from the Project Manager page, click the editor icon at the left of the page. The Editor Options General Settings page opens (Figure 20-4).



Editor Options: General Settings

Default font
Font: Monaco
Size: 9
Tabs: 4
☐ Show invisibles

☐ Honor saved font settings
☒ Honor saved view settings
☒ Confirm saves
☒ Projector aware

☒ Auto indent
☒ Block auto-indent

Delimiter matching
☒ Double-click
☒ While typing

Window titles
☒ Use '•' for dirty files
☒ Append ⌘-key
☐ Show full path

Help
Specify Editor preferences here.

Figure 20-4 Editor Options General Settings page

To access the other two Editor Options pages, do one of the following:

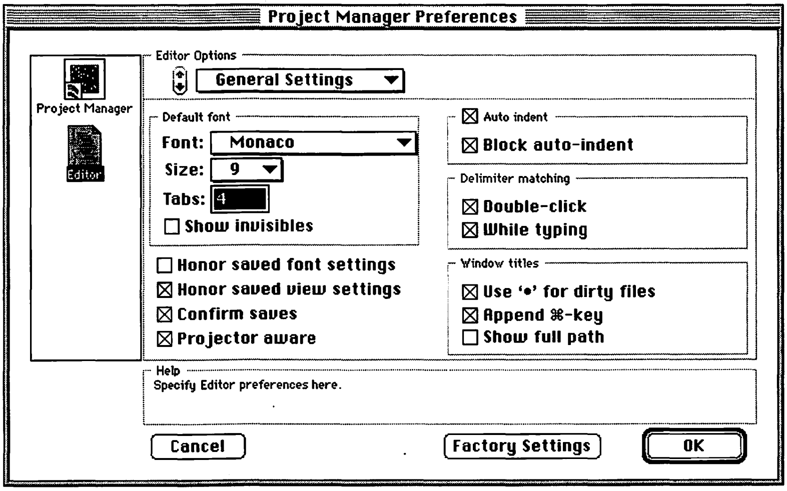
- Click the **Editor Options** pop-up menu and select a page from the list.

- Scroll through the entries on the pop-up menu with the arrow buttons to the left of the menu.
- Press Command-Up Arrow or Command-Down Arrow to open a page.

To return to the Project Manager page, click the Project Manager icon at the left of the page.

Editor Options General Settings page

Use this page to set general settings such as text font, tab settings, and whether you are prompted to save your work.



Editor Options: General Settings

Default font
Font: Monaco
Size: 9
Tabs: 4
□ Show invisibles

□ Honor saved font settings
☒ Honor saved view settings
☒ Confirm saves
☒ Projector aware

☒ Auto indent
☒ Block auto-indent

Delimiter matching
☒ Double-click
☒ While typing

Window titles
☒ Use '•' for dirty files
☒ Append ⌘-key
□ Show full path

Help
Specify Editor preferences here.

Figure 20-5 Editor Options General Settings page

Default Font

Use the Default Font area to specify the default font characteristics of text.

Font. Opens a pop-up menu for selecting the default font used for new Editor windows. The factory setting is Monaco.

Size. Opens a pop-up menu for selecting the default font size used for new Editor windows. The factory setting is 9.

Tabs. Specifies the default number of spaces per tab used for new Editor windows. The factory setting is 4.

Show Invisibles. Displays nonprinting characters with glyphs in new Editor windows. The factory setting is off.

Honor Saved Font Settings
When this preference is set on, the saved font settings in text files are used instead of the global font settings. The factory setting is on.

Honor Saved View Settings
When this preference is set on, windows are reopened to their last screen positions. The factory setting is on.

Confirm Saves
When this preference is set on, the Symantec Project Manager prompts you to save files when you close them or when you try to run a project. If this preference is off, the Symantec Project Manager saves files without asking. The factory setting is on.

Projector Aware
When this preference is set on, the Symantec Project Manager honors the 'ckid' resource in every file it opens. This is the resource used by SourceServer and MPW's SourceServer. The factory setting is on.

To learn about MPW, see the electronic supplemental information.

Auto indent. Tabs to the indent position of the current line when you type a carriage return. The factory setting is on.

Block auto-indent. Indents an extra tab when you press Return after a left code block delimiter (for example, a left brace {). This option outdents when you press Return after a right code block delimiter. The factory setting is on.

Delimiter Matching
Use the Delimiter Matching area to specify matching features.

Double-Click. Highlights the enclosed text when a matched delimiter is double-clicked. The factory setting is on.

While Typing. Flashes the opening delimiter that matches the closing delimiter that was just typed. The factory setting is on.

Window Titles
Use the Window Titles area to specify additional information to be displayed in Editor window titles.

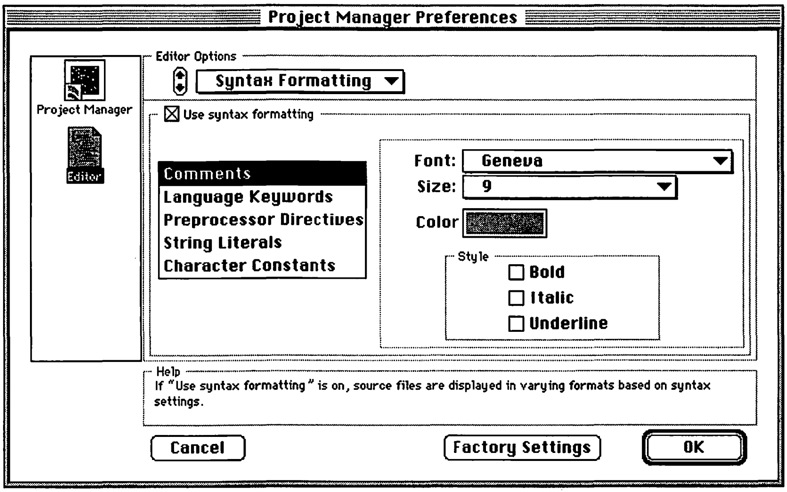
Use '•' for Dirty Files. A'•' is prepended to the window titles of files that have been changed. The factory setting is on.

Append Command-Key. When this preference is set, the numbered Command-key equivalents (if any) for windows are appended to the window titles. The factory setting is on.

Show Full Path. Displays each source file's full pathname in its window's title. The factory setting is off.

Editor Options Syntax Formatting page

Use this page (Figure 20-6) to set font style and color preferences for syntax highlighting.



Syntax Formatting

☒ Use syntax formatting

Font: Geneva

Size: 9

Color

□ Bold

□ Italic

□ Underline

If "Use syntax formatting" is on, source files are displayed in varying formats based on syntax settings.

Figure 20-6 Editor Options Syntax Formatting page

Use Syntax Formatting

This group of preferences lets you display source files in various highlighting formats based on syntax settings that you specify on this page. You can choose a font, size, color, and style for highlighting each of the following syntax categories:

- Comments
- Language keywords
- Preprocessor directives
- String literals
- Character constants

The factory setting is on.

To specify a highlighting format for a syntax category:

1. Select a syntax category in the menu from the Use Syntax Formatting area.

2. Select a font, size, color, and style.

Font
Use this pop-up menu to select the font to be used when displaying the selected syntax category.

Size
Use this pop-up menu to select the default font size to be used when displaying the selected syntax category.

Color
Use this field to select the color to be used when displaying characters of the selected syntax category. You select colors using the standard Macintosh color picker.

Style
Use the Style area to specify the styles to be used when displaying characters of the selected syntax category. You may choose any combination of bold, italic, and underline.

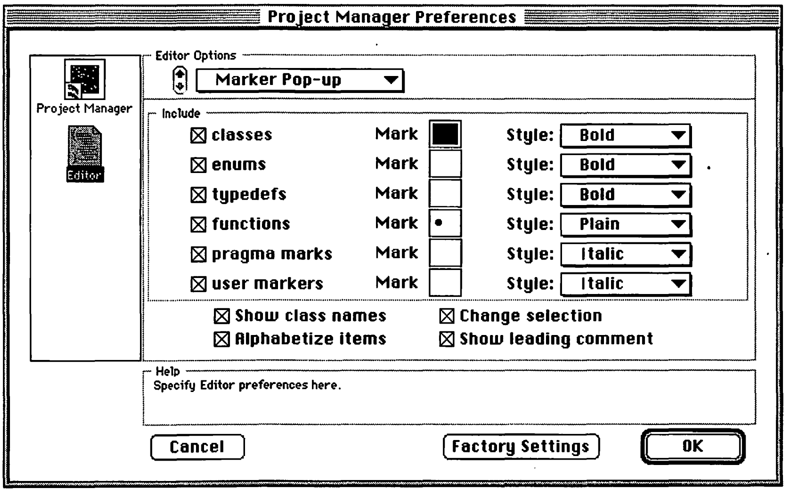
Bold. Displays the characters of the selected syntax category in boldface.

Italic. Displays the characters of the selected syntax category in italic.

Underline. Underlines the characters of the selected syntax category.

Editor Options Marker Pop-up page

Use this page (Figure 20-7) to set preferences for the items in the **Markers** pop-up menu of a Source window.



Project Manager Preferences

Editor Options: Marker Pop-up

Include

☒ classes	Mark	■	Style: Bold
☒ enums	Mark	□	Style: Bold
☒ typedefs	Mark	□	Style: Bold
☒ functions	Mark	•	Style: Plain
☒ pragma marks	Mark	□	Style: Italic
☒ user markers	Mark	□	Style: Italic

☒ Show class names ☒ Change selection
☒ Alphabetize items ☒ Show leading comment

Help
Specify Editor preferences here.

Figure 20-7 Editor Options Marker Pop-up page

Include area

Use the Include area to specify the types of items to be included in a Source window's **Markers** pop-up menu.

Classes. Includes set, class, `struct`, and union declarations in the **Markers** pop-up menu. The factory setting is on.

Enums. Includes `enum` definitions in the **Markers** pop-up menu. The factory setting is on.

Typedefs. Includes `typedef` statements in the **Markers** pop-up menu. The factory setting is on.

Functions. Includes function definitions in the **Markers** pop-up menu. The factory setting is on.

Pragma Marks. Includes `#pragma` mark statements in the **Markers** pop-up menu. The factory setting is on.

User Markers. Includes user-defined markers in the **Markers** pop-up menu. The factory setting is on.

Mark areas. Specify the mark characters, if any, for class declaration, `enum` definition, `typedef` statement, function definition, `#pragma` statement, and user-defined markers.

To specify a mark character, double-click the respective Mark field to select it, and type the character.

Style pop-up menus. Specify the styles, if any, for class declaration, `enum` definition, `typedef` statement, function definition, `#pragma` statement, and user-defined markers.

The factory settings are bold for class declarations, `enum` definitions, and `typedef` statements; plain for functions; and italic for `#pragma` marks and user markers.

Show Class Names

Setting this preference on includes the class names in the menu items for class declarations, `enum` definitions, `typedef` statements, and member functions. The factory setting is on.

Alphabetize Items

When this preference is set, the entries in the **Markers** pop-up menu are alphabetized. The factory setting is on.

Change Selection

Setting this option on moves the edit selection when a **Markers** pop-up menu item is selected. Otherwise, the selected item is scrolled into view. The factory setting is on.

Show Leading Comment

Setting this option on shows the comment, if any, that precedes a selected pop-up menu item when the item is selected. The factory setting is on.

The Search Menu 21

This reference chapter provides detailed descriptions of the commands on the Symantec Project Manager **Search** menu. In addition, the options on the **Find** dialog box, which opens when you choose **Find** from the menu, are outlined. The final section covers the Grep pattern search capability.

Commands in the Search Menu

You use the commands in the **Search** menu to go to a specific string or marker in a text file and to locate reference information or errors. The commands in the **Search** menu are shown in Figure 21-1.

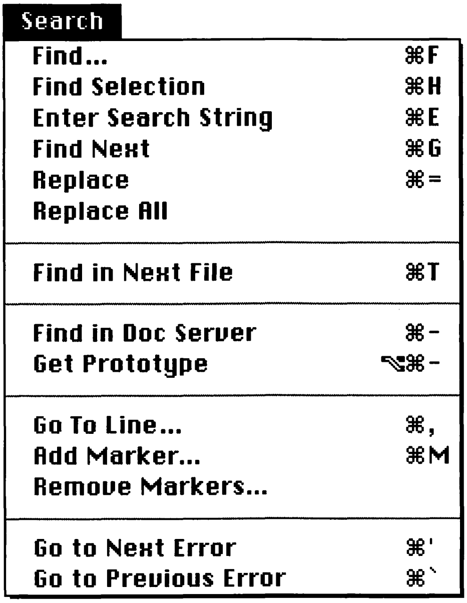


Figure 21-1 Search menu

Note

By holding down some combination of the Shift and Option keys, you can toggle several of the commands in the **Search** menu between two or more commands.

The commands in the **Search** menu can be used to perform the following functions:

- Find and replace text strings, including multi-file searching and sophisticated pattern-matching searching
- Locate information in the Symantec THINK Reference, including finding function prototypes
- Add or remove markers in a file
- Locate compiler errors within a source file

This chapter covers the **Search** menu commands by function in the order listed above.

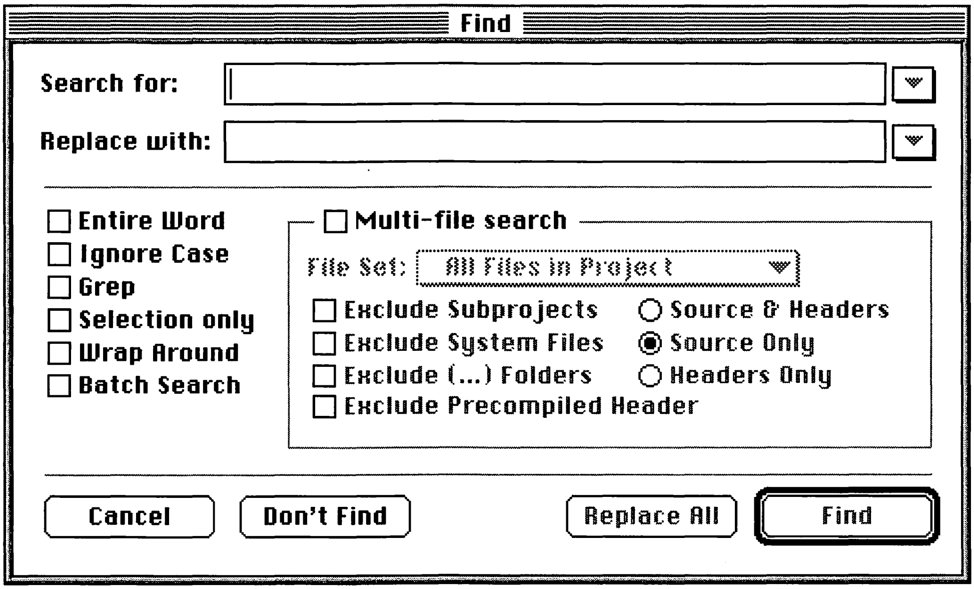
Finding and replacing text strings

You use these commands to find and replace strings in text files.

Find ⌘F

Opens the **Find** dialog box, which you use to specify a search string as well as an optional replacement string (Figure 21-2). You can type the search and replacement strings in the textboxes, or choose from the last five strings you have entered since launching the Project Manager.

If the search string is found, it is highlighted. If not, the editor beeps.



Find

Search for:

Replace with:

□ Entire Word
□ Ignore Case
□ Grep
□ Selection only
□ Wrap Around
□ Batch Search

□ Multi-file search

File Set: All Files in Project

□ Exclude Subprojects
□ Exclude System Files
□ Exclude (...) Folders
□ Exclude Precompiled Header

○ Source & Headers
● Source Only
○ Headers Only

Figure 21-2 Find dialog box

Entire Word
The Editor searches on the basis of whole words only. For example, a search for “stream” does not find the word “streams.”

Ignore Case
The Editor disregards case during a search. This option does not affect the search for diacritical characters.

Grep
The Editor uses a powerful pattern-matching feature based on the Unix `grep` command. See “Searching for a Pattern (Grep)” later in this chapter.

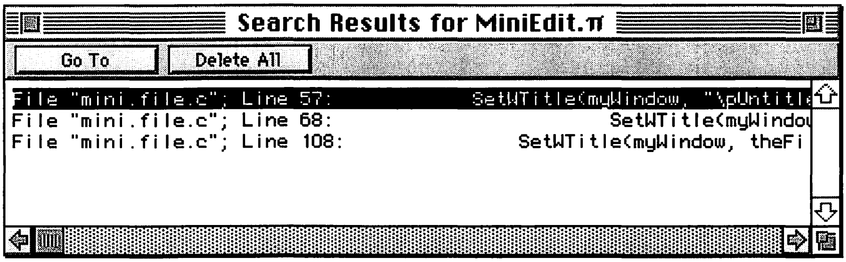
Selection
The search is limited to the currently selected text.

Wrap Around
For a forward search, the Editor reaches the end of the file and then wraps around to the beginning. When this option is set off, a forward search proceeds only from the current position to the end of the file. If the option is set on for a reverse search, the Editor reaches the beginning of the file and wraps to the end.

Batch Search

Instead of stopping at the first occurrence of a search string, the Editor finds all occurrences and displays them in a Search Results window (Figure 21-3). Each project has its own Search Results window.

Each Search Results window displays the results of the last batched **Find** operation for its own project.



```
File "mini.file.c"; Line 57:          SetWTitle(myWindow, "\pUntitl
File "mini.file.c"; Line 68:                    SetWTitle(myWindo
File "mini.file.c"; Line 108:             SetWTitle(myWindow, theFi
```

Figure 21-3 Search Results window

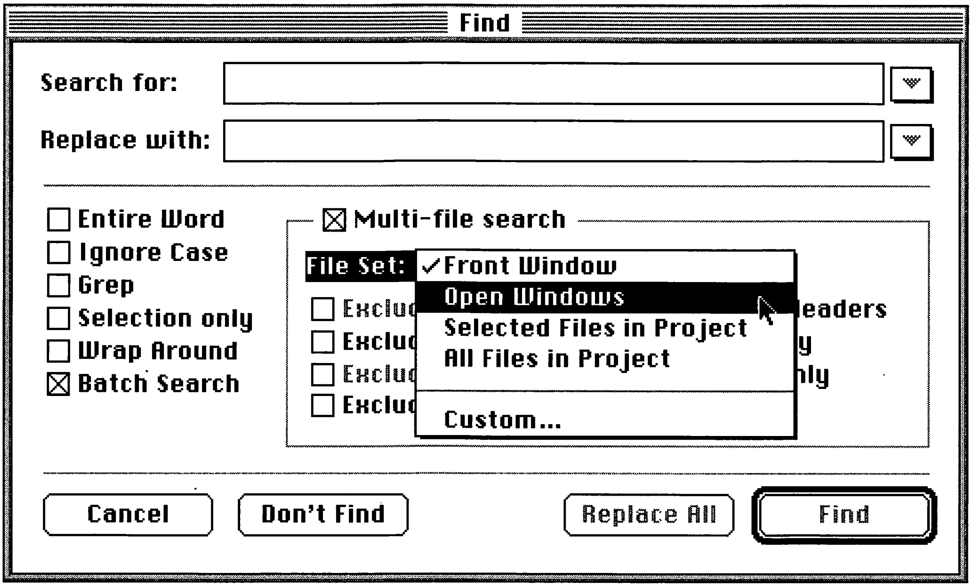
Go To. Clicking Go To in the Search Results window brings you to the selected line in the source file. Double-clicking any entry in the window achieves the same effect.

Delete All. Clicking on Delete All clears the persistent storage of the batch search results from the Search Results window. If the project is opened again, the results from the previous search no longer appear in the window.

Multi-File Search

When this option in the **Find** dialog box is set on, the Editor's search-and-replace functions search multiple files. The collection of files that are searched depends on which multi-file search options are checked.

File Set. Clicking the arrow opens the **File Set** pop-up menu (Figure 21-4).



Search for:

Replace with:

☐ Entire Word
☐ Ignore Case
☐ Grep
☐ Selection only
☐ Wrap Around
☒ Batch Search

☒ Multi-file search

File Set: ✓Front Window

Figure 21-4 File Set pop-up menu

Front Window Limits the search to the frontmost window. If the Project window, Search Results window, or Build Errors window is in front, then this command is disabled.

Open Windows Limits the search to all open text windows. Search Results, Build Errors, Worksheet, and Browsers windows are not included in the search.

Selected Files in Project Limits the search to the currently selected items in the Project window. For groups and subprojects, this means that every file in the group (and its subgroups) is added to the search list.

All Files in Project Includes each file in the project in the file set.

Custom Allows you to specify a folder to be used as the root for building the search list. The folder is searched recursively, and each text file is added to the file set.

Besides making a choice from the **File Set** pop-up menu, you have a number of other multi-file search options on the **File** dialog box, as follows:

Exclude Subprojects. Indicates that files contained by subprojects will not be searched.

Exclude System Files. Indicates that system files in the Symantec Project Manager tree will not be searched.

Exclude (...) Folders. Determines whether, during a custom search, shielded folders are examined. (This option is only enabled for custom file sets.) For further details, see the discussion of shielded folders in Chapter 3, "Starting a Project."

Exclude Precompiled Header. Indicates that the files in the project's precompiled header will not be searched.

Source & Headers. Includes both source files and header files in the search.

Source Only. Searches only source files.

Headers Only. Searches only header files included by source files in the current file set.

There are four buttons in the **Find** dialog box (see Figure 21-2).

Cancel
Click Cancel to stop the operation.

Don't Find
Click Don't Find to accept the new string and option settings, but not initiate a search. This is a useful option if you want to set values for a replace operation without executing the first find.

Replace All
Click Replace All to replace all occurrences of the current selection with the replacement string.

Find
Click Find to go ahead with the search.

Table 21-1 Search dialog box Command key equivalents

Command Key	**Function**
E	Entire word
W	Wrap around
I	Ignore case
G	Grep
B	Batch
M	Toggles multi-file search
F	Find
D	Don't find
R	Replace all
,(comma)	Exclude precompiled header
N	Turns off multi-file search
L	Multi-file, source and headers (Command-A in the THINK Project Manager)
S	Multi-file, Source
H	Multi-file, Headers

Other than **Find**, the **Search** menu contains a number of other find-and-replace commands, as described here (see Figure 21-1).

Find Selection ⌘H

Find Selection Previous

Limits the search to the next occurrence of the selected text. Holding down the Shift key changes **Find Selection** to **Find Selection Previous**. The latter command limits the search to the previous occurrence of the selected text.

Enter Search String ⌘E

Sets the search string to the current selection. Use **Find Next** to begin searching, or **Find** to set search options. Holding down the Option key changes **Enter Search String** to **Enter Replace String**.

Enter Replace String ⌥⌘E

Use this command to set the replacement string for the current operation.

Find Next ⌘G

Find Previous ⇧⌘G

Searches for the next occurrence of the search string. Holding down the Shift key changes **Find Next** to **Find Previous**. Use this command to search backwards for the next occurrence of the selected string.

Replace ⌘=

Replace & Find Next ⌥⌘=

Replace & Find Previous ⇧⌥⌘=

Replaces the current selection with the replacement string. If you do not designate a replacement string, choosing this command deletes the found string. Holding down the Option key changes **Replace** to **Replace & Find Next**. This command replaces the current selection with the replacement string. It then finds the next instance of the search string, but does not replace it.

Use **Replace & Find Next** to step through a series of replacements. After each replacement, you see the next instance of the search string. If you choose to replace the string, use either the **Replace** or **Replace & Find Next** command. To skip to the next instance of the string, use the **Find Next** command. Holding down the Shift key changes **Replace & Find Next** to **Replace & Find Previous**. This command replaces the current selection and finds the nearest instance of the string working in the opposite direction.

Replace All

Replaces every instance of the search string. If the Wrap Around option is set on, this command replaces every instance in the file. If it is set off, this command replaces every instance from the current position to the end of the file. If you have not provided a replacement string, this command deletes the string that has been found.

Find in Next File ⌘T

During a multi-file search, continues the search in the file set. This command also continues an interrupted batch search.

When this command is chosen, a string search is executed. If the search string is found in a given file, an Editor window is opened with the selected string highlighted. At this point, you can edit the string. If you want to search further in the current file, use the **Find**, **Find Next**, **Replace**, or **Replace All** commands, which work within the current file. When you are ready to continue with the multi-file search, use the **Find in Next File** command.

Locating information in the THINK Reference

Use these commands to locate reference information.

Find in Doc Server ⌘- Looks up the current selection in the *THINK Reference* or other external documentation viewer.

Get Prototype ⌥⌘- Calls *THINK Reference* or other external documentation viewer to replace the current selection with its prototype.

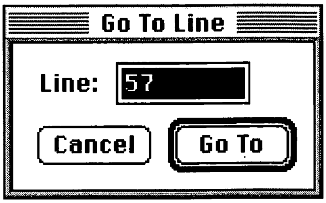
Going to lines or markers

Use these commands to go to specific lines or markers.

Go To Line ⌘,

Go To Marker ⌥⌘,

Opens the **Go to Line** dialog box, which you use to move to a specific line in your file (Figure 21-5). You need to know the line's number. Lines are numbered consecutively from 1.



Line: 57

Figure 21-5 Go To Line dialog box

The default line is the one that contains the insertion point. To change the line, type a different number and click Go To. The Editor moves the insertion point to the beginning of the line with the number you typed. Click Cancel to cancel the operation.

Holding down the Option key changes **Go To Line** to **Go To Marker.** Selecting a marker in the **Go To Marker** dialog box is equivalent to choosing an entry from the **Markers** pop-up menu.

Note

All preferences set for the **Markers** pop-up menu apply also to the **Go To Marker** dialog box.

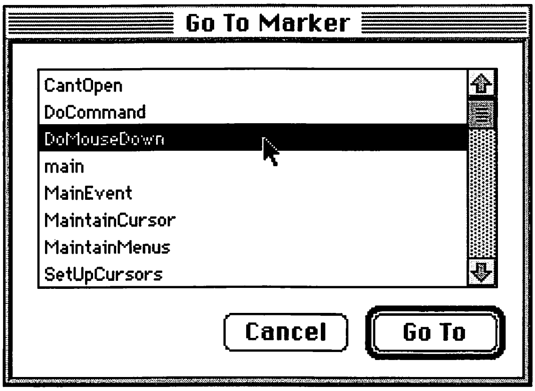


Figure 21-6 Go To Marker dialog box

Add Marker ⌘M

Opens the **Add Marker** dialog box, which you use to place a new marker in a file (Figure 21-7). Place the insertion point at the line you want to mark, or select some text in that line.

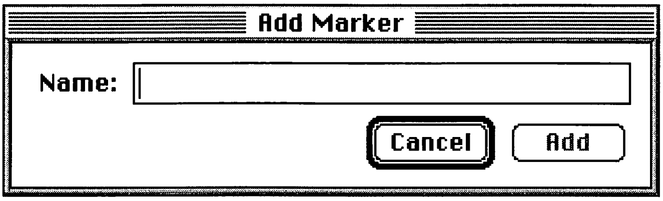


Figure 21-7 Add Marker dialog box

Type the marker name you want, and click Add.

Remove Markers

Opens the **Remove Markers** dialog box, which you use to delete markers from a file (Figure 21-8).

Select the markers to delete and click Remove.

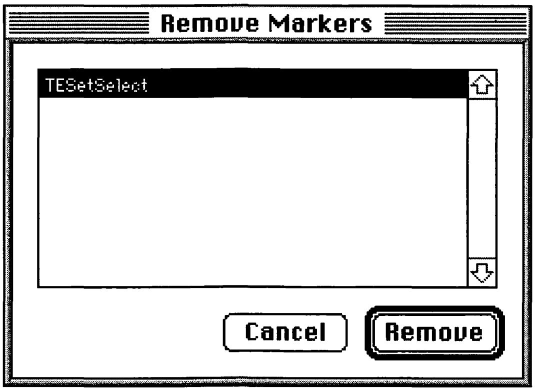


Figure 21-8 Remove Markers dialog box

Locating compiler errors within a source file

Use these commands to locate errors.

Note

> If you are in the Search Results window, the **Search** menu displays the commands **Go to Next Result** and **Go to Previous Result**. If you are in the Build Errors window, the **Search** menu displays the commands **Go to Next Error** and **Go to Previous Error**.

Go to Next Result ⌘' Locates the next search result from the Search Results window.

Go to Previous Result ⌘` Locates the previous search result from the Search Results window.

Go to Next Error ⌘' Locates the next error from the Build Errors window. See Chapter 23, "The Build Menu" for discussion of this window.

Go to Previous Error ⌘` Locates the previous error from the Build Errors window.

Searching for a Pattern (Grep)

In addition to the search and replace functions described in the previous sections, the Editor also provides a powerful pattern search capability called Grep. The Grep search option is based on the Unix Grep utility. The Editor looks for patterns only when the Grep option is set on in the **Find** dialog box.

Patterns

A pattern describes a characteristic (or characteristics) that could apply to many strings. For example, you can build a pattern that means "any word that begins with P." Alternatively, the pattern could mean "any function call with `&event` as an argument."

Note, however, that patterns cannot span lines, so you cannot define a pattern that means "three consecutive lines that begin with a, b, and c."

Simple patterns

The simplest patterns match a single character during a search. Special conventions define the use of the following characters in Grep patterns:

- Use the period to designate any character.

  For example, if you've set Ignore Case on in the **Find** dialog box, any letter matches both its uppercase and lowercase equivalent. So . matches both `a` and `A`, both `b` and `B`, and so on.

- The reverse slash \ followed by any character—except `(`, `)`, <, >, or one of the digits `1` through `9`—defines a pattern that matches that character. For example, `\.` matches `.` and also \\ matches \ (backslash).

- A string of characters *s* surrounded by left `[` and right `]` brackets is a pattern `[s]` that matches any one of the characters in the string `s`. The pattern `[^s]` matches any character not in the string *s*. Using *s* as a string of three characters in the form `a-z` represents all of the characters from `a` to `z` inclusive. All other characters in *s* are taken literally. To include the right bracket `]` in *s*, you must use it as the first character. To include the hyphen `-` in *s*, you must use it at the beginning or end of *s*.

For example, the pattern `[A-Za-z0-9]` matches any alphanumeric character. The pattern `[^!-~]` matches any nonprinting ASCII character. The Ignore Case option has no effect on anything between brackets.

Complex patterns

To match strings, not just individual characters, you need patterns that match consecutive sequences of characters. One way of doing this is to append an asterisk `*` to the end of one of the simple patterns.

- A pattern *x* followed by a `*` is a pattern *x** that matches zero or more consecutive occurrences of characters matched by *x*.

  For example, the pattern `@*` matches a string containing any number of at-signs. If the string does not begin with an at-sign, or if it contains only characters other than at-sign, then the pattern is not found.

You can put patterns together to form more complex patterns.

- A pattern *x* followed by a pattern *y* forms a pattern *xy* that matches any string *ab*, where *a* matches *x* and *b* matches *y*.

  For example, the pattern `P.` matches any two-letter string consisting of `P` and any other character.

- You can concatenate the compound pattern *xy* with another pattern *z*, forming the pattern *xyz*.

To put all this together, consider the pattern `(.*)`. This pattern matches any string enclosed in parentheses. This includes the string `()`, since the subpattern `.*` matches the empty string between the left parenthesis `(` and the right parenthesis `)`.

Does the pattern `(.*)` match the string `(())`? Because the subpattern `.*` matches any number of occurrences of all characters, it might seem that the pattern matches just the `(()` and not the very last `)`. This is not the case. In matching `(())` against the pattern `(.*)`, the inner pair of parentheses matches the subpattern `.*`, so the whole pattern matches `(())`.

Subpatterns

A subpattern is any component of a pattern that is surrounded by reverse slash left parenthesis and reverse slash plus right parenthesis. You can refer to and reuse subpatterns while writing complex patterns. The string `\n` accesses the *n*th subpattern of a given pattern. The combination of grouping subpatterns can become somewhat complicated.

- A pattern surrounded by `\(` and `\)` matches whatever the subpattern matches.

  For example, `\(a[b-y]z\)` matches the same thing as `a[b-y]z`.

- A \ followed by *n*, where *n* is one of the digits `1` through `9`, matches whatever the nth `\( \)` subpattern matched. You can add a `*` to a \*n* pattern to form a pattern \*n** that matches zero or more occurrences of whatever `\n` matched.

  For example, to find two repeated words (like "the the"), you might use a pattern such as this:

  ```
  \([a-z][a-z]*\) \1
  ```

  This pattern matches any string that contains the following: a space, any sequence of letters, a space, and the same sequence of letters. Note that `\1` is not a reapplication of the pattern. Instead it becomes whatever the first `\( \)` pair matched.

Constraining patterns

Finally, you can limit patterns so that they match only if they meet certain conditions in the context outside the string.

- A pattern surrounded by \< and \> matches whatever the pattern matches, provided that the first and last characters of the matched string match `[A-Za-z0-9_]` and that the characters immediately surrounding the matched string don't match `[A-Za-z0-9_]`. In other words, the pattern matches only if the string begins and ends on a word boundary. If you've set Entire Word on in the **Find** dialog box, the entire pattern that you enter is treated as though it were surrounded by \< and \>.

For example, to find occurrences of repeated words, even words not surrounded by spaces, use the pattern `\(\<[a-z][a-z]*\>\)[^a-z]*\1.`

- A pattern *x* preceded by a `^` forms a pattern `^`*x*. If `^`*x* is not preceded by any other pattern, it matches whatever *x* matches as long as the first character *x* matches occurs at the beginning of a line.

- A pattern *x* followed by a `$` forms a pattern *x*`$`. If the pattern *x*`$` is not followed by any other pattern, it matches whatever *x* matches as long as the last character that *x* matches is at the end of a line. If the pattern *x*`$` is followed by another pattern, then the `$` is taken literally.

Because of these last two items, pattern matches must begin or end at line boundaries. They can be combined to form a pattern that matches an entire line only.

Note

You can not perform reverse searches in Grep.

Replacing with Grep

You can use Grep to search for strings, and also to replace them. The following special characters let you alter the replacement string:

- Each occurrence of the character `&` is replaced with whatever the entire pattern matched.

  For example, if you wanted to add a `P` to the beginning of every word that ended with `ptr`, you would search for `\<.*ptr\>` and replace it with `P&`.

- Each occurrence of `\`*n*, where *n* is one of the digits `1` through `9`, is replaced by whatever the *n*th occurrence of `\`*n* matched.

  For example, to change all strings such as `#define FOO 1` to `FOO = 1`, search for

```
#define \(\<[A-Za-z0-9][A-Za-z0-9]*\>\)
\(\<.*\>\)
```

and replace it with

```
\1 = \2
```

- Each occurrence of a string `\x`, where *x* is not one of the digits `1` through `9`, is replaced by *x*.

Grep examples

Grep is not easy to learn. Following are some typical examples to help you get started.

Suppose that you have written a Macintosh application and have forgotten to put a `\p` at the beginning of your strings to signal to the compiler to make them Pascal strings rather than C strings. You can change all your C strings to Pascal strings by specifying

```
"\([^"]*\)"
```

as the search pattern and

```
"\\p\1"
```

as the replacement string.

To convert

```
sym equ (expr+4)        ; a comment
```

to

```
#define sym (expr+4) /* ; a comment */
```

search for

```
\(\<.*\>\)[space tab]*\<equ\>\([^;]*\)\(.*\)
```

and replace with

```
#define \1 \2 /* \3 */
```

Explanation:

- `\<.*\>` matches a symbol.
- The surrounding `\(` and `\)` lets you use the symbol in the replacement string as `\1`.

- The `[`*space tab*`]*` matches any number of spaces or tabs between the symbol and the key word equ. (The words *space* and *tab* stand for the characters here, because you can't see them on paper. To enter a Tab, press Command-Tab in the dialog box.)

- `\<equ\>` matches the word equ. It does not match equ if it is part of another word, for example, `equal`. `\<equ\>` is not surrounded by `\(` and `\)` because it is thrown away in the replacement string.

- `[^;]*` matches an expression formed by any number of characters up to but not including a semicolon (`;`).

- The surrounding `\(` and `\)` lets you use the expression in the replacement string as `\2`.

- The `.*` matches the comment that is the rest of the line.

- The surrounding `\(` and `\)` stores the comment as `\3`.

- If there is no semicolon (`;`) in the line, then `\2` consists of everything after the equ to the end of the line, and `\3` is an empty string.

To convert `$HHHH` to `0xHHHH`, where `H` is a hexadecimal digit, search for

```
$\([0-9A-Fa-f][0-9A-Fa-f]*\)
```

and replaee it with

```
0x\1
```

Explanation:

- `$` matches a `$`.

- `[0-9A-Fa-f]` matches one hexadecimal digit.

- `[0-9A-Fa-f][0-9A-Fa-f]*` matches one or more hexadecimal digits. The pattern `[0-9A-Fa-f]*` matches zero or more hexadecimal digits.

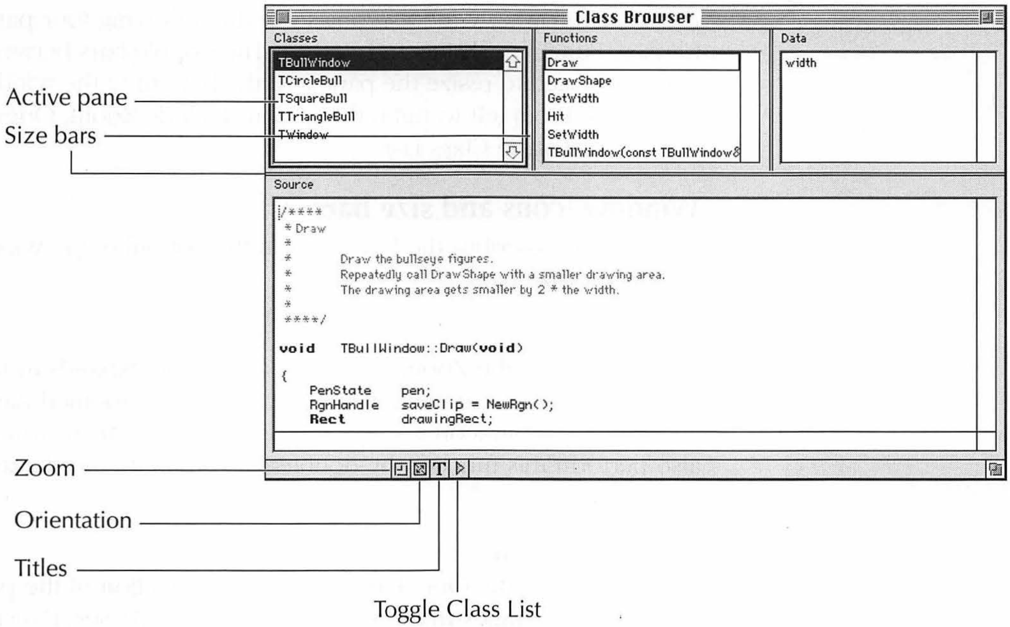
- The surrounding `\(` and `\)` lets you remember the hexadecimal digits in the replacement string as `\1`.

The Class Browser Window

22

The Class Browser displays a list of the classes in a project. You can view the list of your classes in either alphabetic or hierarchical order. You can use the Class Browser to view and edit each class's definition, and the source code for its data members and member functions.

To open a Class Browser window, choose **Class Browser** from the Symantec Project Manager's **Windows** menu (Command-J). If you choose this command with a Class Browser window already open, the window becomes the frontmost window.



```
/****
 * Draw
 *
 *        Draw the bullseye figures.
 *        Repeatedly call DrawShape with a smaller drawing area.
 *        The drawing area gets smaller by 2 * the width.
 *
 ****/

void    TBullWindow::Draw(void)

{
    PenState    pen;
    RgnHandle   saveClip = NewRgn();
    Rect        drawingRect;
```

Figure 22-1 Class Browser window

Note

To open a new Class Browser window when one or more Class Browser windows are already open for a project, hold down the Option key when choosing **Class Browser** from the **Windows** menu.

The Class Browser displays the classes for the active project at the time the browser is first opened. You cannot switch a Class Browser window to a different project after it has been opened. To examine the classes of a different project, make the other project the active project, then open a new Class Browser window.

This chapter discusses the components of the Class Browser window and how to use these components to examine the classes of a project. The first section describes the components of the window, covering the icons first and then the panes. The second section covers configuring the window and navigating the panes.

Components of the Window

The Class Browser window consists of the following four panes: Classes, Functions, Data, and Source. The double bars between the panes are used to resize the panes. At the bottom of the window are four icons. From left to right, these icons include Zoom, Orientation, Titles, and Toggle Class List.

Window icons and size bars

This section describes the four icons at the bottom of the window as well as the size bars.

Zoom icon

When you click the Zoom icon, the active pane expands to fill the entire window. If the window already contains a zoomed pane, clicking the Zoom icon restores the original four panes. You may also perform this function by double-clicking in the border around the pane.

Orientation icon

When you click the Orientation icon, the orientation of the panes in the window changes from horizontal (with the Classes, Functions, and Data panes along the top) to vertical with the panes along the left side of the window. If the window already is vertically aligned, clicking the Orientation icon returns it to the horizontal state.

Titles icon
You can toggle the display of the pane title on and off by clicking the Titles icon.

Toggle Class List icon
You can toggle the list of classes in the Classes pane between an alphabetic and a hierarchical order by clicking the Toggle Class List icon. Note that if the display is in hierarchical order, the derived classes are indented and listed after the class from which they are derived.

Note

> When a class is derived from more than one base class, the hierarchical class list displays that class under each base class from which it is derived.

Size bars
You may change the relative size of the panes by dragging the size bars. Once you have established a new relative size for a pane, it is maintained when the window is resized.

Panes of the window

The active pane, which has a black border around it, receives all menu commands and keystrokes. You may change the active pane by clicking in a different pane, or you may cycle through the four panes using Command-Tab to go forward and Command-Shift-Tab to go backward.

Classes pane
All the classes defined within the active project are listed in the Classes pane.

If you select a class in the Classes pane, the member functions it implements are displayed in the Functions pane. The data members it defines are displayed in the Data pane. Inherited member functions and data members are not displayed. You may display a class declaration in the Source pane by double-clicking its name in the Classes pane. To toggle the class list between alphabetic and hierarchical order, click the Toggle Class List icon at the bottom of the window.

When the display in the Classes pane is in hierarchical order, you can collapse and expand the derived classes using the following key combinations (Table 22-1).

Table 22-1 Key combinations for Classes pane hierarchical view

Command-Left Arrow	Collapse selected class
Command-Right Arrow	Expand selected class
Command-Option-Left Arrow	Collapse all derived classes
Command-Option-Right Arrow	Expand all derived classes

Note

Classes do not appear in the Classes pane until the source code file that contains their definition has been compiled at least once.

Classes defined in subprojects are not displayed in a project's Class Browser window—except for base classes from which one or more classes in a project are derived. These base classes are listed in italic in the Classes pane. You cannot examine these classes in the Functions, Data, or Source panes. To browse classes defined in a subproject, make the subproject the frontmost project, then open a new Class Browser window.

Function pane

The Function pane displays an alphabetical list of the member functions defined by the class currently selected in the Classes pane. If you double-click a member function's name, its definition is displayed in the Source pane.

Data pane

The Data pane displays an alphabetic list of the data members of the class currently selected in the Classes pane. If you double-click a data member's name, its definition is displayed in the Source pane.

Source pane

The Source pane displays the source code for a class declaration, a member function definition, or a data member definition. All editing operations available in source file Editor windows are available in the Source pane. Any editing operation done within the Source pane is synchronized with all open source file Editor windows.

When the Source pane displays code, you can open the file that contains that code in an Editor window. Hold down the Option key and choose **Open in Editor** from the **File** menu or press Command-Option-D.

Warning

> The Class Browser window does not automatically keep member function's declarations synchronized with their definitions. If you change a member function's declaration or definition in the Source pane, you must manually update the other part to match.

Working in the Class Browser Window

This section lists customization options for the Class Browser window and discusses the selection of items from lists in the Classes, Functions, and Data panes.

Configuring the window

You may configure the Class Browser window through the following actions:

- Zoom one pane to fill the whole window.
- Toggle the layout between horizontal and vertical.
- Toggle the display of pane titles on and off.
- Toggle the class list between alphabetic and hierarchical order.
- Change the relative size of the panes.

Note

Customizations, such as pane splits, are not saved when you close the Class Browser window.

Navigating the panes

The Classes, Functions, and Data panes are all lists. You scroll a list until an item is visible. You can also type the first few letters of the item's name and the list automatically scrolls to the first item that begins with those letters. After selecting an item in this manner, you can cycle the selection among all items in the list that begin with these letters using the Tab key (to move down the list) and Shift-Tab (to move up the list). You may also use the arrow keys to navigate the lists.

The Build Menu

23

This reference chapter documents the commands of the Program Manager's **Build** menu. The final section of this chapter describes the use of the Build Errors window.

Many of the commands in this menu, such as **Compile**, operate on the currently selected file or files. If the frontmost window is an Editor window, the selected file is the source file in that window. If the frontmost window is a Project window, all files currently selected in that Project window will be affected.

Commands in the Build Menu

The **Build** menu contains all the commands that turn source code into object code. You use the commands in this menu to precompile header files to allow for faster builds, to turn source code into object code, and to link the object code in a project into an application or library.

Figure 23-1 shows the commands available from this menu.

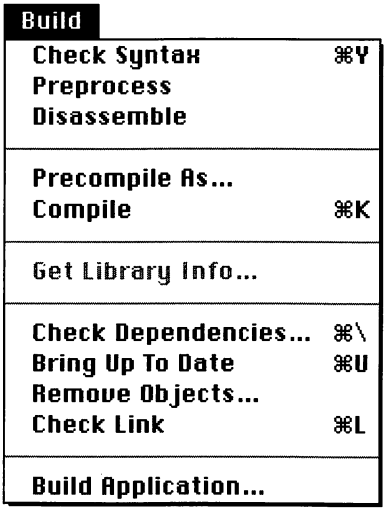


Figure 23-1 Build menu commands

You use the **Build** menu commands to perform the following primary functions:

- Analyze code
- Compile code
- Bring a project up-to-date
- Build the target

This chapter discusses the **Build** menu commands by function, as noted above. The final section covers the Build Errors window.

Analyzing code

These commands give you the ability to evaluate how the compiler sees your code as well as how it sees the assembly language code generated by the compiler.

Check Syntax ⌘Y

Lets you check the syntax of a file without generating code. This command processes the source code of a file as if it were being compiled but does not generate any object code and does not add the file to the project.

If errors are detected while the file's syntax is being checked, they are displayed in the project's Build Errors window. If the project's Build Errors window is not already open, the Program Manager opens it when an error is detected. For more information on the Build Errors window, see the last section of this chapter.

Preprocess

Processes the selected file's source code through the translator's preprocessor and displays the result in a new window. This command is useful for resolving problems with macros. The preprocessor expands your macros, adds the contents of the `#include` files, and evaluates the `#ifdef` statements. You can save and print the contents of this window as you would any other.

If the frontmost window is an Editor window, the command operates on the source code in that window. If the Project window is frontmost, this command operates on the currently selected file in the Project window.

If errors are detected while preprocessing the file, the project's Build Errors window opens to display them. For more information on the Build Errors window, see the last section of this chapter.

Disassemble

Opens a window displaying the assembly language code that the compiler is generating from the code in the frontmost window (or the currently selected file in the Project window, if that is frontmost). You can use this command to see the code that the compiler is generating. You can save and print the contents of this window as you would any other. If the Project window is frontmost, this command operates on the currently selected file in the Project window.

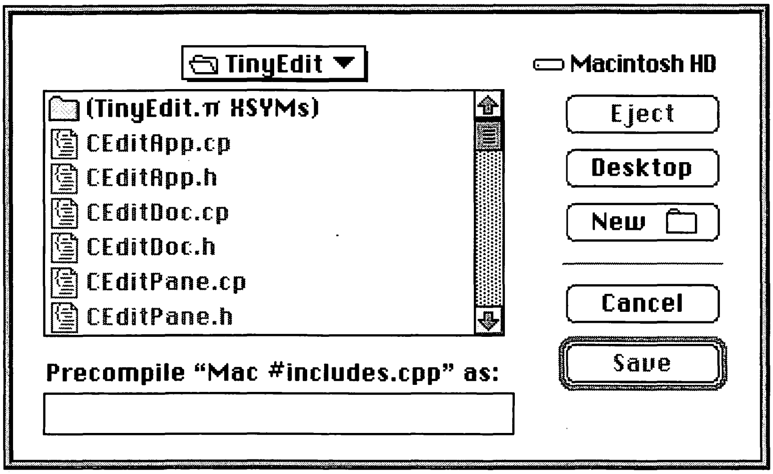
If errors are detected while disassembling the file, the project's Build Errors window opens to display them. For more information on the Build Errors window, see the last section of this chapter.

Compiling code

These commands deal with turning the elements of your project file into object code and turning the object code into an application or library. They all operate on the currently selected file if the frontmost window is a Project window or on the source code in the frontmost Editor window.

Precompile As

Opens the **Precompile** dialog box, which you use to process a source file to produce a precompiled header.



TinyEdit

(TinyEdit.π HSYMs)
CEditApp.cp
CEditApp.h
CEditDoc.cp
CEditDoc.h
CEditPane.cp
CEditPane.h

Precompile “Mac #includes.cpp” as:

Figure 23-2 Precompile dialog box

If you enter the name for your compiled header file and click Save, the source file is added to the project and the Project Manager registers that this source file produces the precompiled header file as well as the header files on which it depends. If one of the header files changes, the Project Manager automatically rebuilds the precompiled header file and updates build information for the project accordingly. Precompiled headers cannot contain code or data definitions. For more details on precompiled headers, see the *Symantec C++ Compiler Guide.*

If errors are detected while precompiling the file, the project's Build Errors window is opened to display them. For more information on the Build Errors window, see the last section in this chapter.

Compile
Precompile
Load
Update

Brings the currently selected file or files up-to-date. Depending on the kind of file selected, this command can take one of four different names.

Compile
If the frontmost window is an Editor window or a Project window with one or more source files selected, this command is named **Compile**. Selecting it compiles the file (or files, if more than one source file is selected in the Project window). If the frontmost window is an Editor window containing a file that is not already in

the Project window, the file is added to the project if it is successfully compiled.

Precompile
If a precompiled header file is selected in the Project window, this command is named **Precompile**. Selecting it precompiles the header file.

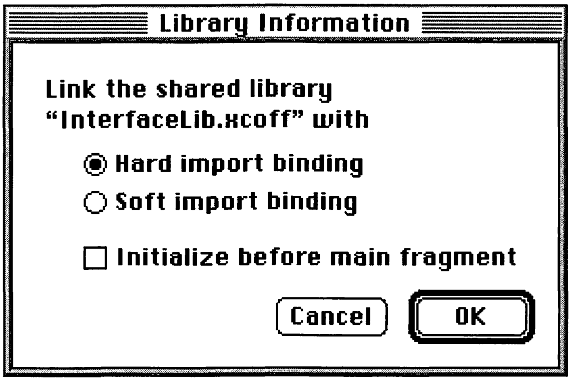
Load
If a shared library file is selected in the Project window, this command is named **Load**. Selecting it loads the library's object code into the Project file.

Update
If a subproject is selected in the Project window, this command is named **Update**. Selecting it causes the subproject to be brought up-to-date.

If errors are detected while updating a file, the project's Build Errors window opens to display them. For more information on the Build Errors window, see the last section of this chapter.

Get Library Info

Opens the **Library Information** dialog box (Figure 23-3), which you use to set the link parameters for the shared libraries in a project.



Link the shared library "InterfaceLib.xcoff" with

● Hard import binding

○ Soft import binding

□ Initialize before main fragment

Figure 23-3 Library Information dialog box

Binding options
You have two binding options: hard import and soft import.

Hard Import Binding. Generates a run-time error, thus preventing the application from running if the application attempts to link to a library that does not export one or more of the symbols (functions and variables) the application is expecting.

Soft Import Binding. If you select soft import binding, no run-time error is generated if your application attempts to link to a library that does not export all of the symbols your application is expecting. When you choose this option, testing all entry points (functions and globals) before using them becomes your responsibility.

For example, if your application calls the function `LibFunc` from the library, you might do something like this:

```
if   ( &LibFunc != 0 )// test function address
{  // function is in library, ok to call
result = LibFunc();
}
else { // function is not in library
   // work around absence of function
}
```

If you soft-link with a shared library that does not export one or more of the symbols imported by your application, the address of those symbols in the application will be set to zero. As a result, the application crashes if you call a function that the library does not export.

Initialize Before Main Fragment
If this option is on, the initialization code for the shared library is called before it is linked with an application at run-time.

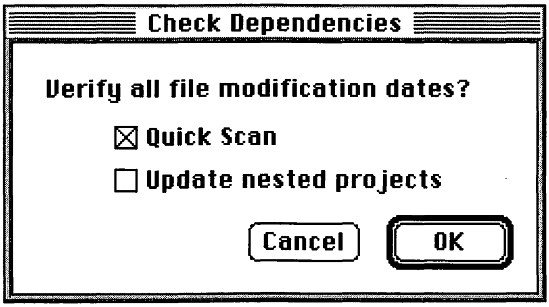
Bringing your project up-to-date

The commands described in this section operate on all the elements of a project. They are used to prepare the project for linking as well as for building the target.

Check Dependencies ⌘

Opens the **Check Dependencies** dialog box (Figure 23-4), which operates on the active project. This command examines the modification date of each file in the project to determine if the file needs to be updated. When finished, the make information for all files in the project is current, but no files have actually been brought up-to-date.

This is an appropriate step if you are sharing a file between projects and have edited it while the current project was closed.



Check Dependencies

Verify all file modification dates?

☒ Quick Scan

□ Update nested projects

Figure 23-4 Check Dependencies dialog box

Quick Scan
The modification dates of all files are rapidly checked. Enabling this option does not display any progress information, but you may interrupt the checking process with Command-Period. If **Quick Scan** is not checked, a dialog box displays the progress of the command as it does a slower check of the project's contents.

Update Nested Project
All nested projects are updated and their targets are built.

Bring Up To Date ⌘U

Brings all files in the active project up-to-date, as determined by their make status, and links them. This includes precompiling all precompiled headers, compiling all source code, and loading all libraries.

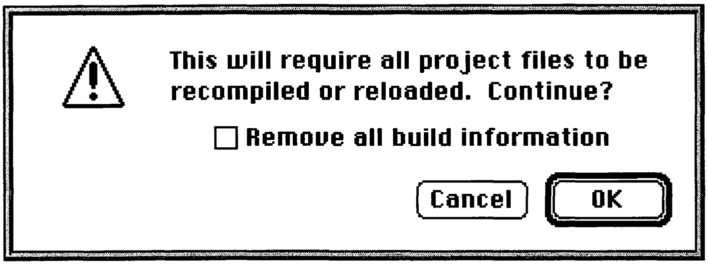
If the included project has debugging enabled (as indicated in the Project window), a special version of the library is built to permit debugging of code within the included project. Otherwise, the actual target specified in the included library's options is built.

If errors are detected while the project is being brought up-to-date, they are displayed in the project's Build Errors window. See the last section in this chapter.

Remove Objects

Operates on the active project. This command removes all object code and debug information generated by the compiler for each file in the project. This reduces the size of the Project file; every file in the project will need to be rebuilt.

When you select **Remove Objects**, the dialog box shown in Figure 23-5 opens.


This will require all project files to be recompiled or reloaded. Continue?

□ Remove all build information

Figure 23-5 Remove Objects dialog box

Remove All Build Information
Removes all information recorded for each file in the active project, including browser records and header file dependency information. Selecting this option saves space in the Project file, but features such as the **Headers** pop-up menu of the Class Browser and Editor windows are unavailable until the files are recompiled, and the rebuilding will be slightly slower.

Check Link ⌘L

Brings the active project up-to-date, then checks for link errors. If any errors are found, the Link Errors window for the project is opened to display them. This window contains all the unresolved references and multiple definitions for files in the project.

Building the target

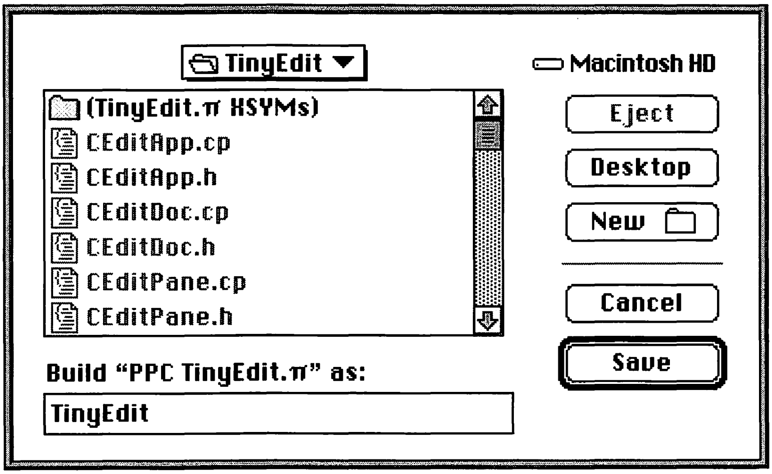
The Build Application and Build Library commands are used to build your project's target.

Build Application
Build Library

Brings the active project up-to-date and links it. If any errors are found during this process, they are displayed in the Build Errors window (see the next section). If there are no errors, the target is built.

This command is titled **Build Application** if the project type is an application, or **Build Library** if the project type is either a static library or a shared library.

If the Always Ask for Destination option on the Project Type page of the **Project Options** dialog box is set off, the target is named and placed automatically. If the option is set on, the user is prompted for the name and destination for the target with a standard **File Save** dialog box (Figure 23-6).



Build "PPC TinyEdit.π" as:

TinyEdit

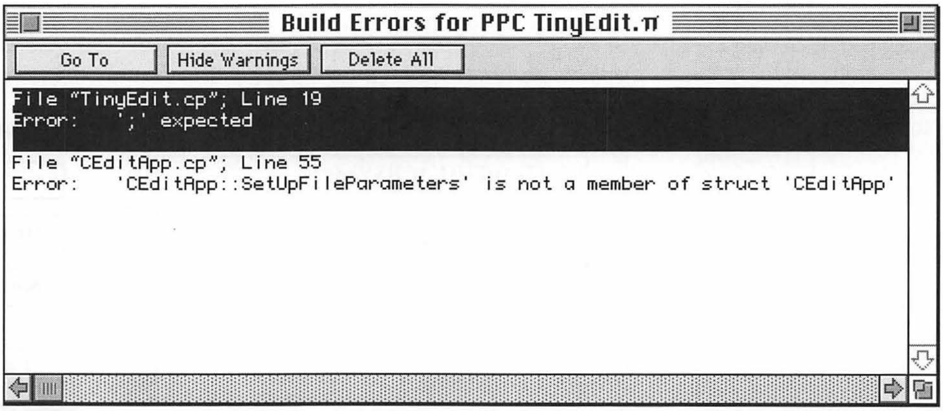
Figure 23-6 Build Application dialog box

You should name the application in the textbox provided. When you click Save, the Project Manager builds your double-clickable application and saves it with the name you entered in the dialog box.

Build Errors Window

When an error (or warning) occurs while a compiler compiles or precompiles a source code file, the Symantec Project Manager displays the compiler's message in the Build Errors window for that file's project. The Build Errors window opens automatically any time errors are detected during compilation. You can also open the window by choosing **Build Errors** from the **Windows** menu.

The contents of the Build Errors window are saved when the window is closed and are displayed again when the window is next opened.



```
File "TinyEdit.cp"; Line 19
Error:    ';' expected

File "CEditApp.cp"; Line 55
Error:    'CEditApp::SetUpFileParameters' is not a member of struct 'CEditApp'
```

Figure 23-7 Build Errors window

The Build Errors window displays compilation errors and warnings for all files in the project. Each error or warning is displayed on two lines: the first lists the file and line number where the error or warning occurred and the second gives the message itself.

Error messages and warning messages for all files in the project are listed in the Build Errors window. The most recent messages are displayed at the bottom of the list. If a file for which the Build Errors window was displaying errors or warning messages is recompiled, the existing messages are deleted from the window and any new messages are added at the end of the list.

The Build Errors window displays three buttons on its toolbar that initiate actions on error messages:

- Go To
- Hide/Show Warnings
- Delete All

Go To

Clicking this button takes you to the source code for the currently selected message. It opens the source file and scrolls the display to show the line that contains the error or warning. You can also go to the source file for any message in the Build Errors window by double-clicking on the message.

Hide/Show Warnings

This button is named Hide Warnings if warnings currently are being displayed in the Build Errors window. It is named Show Warnings if they are hidden.

Clicking on Hide Warnings causes the Build Errors window to hide all compiler warning messages. The messages are still stored and may be shown at any time by clicking on Show Warnings.

Delete All

Clicking this button permanently erases all messages in the Build Errors window.

The Debugger Windows

24

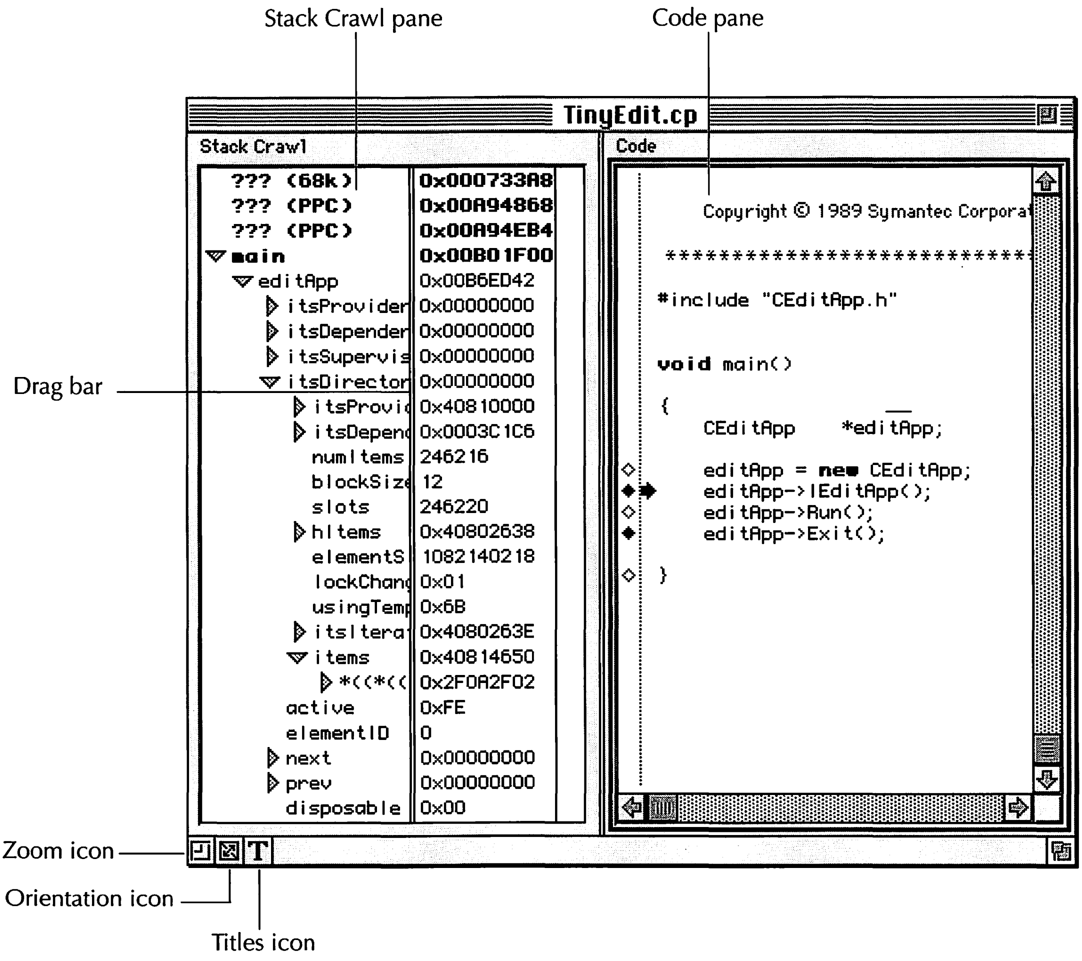
This chapter describes the Symantec Debugger's windows. With these windows, you can browse through source code as it runs, set breakpoints, and examine the values of variables.

The Debugger provides five different kinds of windows: the Main debugging window, Debug Browser windows, the Data window, the Control palette, and the Debugger Log window. Between debugging sessions, the Debugger retains the location and arrangement of any open windows.

Note that the Debugger windows and menus are separate from those of the Symantec Project Manager. For details on the menus, see Chapter 25, "The Debugger Menus."

The Main Debugging Window

Starting a debugging session opens the Main debugging window. This window is shown in Figure 24-1.



```
??? (68k)              0x000733A8
??? (PPC)              0x00A94868
??? (PPC)              0x00A94EB4
▽ main                 0x00B01F00
  ▽ editApp            0x00B6ED42
     ▷ itsProvider     0x00000000
     ▷ itsDepender     0x00000000
     ▷ itsSupervis     0x00000000
     ▽ itsDirector     0x00000000
        ▷ itsProvi     0x40810000
        ▷ itsDepen     0x0003C1C6
          numItems     246216
          blockSiz     12
          slots        246220
        ▷ hItems       0x40802638
          elementS     108214021 8
          lockChan     0x01
          usingTem     0x6B
        ▷ itsItera     0x4080263E
        ▽ items        0x40814650
           ▷ *((*((     0x2F0A2F02
       active          0xFE
       elementID       0
     ▷ next            0x00000000
     ▷ prev            0x00000000
       disposable      0x00
```

```
        Copyright © 1989 Symantec Corpora
 **************************

 #include "CEditApp.h"


 void main()

 {
         CEditApp      *editApp;

◇        editApp = new CEditApp;
◆➧       editApp->IEditApp();
◇        editApp->Run();
◆        editApp->Exit();

◇ }
```

Figure 24-1 Main debugging window

This window is a multipaned window such as those used by the Class Browser in the Project Manager. This window contains two panes, the Code pane and the Stack Crawl pane, that display information about the process currently being debugged. Individual panes are printable. This window contains no close box and remains open throughout a debugging session.

To change the relative sizes of the panes, drag on the bar that separates the panes. To hide either of the panes, select the other pane and click the Zoom icon at the lower left of the window. To change the relative orientation of the panes, click the Orientation icon. To hide the titles of the panes, click the Titles icon. For more information, see Chapter 6, "Using the Debugger."

Code pane

This pane is used to browse through code as it runs. The current statement indicator shows the location of the program counter in the execution of the code. It always points at the next statement that will be executed.

Note

> If you launch the Debugger without having a project open, the Code pane displays the message "No debugging information available," instead of the application's source code. In addition, the window title bar displays the word "Source," rather than your source file's name.
>
> This also happens if execution steps into a file for which no debugging information is available.

You can set a breakpoint by clicking on the diamond to the left of the line or by selecting the line and choosing **Set Breakpoint** from the **Source** menu (Command-B).

If you hold down the Option key while clicking on a diamond, the Debugger sets a temporary breakpoint, then begins executing the code.

You can use the **Go To Line** and **Go To Marker** dialog boxes with this pane, just as you would in the Project Manager. You can also open the **Markers** pop-up menu by Command-clicking on the title bar and the **Headers** pop-up menu by Option-clicking on the title bar, just as you would in an Editor window. This works only when a Code pane is visible. Only debuggable functions are listed in the **Markers** pop-up menu.

Stack Crawl pane

This pane contains a list of all active stack frames for the current execution point in the code. Stack frames are expandable items that contain all of the local variables for the frame being displayed. Each stack frame is described by the name of the function at that frame (if known) and the location (in hexadecimal) of the program counter for the selected frame. You can use the Stack Crawl pane to look at variables that are not in the current frame.

Double-clicking on a stack frame causes its location to be displayed in any Code panes that are displayed in the same Main or Debug Browser window.

Holding down the Option key while clicking on the triangle adjacent to a variable's name expands all of the variable's subitems. These are shown immediately below the stack frame itself.

Note

> Expanding all of the variable's subitems can be a time-consuming operation.

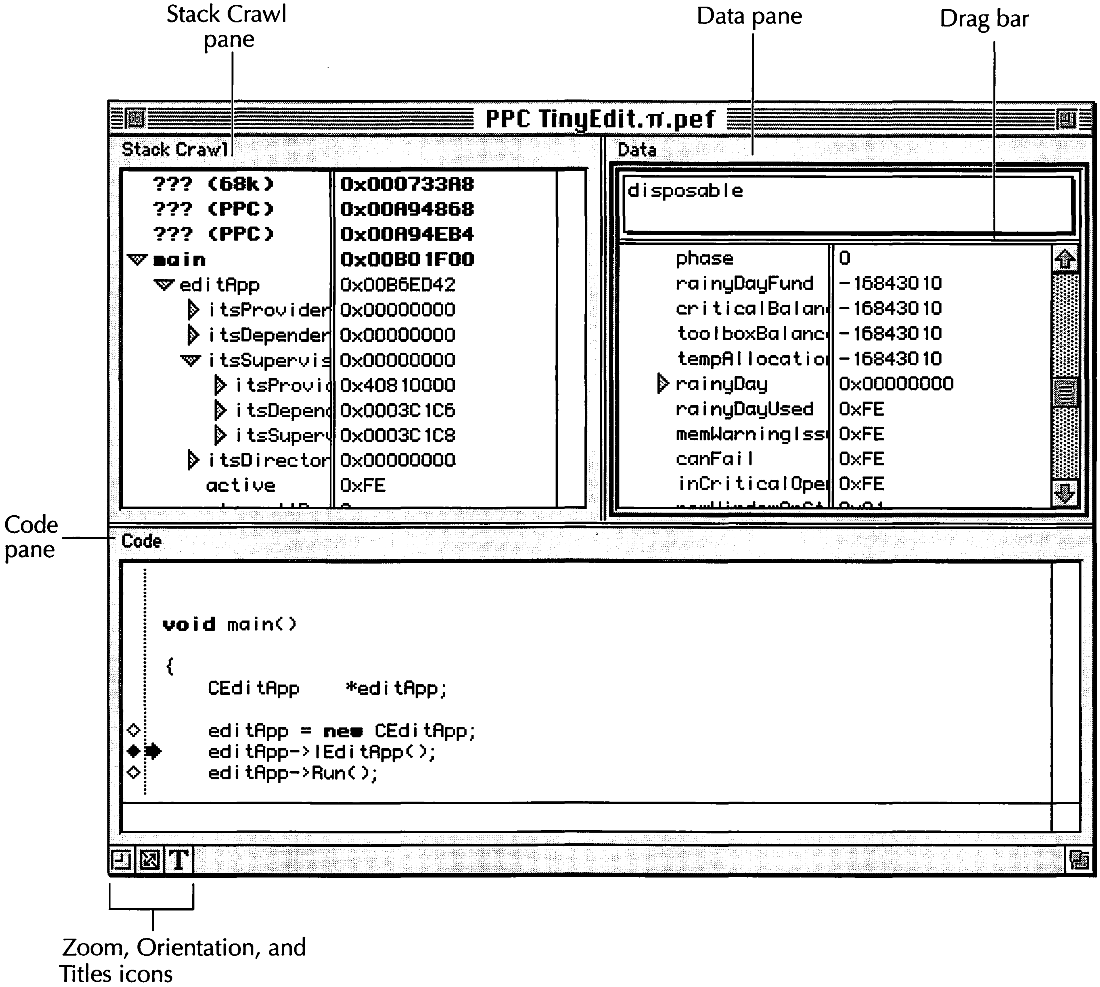
If the Debugger cannot find a corresponding source file for a stack frame in the project, it tries to map it to a name using other low-level information. If this doesn't succeed, the Debugger identifies the frame as either a 68K or Power Macintosh frame with a name displayed as either `???(68K)` or `???(PPC)`.

Arrays are displayed using the declared array bounds, if bounds are present. If no bounds are present, the Debugger uses a default array size of ten elements. You can change the array size and starting index by double-clicking on the array's name or by using the **Set Array Bounds** command from the **Data** menu. See **Set Array Bounds** in the section "Data Menu," in Chapter 25, "The Debugger Menus."

Debug Browser Windows

The Debugger lets you open auxiliary Debug Browser windows to give you maximum flexibility when debugging. You open a Debug Browser window by choosing **New Browser** from the Debugger's **File** menu (Command-N).

A sample Debug Browser window is shown in Figure 24-2.



```
??? (68k)        0x000733A8
??? (PPC)        0x00A94868
??? (PPC)        0x00A94EB4
▽ main           0x00B01F00
  ▽ editApp      0x00B6ED42
    ▷ itsProvider 0x00000000
    ▷ itsDepender 0x00000000
    ▽ itsSupervis 0x00000000
      ▷ itsProvi  0x40810000
      ▷ itsDepen  0x0003C1C6
      ▷ itsSuper  0x0003C1C8
    ▷ itsDirector 0x00000000
      active      0xFE
```

```
disposable

  phase           0
  rainyDayFund    -16843010
  criticalBalan   -16843010
  toolboxBalanc   -16843010
  tempAllocatio   -16843010
▷ rainyDay        0x00000000
  rainyDayUsed    0xFE
  memWarningIss   0xFE
  canFail         0xFE
  inCriticalOpe   0xFE
```

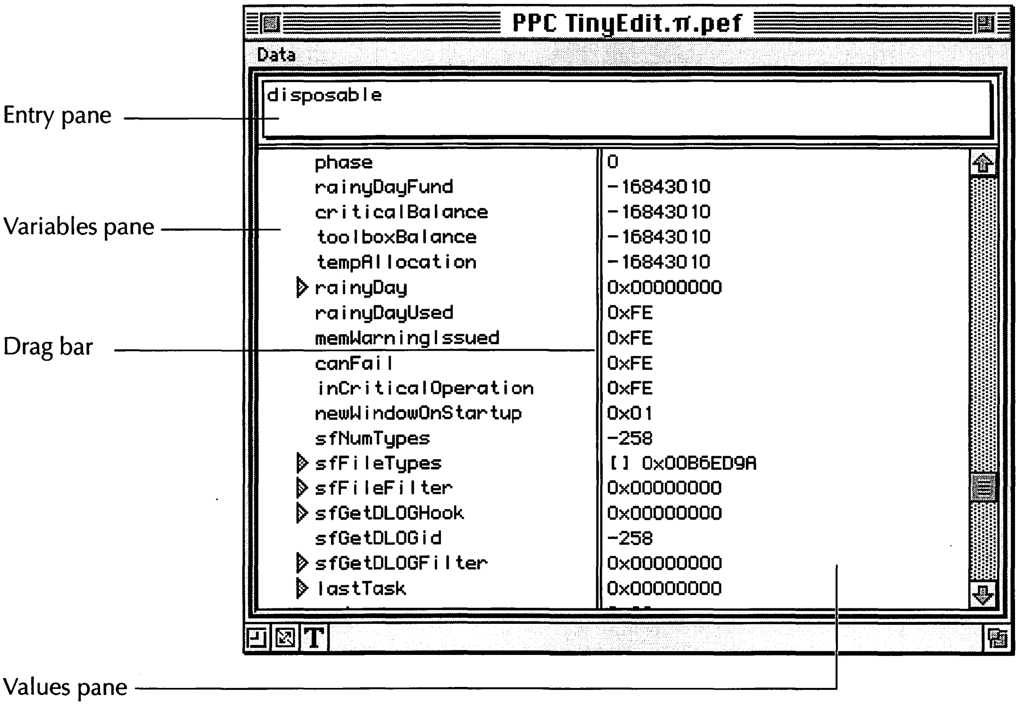
```
   void main()
   {
       CEditApp     *editApp;

◇      editApp = new CEditApp;
◆➜     editApp->IEditApp();
◇      editApp->Run();
```

Figure 24-2 Debug Browser window

These windows contain three panes: Code and Stack Crawl panes (as does the Main debugging window) and a Data pane. You can use the drag bars and the Zoom, Orientation, and Titles icons as in the Main debugging window.

Data pane

This pane is used to browse through variables. A sample Data pane is shown in Figure 24-3.



disposable

phase	0
rainyDayFund	-16843010
criticalBalance	-16843010
toolboxBalance	-16843010
tempAllocation	-16843010
▶ rainyDay	0x00000000
rainyDayUsed	0xFE
memWarningIssued	0xFE
canFail	0xFE
inCriticalOperation	0xFE
newWindowOnStartup	0x01
sfNumTypes	-258
▶ sfFileTypes	[] 0x00B6ED9A
▶ sfFileFilter	0x00000000
▶ sfGetDLOGHook	0x00000000
sfGetDLOGid	-258
▶ sfGetDLOGFilter	0x00000000
▶ lastTask	0x00000000

Figure 24-3 Data pane

This pane is subdivided into three smaller units: an entry pane at the top, a variables pane at the bottom left, and a values pane at the bottom right. You can change the relative sizes of these panes by dragging on the drag bars.

To enter expressions in the current line's or a selected line's context, type them in the entry pane and press Return or Enter. The expression itself is shown in the variables pane and its value is shown in the values pane.

To copy an expression selected in the Code pane or Stack Crawl pane into the Data pane, choose **Copy To Data** from the Debugger's **Edit** menu (Command-D).

Choosing **Show Context** in the Debugger's **Data** menu when an expression is selected in the Data pane selects the expression's context in the Code pane. This is the line of code in which the Debugger evaluates the expression.

Use the Clear or Esc key to remove expressions from the Data pane.

Data Window

This window works like the Debug Browser window except that it contains only a Data pane. A Data pane is used to evaluate any expression—for example, to find the value of a variable, an array element, or an element in a record.

To copy a selected expression into the Data window and bring the Data window to the foreground, choose **Copy To Data** from the Debugger's **Edit** menu (Command-D). You can do this from the Code pane or Stack Crawl pane of any Main or Debug Browser window that does not have a Data pane.

Choosing **Show Context** from the Debugger's **Data** menu when an expression is selected in the Data window brings the Main debugging window to the foreground and selects the expression's context. This is the line of code in which the Debugger evaluates the expression.

Use the Clear or Esc key to remove expressions from the Data window.

Control Palette

The Control palette contains buttons that both control the current process and reflect the state of that process. The **Go** button is lit when the process is running and the **Stop** button is lit when the process is stopped. You can hide the Control palette by choosing **Hide Control Palette** from the **Windows** menu. The Control palette is shown in Figure 24-4.

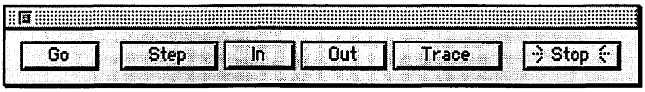


Figure 24-4 Control palette with the Stop button lit

Go

Starts execution of the code, if it was stopped, or brings your application to the foreground, if the application was already running. Execution continues until a breakpoint or exception (such as an illegal instruction) is reached or until you stop it.

Choosing **Go** from the Debugger's **Debug** menu (Command-G) also executes this command.

Step

Executes the current statement. Execution goes on to the next statement in the current function. (See **Trace** if you want execution to go on to the next statement regardless of the function it is in.) If you are at the end of a function, **Step** returns to the calling function. The first time you step through a `for` loop, for example, execution jumps to the end of the loop, as that's where the test is done in the generated code.

Use this button when you want to follow the execution within a function without falling into a different function. Choosing **Step** from the Debugger's **Debug** menu (Command-S) also executes this command.

In

Executes **Trace** commands until execution falls into a function. This is useful when you want to skip over a set of assignments to fall into the next function. If execution reaches the last statement of the current function without falling into another function, it stops immediately after the function returns.

Choosing **Step In** from the Debugger's **Debug** menu (Command-I) also executes this command.

Out

Executes **Step** commands until execution falls out of the current function. You can also leave the current routine by choosing **Go Until Here** from the Debugger's **Debug** menu (Command-H).

Choosing **Step Out** from the Debugger's **Debug** menu (Command-O) also executes this command.

Trace

Executes the current statement. In most cases, execution will go on to the next statement, even if the next statement is in a different function. (See **Step** if you want execution to remain within the same function.) The exception is when execution steps into code for which the Debugger does not have the source text. This happens when you step into a function that has no corresponding Project file; for a brief period, execution is somewhere in that library, rather than in your code.

Choosing **Trace** from the Debugger's **Debug** menu (Command-T) also executes this command.

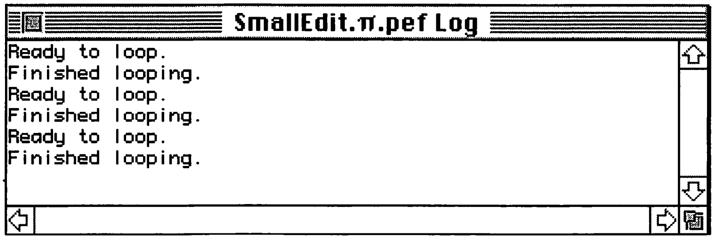
Stop

Stops execution of code. Choosing **Stop** from the Debugger's **Debug** menu (Command-Period) also executes this command.

Debugger Log Window

The Debugger Log window contains a list of `DebugStr()` messages from the current process. `DebugStr()` is a function that causes the execution of code to stop and drop into the Debugger just as if you had set a breakpoint. You can use it in conditional statements to report as well as to avoid situations in which you would generate an exception, such as dereferencing an illegal address. `DebugStr()` takes a Pascal string as an argument.

A sample Debugger Log window is shown in Figure 24-5.



```
Ready to loop.
Finished looping.
Ready to loop.
Finished looping.
Ready to loop.
Finished looping.
```

Figure 24-5 Debugger Log window

You can save the Debugger Log window's contents to a file. If the last two characters in the string passed to `DebugStr()` are either `;g` or `;G`, then the Debugger logs the message and automatically continues executing the current process.

The Debugger Menus

25

This chapter describes the six menus of the Symantec Debugger. You use the menus listed below to test and debug applications:

- File
- Edit
- Debug
- Source
- Data
- Windows

The Debugger's windows are described in Chapter 24, "The Debugger Windows."

File Menu

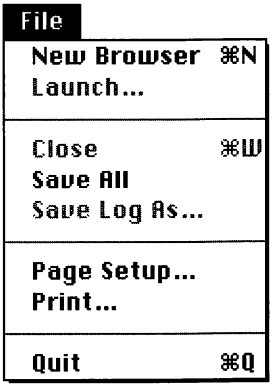


Figure 25-1 File menu

New Browser ⌘N

Opens a new Debug Browser window. See Chapter 24, "The Debugger Windows," for details.

Launch

Opens the **Launch** dialog box in which you can choose an application to be launched from the Debugger (Figure 25-2).

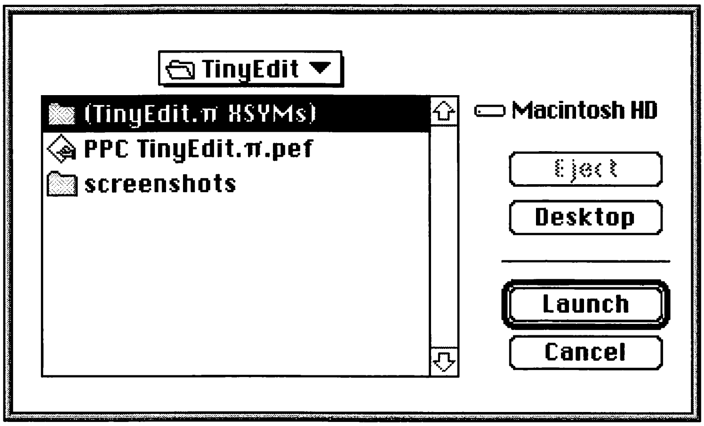


Figure 25-2 Launch dialog box

Click Launch to launch an application. The Debugger automatically stops execution at the beginning of the application.

Because only one project can be debugged at a time, you can choose this command only when you are not debugging another project. **Launch** is enabled if you launch the Debugger from the Finder and click Cancel in the **File Open** dialog box. It is also enabled if an application that you are debugging quits and you have not selected the Quit Debugger When Application Quits option on the **Debugger Preferences** dialog box. See the section "Edit Menu" later in this chapter for details on how to set preferences.

Close ⌘W

Closes the frontmost window. You can also close a window by clicking its close box, if it has one.

Save All

Saves your current debugging session, including the breakpoints and the contents of the Data window and Data pane, so the Debugger can restore the session when you next use it. If you set the Save Expressions and Breakpoints option on the **Debugger Preferences** dialog box, the Debugger saves the session whenever you quit. See the section "Edit Menu" later in this chapter for details on how to set preferences.

Save Log As

Saves the Debugger Log window. See Chapter 24, "The Debugger Windows," for details on that window.

Page Setup

Opens a standard **Page Setup** dialog box, in which you set up.pages for printing.

Print

Opens a standard **Print** dialog box for printing a source file.

Quit ⌘Q

Quits the Debugger. If you quit the Debugger while debugging, the Debugger quits the application by calling `ExitToShell()`.

Edit Menu

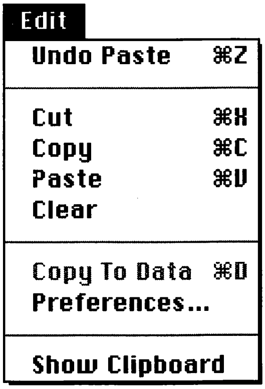


Figure 25-3 Edit menu

Undo ⌘Z

Removes the effect of the most recent edit operation in the Data window or a Data pane. The name of this command changes to **Redo** after you choose it and remains **Redo** until you perform another editing operation.

Cut ⌘X

Deletes the selected text from the entry field of a Data window or Data pane and copies it to the Clipboard. Note that you cannot cut text out of a Code pane.

Copy ⌘C

Copies the selected text to the Clipboard.

Paste ⌘V

Pastes the text in the Clipboard into the entry field of a Data window or Data pane. You cannot paste into a Code pane.

Clear

Removes the selected expression from a Data window or Data pane without placing it on the Clipboard. You cannot clear text from a Code pane. You can also execute this command by pressing Clear or Esc.

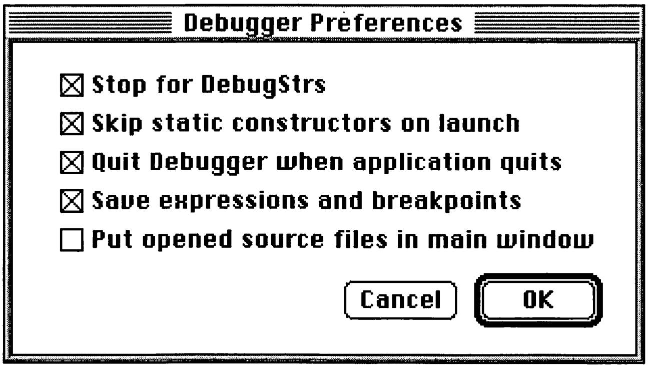
Copy to Data ⌘?

Copies a selected expression from a Code pane or a Stack Crawl pane, compiles and evaluates the expression, then pastes the expression and its value directly into a Data window or Data pane. If the selected expression is in the Code pane or the Stack Crawl pane of a Debug Browser window, the command pastes it into the Data pane of the same window.

If the selected expression is in the Code pane or the Stack Crawl pane of the Main debugging window, the command opens the Data window, if it is not yet open, and pastes the expression into that window. If the Data window is open, the window is brought to the foreground before pasting occurs. This command does not affect the Clipboard.

Preferences

Opens the **Debugger Preferences** dialog box (Figure 25-4).



Debugger Preferences

☒ Stop for DebugStrs
☒ Skip static constructors on launch
☒ Quit Debugger when application quits
☒ Save expressions and breakpoints
□ Put opened source files in main window

Figure 25-4 Debugger Preferences dialog box

Stop for DebugStrs
The Debugger stops whenever a `DebugStr()` call is hit and displays the message passed to `DebugStr()` in the Debugger Log window. See Chapter 24, "The Debugger Windows," for details on the Debugger Log window. The default setting for this option is on.

When this option is set off, the Debugger logs the message, leaves the window hidden or displayed (depending on whether the command **Show Log Window** has been chosen in the Debugger's **Windows** menu), then automatically continues executing the code. You may want to set the option off if you are using `DebugStr()` to collect a large amount of information and do not want to see each message.

Skip Static Constructors on Launch
The Debugger automatically sets a breakpoint in your `main()` routine, and executes a **Go** command past your static constructors. The default setting is on.

When the option is set off, the Debugger stops the application on loading before any code (including any initialization code) has been executed. You may want to leave the option off if you need to debug your static constructors or if you want to debug the loader code that starts up the application. If you have any `Debugger()` or `DebugStr()` calls in static constructors or if the application crashes before reaching `main()`, the debugger comes up at that point instead of at `main()`.

Quit Debugger When Application Quits
The Debugger quits when the application quits. The default setting is on.

When the option is set off, the Debugger remains running even after the target application exits. You may want to leave the option off if you need more memory for compiling when you have finished running your target.

Save Expressions and Breakpoints
The Debugger saves the expressions and breakpoints in active windows whenever it quits. The default setting is on.

When the option is set off, the expressions and breakpoints are not saved. The current set of breakpoints and expressions can be saved at any time by using the **Save All** command in the **File** menu.

Put Opened Source Files in Main Window
The Debugger puts files that are opened with the Symantec Project Manager's **Debug File** command into the Main debugging window.

When the option is set off, the Debugger opens a new debugging window for each source file that you open. The default setting is off.

Show Clipboard

Displays the current contents of the Clipboard.

Debug Menu

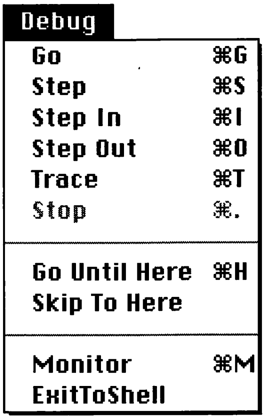


Figure 25-5 Debug menu

You can access the first six commands on this menu from the Debugger's Control palette. See Chapter 24, "The Debugger Windows," for details.

Go Until Here ⌘H

Starts execution of code and stops at the selected line. Choosing this command has the same effect as setting a temporary breakpoint at the selected line. Use this command when you want to move through a block of code quickly. Option-clicking a breakpoint also executes this command.

Skip To Here

Jumps ahead or back to the selected line without executing any intervening code. Use the command when you want to skip over code that you know contains bugs but that is not crucial to the rest of the code's operation.

Note

> **Skip To Here** is potentially dangerous, especially when debugging optimized code.

Monitor ⌘M

Drops you into a low-level debugger (such as MacsBug) in the context of your application. At this point, you can use the debugger to examine memory or heap structures for your application. The application's heap will be the current heap, and low-memory globals will be correct. When you are using the low-level debugger, exit the debugger normally (in MacsBug, use Command-G).

Note

You can use the **Monitor** command only if you have a low-level debugger installed.

ExitToShell

Quits the debugger. This command may bypass clean-up operations in some applications. Use it only when you cannot use your application's **Quit** command.

Source Menu

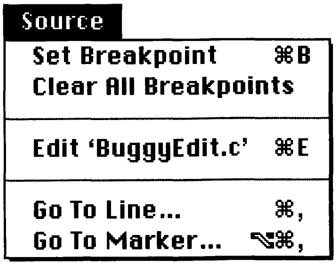


Figure 25-6 Source menu

Set Breakpoint ⌘B

Clear Breakpoint

Clears or sets a breakpoint at the selected statement in a Code pane. If you have not set a breakpoint at the selected line, the command reads as **Set Breakpoint**. If you have set a breakpoint, you see **Clear Breakpoint**. Another option for setting or clearing a breakpoint is to click the diamond to the left of the line.

Clear All Breakpoints

Clears all breakpoints in a project.

Edit *filename* ⌘E

Brings the Symantec Project Manager to the foreground and opens an Editor window for the file in the Code pane. That file's name appears in the command. This command is the inverse of the **Debug File** command in the Symantec Project Manager's **Project** menu. See Chapter 18, "The Project Menu," for details on that command.

Go To Line ⌘,

Opens the **Go To Line** dialog box (Figure 25-7), which you use to scroll to a line in your code.

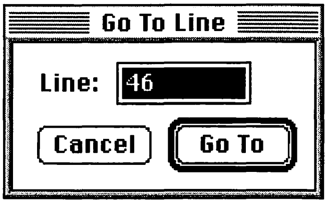


Figure 25-7 Go To Line dialog box

Go To Marker ⌥⌘,

Opens the **Go To Marker** dialog box (Figure 25-8), which you use to scroll to a marker in the code. Click the marker you want, then click Go To. Alternatively, double-click the marker's name in the dialog box.

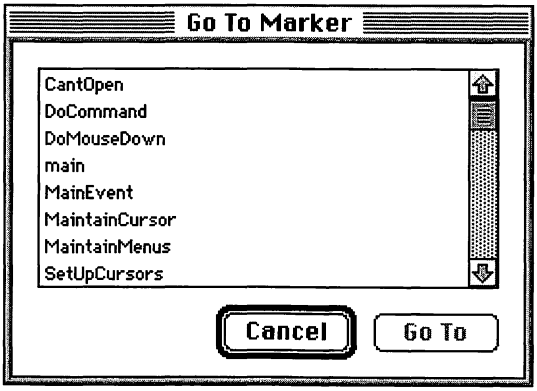


Figure 25-8 Go To Marker dialog box

Data Menu

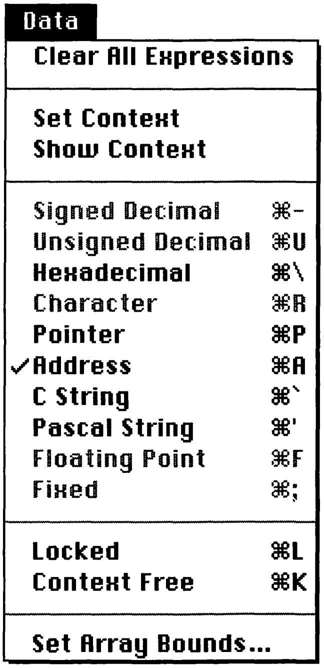


Figure 25-9 Data menu

Clear All Expressions

Clears the contents of the Data window or a Data pane. This command is especially useful if you do not want the Debugger to restore the contents of your Data window or Data panes the next time you use the Debugger.

Set Context

Makes the selected statement in a Code pane the context of the selected expression in the Data window or Data pane.

Show Context

Highlights the statement in a Code pane that is the context of an expression that is selected in the Data window or a Data pane.

Data formatting commands

The next ten commands control how expressions are displayed in the Data window or a Data pane. The default format depends on the data type of the expression. The type of the expression also determines the other formats you can use. The format for the currently selected expression has a checkmark next to it in the menu. Only those formats that can be used with the selected expression are available in the **Data** menu.

Signed Decimal ⌘-

Displays a selected expression's value in signed decimal format, in other words, as a positive or negative integer. Examples of this format include `4523345` and `-1`. This is the default format for integer-type expressions.

Unsigned Decimal ⌘U

Displays a selected expression's value in unsigned decimal format, in other words, as a positive integer. Examples of this format include `4523345` and `65535`. This is the default format for unsigned-type expressions.

Hexadecimal ⌘\

Displays a selected expression's value in hexadecimal format. An example of this format is `0xA09E1487`. This is not a default format for any standard type.

Character ⌘R

Displays a selected expression's value in character format. Examples of this format are `'c'` and `'TEXT'`. This is not a default format for any standard type.

Pointer ⌘P

Displays a selected expression's value in pointer format. An example of this format is `0x007A7000`. This is the default format for pointer-type expressions.

Address ⌘A

Displays a selected expression's value in address format. Examples of this format are `[] 0x0009FE44` and `struct 0x0008FC14`. This is the default format for array-, structure-, union-, and address-type expressions.

C String ⌘`

Displays a selected expression's value in C string format. An example of this format is `"hello\n"`. This is not a default format for any standard type.

Pascal String ⌘'

Displays a selected expression's value in Pascal string format. An example of this format is `"\phello\n"`. This is not a default format for any standard type.

Floating Point ⌘F

Displays a selected expression's value in floating-point format. An example of this format is `1961.0102`. This is the default format for floating-type expressions.

Fixed ⌘;

Displays a selected expression's value in fixed format. An example of this format is `1961.0102`. This is the default format for fixed-type expressions.

Locked ⌘L

Re-evaluates all expressions in the Data window or a Data pane every time execution stops. To keep an expression from being re-evaluated, select it, then choose this command. A small lock icon is displayed next to the expression.

To unlock a locked expression, select the expression, then choose **Locked** again.

Note

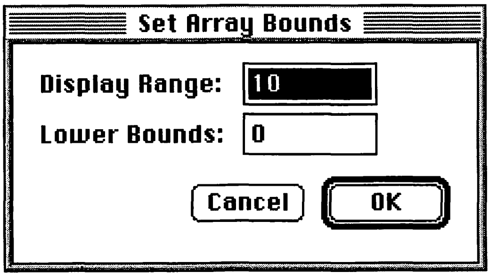
> You cannot lock fields of expanded items without explicitly adding them to the Data pane. For example, to lock the `top` field of `Rect r`, enter `r.top` into the Data pane and lock it.

Context Free ⌘K

Makes an expression context free. For example, if you are using the same variable name in several routines and you would like to see the value of the variable whenever you enter one of those routines, select the variable in the Data window or a Data pane, then choose this command. The Debugger marks the variable with a small arrow to tell you that the variable's context is the same as that of the current statement.

Set Array Bounds

Opens the **Set Array Bounds** dialog box, which you use to set the bounds of an array.



Display Range: 10

Lower Bounds: 0

Figure 25-10 Set Array Bounds dialog box

Enter the number of items in the array and the base index of the array and click OK.

You can also access this dialog box by double-clicking the array. Arrays are displayed using the declared array bounds, if bounds are present. If no bounds are present, the Debugger uses a default array size of ten elements. Array size changes do not take place until the array is closed and reopened.

Set Array Bounds is available only when you have selected an expression that is formatted as an array in a Data or Stack Crawl pane. Double-clicking on the expression or choosing the command has the same effect. Note that you can use this command for pointer expressions that have been formatted as arrays. An expression displayed as an "address" can also be expanded as an array.

Windows Menu

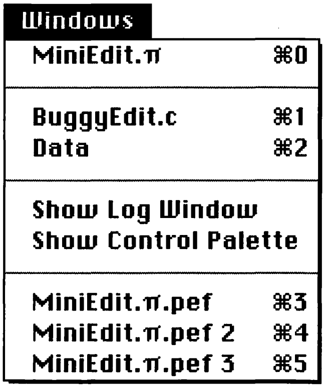


Figure 25-11 Windows menu

Filename.π ⌘0

Brings the Symantec Project Manager to the foreground and makes the Project window that is listed in the menu active.

Source file ⌘1

Makes the Main debugging window active. The Main window contains the source file listed in the menu.

Data ⌘2

Makes the Data window active.

Show Log Window
Close Log Window

Opens or hides the Debugger Log window. When you choose **Show Log Window**, the Debugger Log window is displayed and the command changes to **Close Log Window**. See Chapter 24, "The Debugger Windows," for details on that window.

Show Control Palette
Hide Control Palette

Opens or hides the Control palette. When you choose **Show Control Palette**, the Control palette is shown and the command changes to **Hide Control Palette**. See Chapter 24, "The Debugger Windows," for details on the Control palette.

Debug browsers

Makes a Debug Browser window active for the project you are debugging. A debug browser entry is added to the menu each time you open a Debug Browser window, and an entry is deleted each time you close such a window. These windows are assigned Command-keys 3 through 9, as needed.

The Windows Menu

26

This reference chapter provides detailed descriptions of the commands in the Symantec Project Manager's **Windows** menu. The **Windows** menu contains commands to help manage the windows you open in the Symantec Project Manager.

Commands in the Windows Menu

You use the commands in the **Windows** menu to size, order, and arrange open windows. You also use these commands to open the Build Errors, Search Results, Class Browser, and Worksheet windows. Figure 26-1 displays the commands in the **Windows** menu.

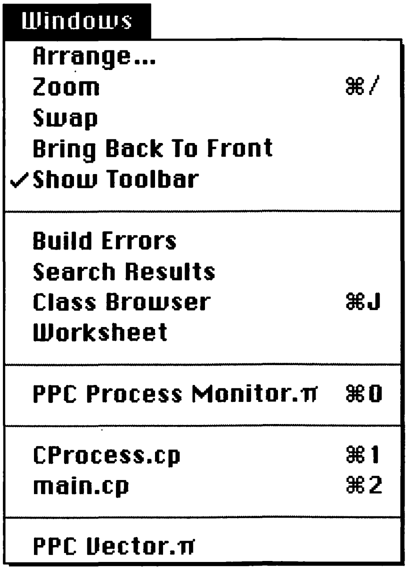


Figure 26-1 Windows menu

These commands primarily are used to perform the following functions:

- Arrange windows
- Open windows

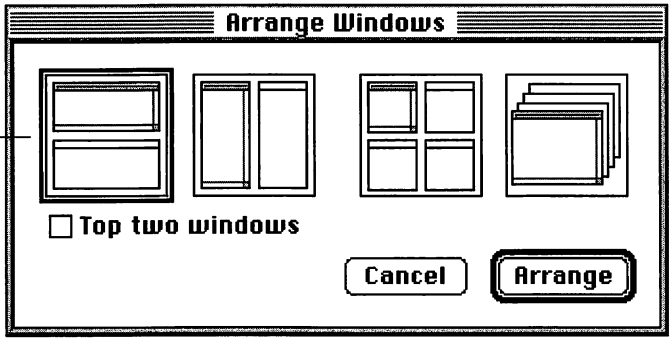
This chapter discusses the **Windows** menu commands by function in the above order.

Arranging windows

These five commands determine the arrangement and size of open windows and let you choose to hide or display the toolbar of the frontmost window.

Arrange

Opens the **Arrange Windows** dialog box (Figure 26-2), in which you can rearrange open Editor windows on the screen.

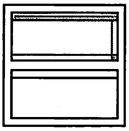


Current selection

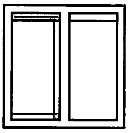
Figure 26-2 Arrange Windows dialog box

The four pictures in the dialog box represent possible arrangements for open Editor windows. To select an option, double-click on its picture. Alternatively, click once on a picture, then press Return (or click the Arrange button). To exit the dialog box without rearranging any windows, click Cancel.

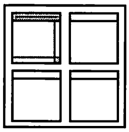
You can use the **Arrange** command to resize and reposition only Editor windows and the Worksheet window. It cannot be used on a Project window, a Class Browser window, or special windows such as Build Errors or Search Results.



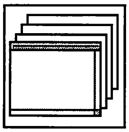
Top to bottom
This option tiles all open Editor windows vertically. If the Top Two Windows option is set on, only the frontmost two Editor windows are resized and repositioned. The window order (front to back) is unchanged.



Left to right
This option tiles all open Editor windows horizontally. If the Top Two Windows option is set on, only the frontmost two Editor windows are resized and repositioned. The window order (front to back) is unchanged.



Tiled
This option organizes the display into enough rows and columns (to a maximum of 6 rows and 4 columns) to display all Editor windows without overlap. The windows are placed from left to right and top to bottom. (For more than 24 windows, extra windows are arranged in the same rows and columns under the frontmost 24.) The Top Two Windows option is set off when this arrangement is selected.



Stacked
This is the only arrangement that changes the window order. All open Editor windows are placed in front of all other windows. They are then resized and positioned with the frontmost window on the bottom left, with each following window slightly above and to the right of the window in front of it. The window order between Editor windows is not changed by this arrangement. The Top Two Windows option is set off when this is selected.

Zoom ⌘/

Performs the same action as clicking in the zoom box of the frontmost window. A zoomed window resizes to fill the screen. For a window that already fills the screen, the window is reset to its prezoom size and location.

Swap

Switches the frontmost window with the one just behind it.

Send Front To Back
Bring Back To Front

Sends the frontmost window behind all other open windows. If you hold down the Option key while opening the menu, this command changes to **Bring Back To Front**. Selecting this command brings the rear window to the front.

Show Toolbar

Hides the toolbar of the frontmost window. If that toolbar is already hidden, selecting this command displays the toolbar. You can find more information about each type of window and their toolbars in the reference chapters for the windows.

Opening windows

These commands are used to open various windows in the Symantec Project Manager, such as the Build Errors and Class Browser windows.

Build Errors

Opens the Build Errors window to display the current list of build errors for the active project. If the Build Errors window is already open, selecting this command brings it to the front. For more information on the Build Errors window, see Chapter 23, “The Build Menu.”

Search Results

Opens the Search Results window to display the results of the last batched **Find** operation in the active project. If the Search Results window is already open, selecting this command brings it to the front. For more information on the Search Results window, see Chapter 21, “The Search Menu.”

Class Browser ⌘J
New Class Browser

When no Class Browser windows are open, a new Class Browser window displays the classes defined within the active project. When one or more Class Browser windows are open, selecting this command brings one to the front. If you hold down the Option key while opening the menu, this command changes to **New Class Browser**. Selecting this command opens an additional Class Browser window, regardless of the number that are currently open. For more information on Class Browser windows, see Chapter 22, “The Class Browser Window.”

Worksheet

Opens the Worksheet window for communicating with ToolServer and SourceServer. If the Worksheet window is already open, selecting this command brings it to the front. For more information on the Worksheet window, see the electronic supplemental information.

active project filename **⌘O**

Opens the Project window of the active project if it was closed, or brings the window to the front if it was open.

source files

Makes the window you select from the list of open Editor windows the frontmost window on the screen. This list includes all open Editor windows regardless of the project (if any) to which they belong. These windows are listed in alphabetical order.

Editor windows can be accessed using key combinations. A key combination, which is assigned to an Editor window when the window is first opened, takes the form Command-*number* (where *number* is a number in the range from 1 to 9). Note that if nine Editor windows are already open, any new Editor windows you open will not be assigned key combinations.

Key combinations, which do not change as long as the file remains open, are used to bring a window to the front. The effect is the same as selecting the file's name from the **Windows** menu.

other project filenames

Any open Project windows for projects other than the active one are listed at the bottom of the **Windows** menu. Selecting one of these projects brings its window to the front of the screen but does not make it the active project. To make a project active, you must choose **Switch Main Project** from the **Project** menu.

Symantec C++

TCL and VA Reference

Part Five

32 Visual Architect Pane Menu

33 Visual Architect Options Menu

34 Visual Architect Tools Menu

35 VA Symantec Project Manager Menu

TCL and VA: Basic Concepts ◆ 27

This chapter expands on the description of the THINK Class Library (TCL) and Visual Architect (VA) presented in Chapter 2, "Introducing Symantec C++ 8.0." The chapter briefly reviews the basic structures of the THINK Class Library, as well as how these structures interact and the part they play in an application. This chapter also more fully describes Visual Architect, a development tool for constructing the user interface for applications using the THINK Class Library.

This chapter discusses both TCL and VA in more detail than in Chapter 2 and contains many examples and illustrations. For more reference information on TCL, see Chapter 28, "Programming with the THINK Class Library." For more reference information on VA, see Chapters 29 to 35.

THINK Class Library

The THINK Class Library is a collection of C++ classes that help you implement standard Macintosh applications, I/OStreams-based applications, and even libraries for custom applications. These TCL classes handle basic Macintosh functions.

This section describes TCL structures, outlines their interactions, and explains how to create a THINK Class Library application.

TCL structures

The THINK Class Library is organized into three distinct, interacting structures: the class hierarchy, the visual hierarchy, and the chain of command. This section provides detail on these structures, particularly the chain of command by which actions are handled.

Class hierarchy

The class hierarchy is the set of classes that make up the THINK Class Library. The hierarchy is organized around the concept of inheritance. It contains a set of base classes from which other classes inherit their behavior (member functions) and attributes (data members).

Visual hierarchy

The visual hierarchy describes the organization of all visible entities, such as windows and buttons, that could be included in a given application. The hierarchy describes all the views that the THINK Class Library contains. A TCL view is an object of a class descended from the class CView. CView is an abstract class that is used for implementing objects with visual representations. Views respond to commands involving the mouse, and they can be links in the chain of command.

The visual hierarchy is based on the concept of enclosure. Everything you see on the screen belongs to—is enclosed by—another visual entity. This is in contrast to the class hierarchy, which is based on derived class relationships. In the visual hierarchy, because Activate and Update events can be processed in a modular fashion, each visual entity can process information locally, delegating the processing of enclosed views to their respective objects in turn.

Views and the visual hierarchy. Three kinds of views make up the visual hierarchy: the desktop, the windows, and the panes. These views are derived from the class CView. Abstract classes such as CView impart common behaviors to their derived classes. At the top of the hierarchy is the desktop, an object of class CDesktop. Every THINK Class Library application has a unique desktop object. The desktop encapsulates many of the properties and behavior of the Macintosh Window Manager, and it is the only view that does not have an enclosure. Instead, it encloses all the windows in an application. Every other view has an enclosure that specifies its place in the hierarchy.

Windows are objects of class CWindow. Each window encloses one or more panes—the most important kind of views because all drawing takes place in them. Panes are objects of class CPane and each may enclose other panes.

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 being concerned about where it is on the screen. Because many objects that you want to display are bigger than can fit in a pane, the THINK Class Library has a CPanorama class that you can use to implement scrollable panes. Such panes have logical dimensions that exceed the physical size of the enclosures in which they are displayed.

Figure 27-1 illustrates a typical visual hierarchy. The desktop encloses three different windows, each of which encloses at least one pane. Two of the panes, in turn, enclose additional panes.

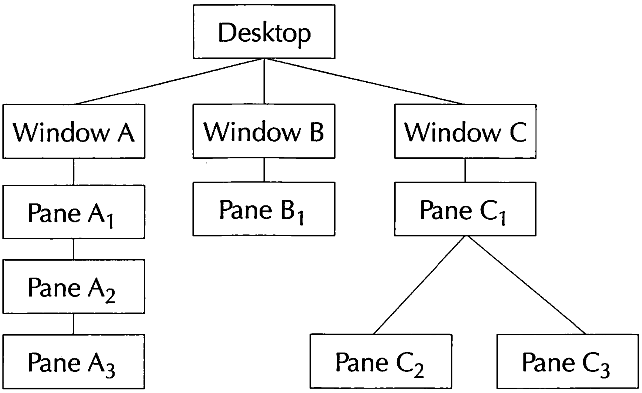


Figure 27-1 Typical visual hierarchy

Unlike the class hierarchy, the visual hierarchy is dynamic. The relationship of visible components changes as your program runs. For example, when you open a new document, you add another window to the desktop, and you add another set of panes to the new window.

Chain of command

The chain of command specifies those objects in an application that can handle certain commands and in what order. Because you are responsible for choosing this assignment of objects in an application, you also need to decide the level of abstraction at which to handle an action in the chain of command. A glossary of basic terminology for this section is provided in Table 27-1. Note that some of definitions there are defined by terms that appear later in the table.

Table 27-1 Basic terminology

Application	The application is the object at the end of the chain of command. It is the only bureaucrat without a supervisor.
Bartender	The bartender is the object that manages the menu bar, menus, and menu items.
Bureaucrat	A bureaucrat is any object in the THINK Class Library that is able to respond to menu commands or keystrokes. A bureaucrat can either respond to a command or a keystroke, or it can pass the responsibility for doing so to its supervisor.
Desktop	The desktop is the object that manages window layering. It encloses all of the windows in an application and is the root of the visual hierarchy.
Director	A director is a bureaucrat that manages a specific window. To interact with any element within a window, the window must be supervised by a director. The supervisor of a director must be the application or another director.
Document	A document is a director that manages a window and an associated file.
Gopher	The gopher is the first bureaucrat that has an opportunity to handle a command.
Pane	A pane is a view in which drawing takes place. Panes are enclosed by windows or other panes.

Table 27-1 Basic terminology *(Continued)*

Supervisor	A supervisor is a bureaucrat to which another bureaucrat passes commands or keystrokes that it cannot handle.
Switchboard	The switchboard is the object that converts Macintosh events to THINK Class Library member function calls that affect either the visual hierarchy or the chain of command. Every program creates one instance of the class `CSwitchboard.`
Window	A window is an object that encapsulates the behavior of a Macintosh window. Windows enclose panes.

Processing of commands. A bureaucrat can respond to either a command or a keystroke. When it 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. When a bureaucrat does not know how to handle a particular keystroke, it again defers to its supervisor by calling the supervisor's `DoKeyDown` command with that keystroke as its argument. The supervisor of a bureaucrat is itself a bureaucrat.

A bureaucrat is an instance of a class derived from CBureaucrat. When a bureaucrat is constructed, its supervisor is specified as an argument to the CBureaucrat constructor. 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. To handle commands specific to an application, you must override the `DoCommand` function in any derived class of CBureaucrat.

Every TCL 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.

Views and the chain of command. As previously stated, the chain of command specifies those objects that handle specific actions. But how do events in the visual hierarchy get processed? TCL has a special class to deal with such situations.

A bureaucrat that supervises a window is called a director. This kind of bureaucrat is important enough to merit a class, CDirector, derived from CBureaucrat. Directors handle communication between the visual hierarchy and the chain of command. Because a view is also a bureaucrat, the director bureaucrat can process events in the visual hierarchy.

For example, when a window gets an Activate event, it calls the `ActivateWind` function of its supervisor, which is a director. This director can then take some action as a result of the window becoming active.

Note

> The chain of command can contain more than one director—for example, ... → *director*→ *director*→ *application*. This type of chain can handle such cases as documents needing more than one window. The director highest in the chain is the "owner" of directors further down the chain. This nesting is determined when the object is created.

The gopher and the chain of command. The first bureaucrat that has the chance to handle a command is called the *gopher.* You are responsible for appointing the bureaucrats that act as gophers in your application. 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 example, a dialog text pane must become the gopher in order to receive the commands generated by keystrokes and by menu choices. Objects further up in the visual hierarchy are seldom gophers. 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 when no documents or windows are open, or when the application is in the background.

If the gopher cannot handle a command, it passes the command on to its supervisor. If its supervisor can handle the command, it returns; otherwise it defers to its own supervisor. Eventually, some bureaucrat takes action—or each one in the chain defers to its supervisor until ultimately the command reaches the application object. If the application object doesn't handle the command, no object does, and the command is ignored.

In summary, the chain of command is the list of bureaucrats starting with the gopher and ending with the application object.

A sample interaction between TCL structures

To illustrate the interaction between the chain of command and visual hierarchy, this section presents two sample scenarios, each based on a typical application—a snapshot of the initial setup of the chain of command is shown in Figure 27-2. A typical flow of control between the chain of command and the visual hierarchy based on this setup is shown in Figure 27-3.

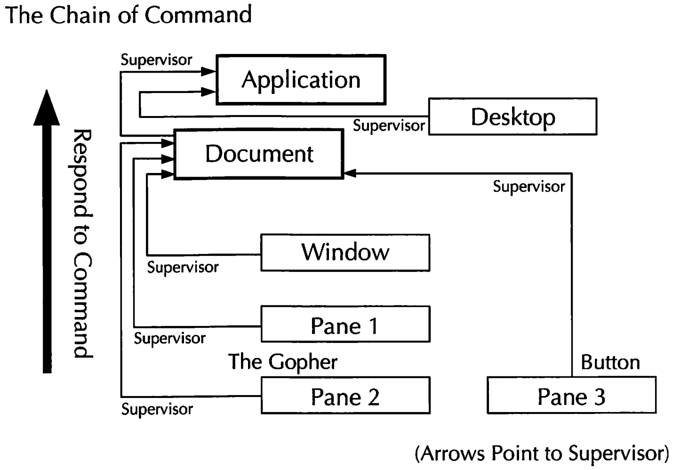


Figure 27-2 Initial setup of a chain of command

In the first scenario, Pane 3 is a button within Pane 1. What happens when it is pressed? In the second scenario, a command is chosen from the menu bar with Pane 2 as the gopher. Pane 2's supervisor is a document, which also supervises the other two panes (Panes 1 and 3) and the window. A document is a director that manages the communication between windows and files.

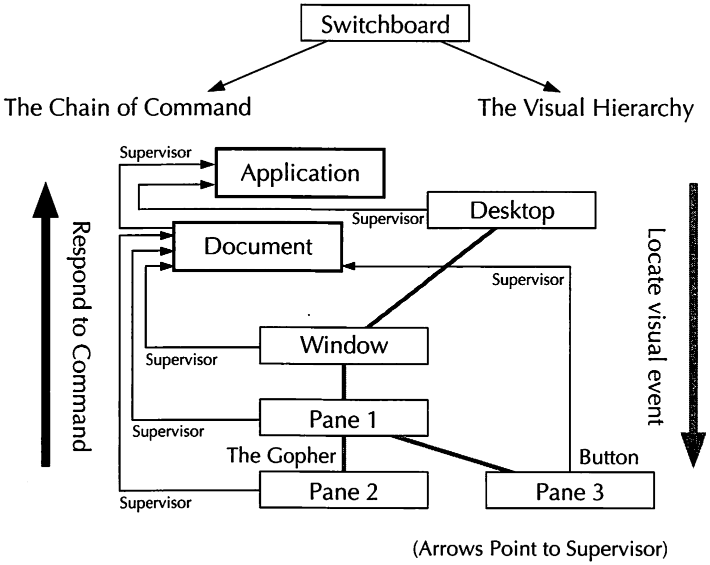


Figure 27-3 Gopher, chain of command (black lines), visual hierarchy (gray lines), and typical flow of control

Scenario 1

A button, Pane 3, is pressed. In this case, because the event is a button press, the switchboard directs this event to the visual hierarchy. Visual events work their way down the hierarchy, starting from the desktop view, following that view's pointer relationships until the switchboard determines in which view the mouse was positioned. In this case, it determines that the mouse click occurred in Pane 3 and calls that pane's `DoClick` function. Pane 3 responds to the mouse click by calling its supervisor's (the document's) `DoCommand` function. Note that neither Pane 2 nor Pane 1 ever responds to this command.

If the document cannot handle the command, it passes the command to the application. Thus, the visual hierarchy is used to locate visual events, such as mouse clicks, and the chain of command is used to respond to the commands associated with these events.

Scenario 2
Pane 2 is the gopher when a command is chosen from the menu bar. In this case, because the event is a command, the switchboard directs the request to the chain of command. Menu commands go through the gopher and up the chain of command. Thus, if Pane 2 cannot handle the command, it passes the command to its supervisor, the document.

In Figure 27-3, the enclosure relationships of the visual hierarchy are drawn in gray; the supervisory relationships are drawn in black. Notice that the chain of command and the visual hierarchy overlap—the panes and the window (located in the visual hierarchy) have the document (located in the chain of command) as their supervisor. Nevertheless, the visual hierarchy and chain of command are distinct.

Creating a THINK Class Library application

To create an application that uses the THINK Class Library, you derive classes from existing classes that the library provides. At a minimum, you need to derive classes from CApplication, CDocument, and CPane. The application classes you choose determine the overall structure of the application. The document class implements file handling for the application, and the pane classes respond to user interaction and implement how the information in files is shown in the document windows.

In addition, you need to define both a menu structure to contain the commands that the application implements and the linkage between menu commands and actions (that is, the action is performed in response to a specific command). Finally, virtually all Macintosh applications make use of resources, and a TCL-based application is no exception. In fact, a TCL application requires standard resources to function.

These resources are found in the file `TCL Resources`. You must add these resources to your project, either by copying them into the `.rsrc` file that contains additional application-specific resources of your own, or by creating a project using one of the VA Application project models. For a more detailed discussion concerning the creation of a TCL application, see the section "Writing an Application with the TCL," in Chapter 28, "Programming with the THINK Class Library."

Visual Architect

Visual Architect is a powerful development tool with which you can rapidly create applications that are constructed from the THINK Class Library. Visual Architect streamlines the process of creating, editing, and connecting the classes, menus, commands, and supporting resources needed by an application.

The role of Visual Architect

Visual Architect automatically generates C++ source code files and lets you maintain relationships between the TCL resources that make up an application. The source code files contain definitions and declarations for the classes created for an application. The resources contain information needed to initialize window and pane classes according to your specification, as well as the menus and their associated commands. When you use the Symantec Project Manager to start a new Visual Architect project, it automatically creates a file called `Visual Architect.rsrc`. Visual Architect works with this resource file.

The source code files are standard C++ `.cp` and `.h` files, and as such can be opened and edited using the Symantec Project Manager. The resources typically are edited only with Visual Architect; however, you can add resources to existing TCL resources using a standard resource editor such as ResEdit.

Although it can be used on its own, Visual Architect is designed to work with Symantec Project Manager. Because Visual Architect is tightly integrated with the TCL and the Symantec Project Manager, you can develop an application's user interface at the same time as you develop its underlying code. The Symantec Project Manager compiles files generated by Visual Architect even when Visual Architect is open.

Visual Architect uses special files called macros to generate source code. A macro file is an ordinary text file that contains C++ source 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 THINK Class Library application. Because macro files are ordinary text files, you can, if necessary, modify them to suit your programming needs or extend them with new capabilities.

Note

A Visual Architect project also contains another resource file, `Project Resources.rsrc`. This file initially contains resources needed by the THINK Class Library, but not used by Visual Architect. You can add more resources to either file using Symantec Rez or ResEdit.

Starting Visual Architect

You begin a Visual Architect application from the Symantec Project Manager by choosing a VA Application project model. (See "Opening projects, Editor windows, and files," in Chapter 16, "The File Menu.") If you have created your application with one of these project models, a file named `Visual Architect.rsrc` is listed in the Project window.

To launch Visual Architect from the Project window, double-click the `Visual Architect.rsrc` icon. The View List window opens, displaying a list of the views defined in the `Visual Architect.rsrc` file. The window initially contains a view called Main. This view is the graphical equivalent of the familiar "Hello World." You can generate all files and run the project without changing the `Visual Architect.rsrc` file.

Creating and modifying classes

One of Visual Architect's most powerful features is how it facilitates the implementation of views and the classes constructed from these views with the help of an interactive graphical environment. Visual Architect automatically derives classes from views using the THINK Class Library, as well as defining the classes, data members, and member functions. Visual Architect also generates source code files that contain these data and function members; thus, you do not need to determine the class definitions that are required for a specific application. Visual Architect adds these source files to a Symantec C++ project.

Typically, the classes you construct fall into one of two categories:

- VA views (director or document classes)
- Panes

You can add, delete, and modify derived classes in either of the categories. If a derived class is currently in use, any change to that class necessitates a corresponding change in all VA views containing objects of that class. Visual Architect makes these changes automatically. Visual Architect allows you to change the attributes of a VA view through dialog boxes, instead of the more time-consuming method of writing the code by hand.

Because the implementation of a specific VA view is based on a particular THINK Class Library class, the classes you create in Visual Architect must be derived from the set of TCL base classes that Visual Architect recognizes (or from a class derived from one of these).

Working with Visual Architect views

With Visual Architect, you have predefined views available for implementing common graphical representations, such as document windows (which are used to view the contents of a file), dialog boxes, floating windows, splash screens, subviews, floating tool palettes, and basic windows.

Note

A view in the THINK Class Library is an object whose class is derived from CView. In contrast, a "view" in Visual Architect refers to the windows, dialogs, tear-off menus, floating windows, and modular "subviews" that contribute to the interface you are designing. In other words, all Visual Architect views are THINK Class Library views, but the converse is not true.

All VA views, except for subviews, consist of at least two objects: a window and a director. (In the case of subviews, only a panorama is involved.) 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 generated by or for that 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 carrying out the change.

When you add a VA view to a project, Visual Architect generates the director class necessary for proper functioning of the view, as well as the associated resources. The default name given to the director class is C*viewname*, where *viewname* is the name you give the view.

Note

You can change the class name of a view's director class in the **Classes** dialog box, but you cannot delete the class or change the base class.

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, serving as the focus of attention for a user.

Adding graphical elements

Visual Architect's Tool palette (Figure 27-4) contains tools for quickly creating the different interface elements displayed in a view. Each interface element is a pane object. You can add pane 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 and supporting resources.

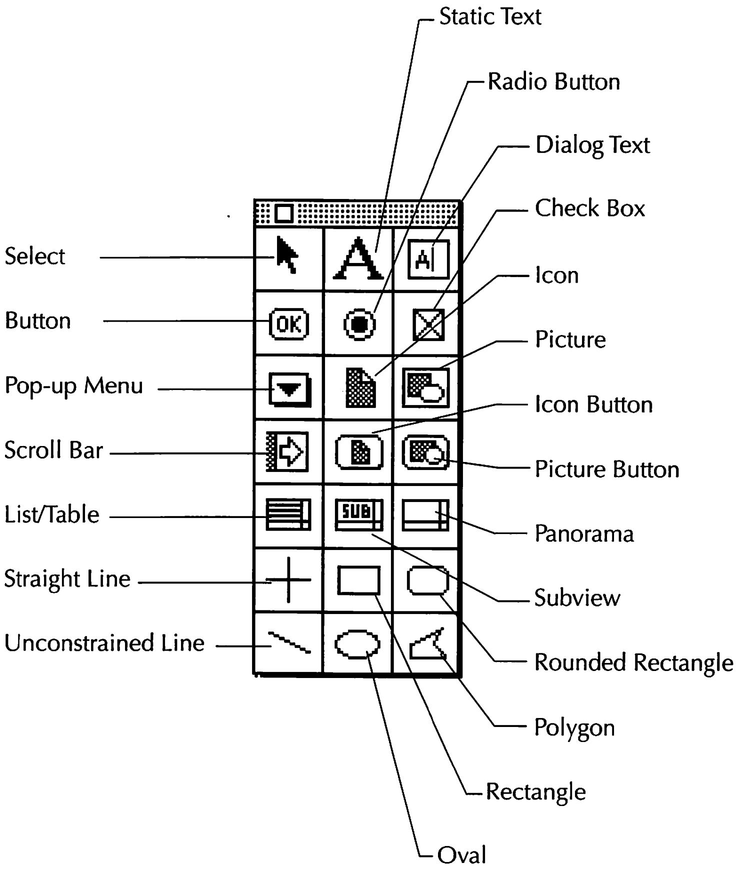


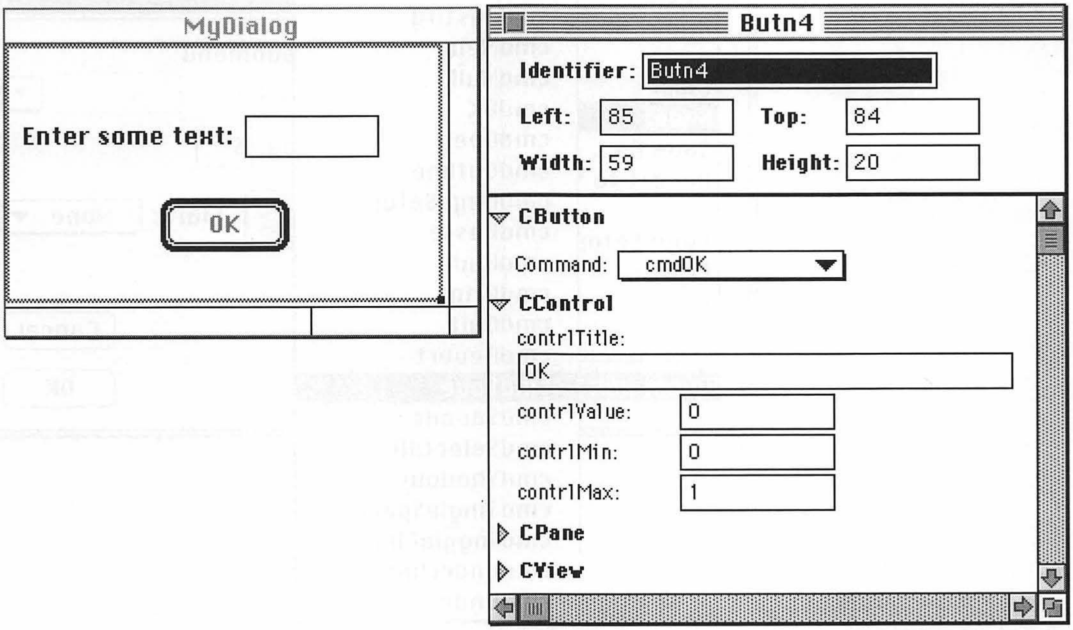
Figure 27-4 Tool palette

For a detailed discussion on each of the tools, see Chapter 34, "Visual Architect Tools Menu."

Changing data member values

You can easily change data member values in each pane object you create with the Tool palette. For example, if you are constructing a dialog box and want to initialize its OK button, you access the relevant data members from the button's CButton, CControl, CPane, and CView classes. Clicking a right-pointing triangle to the left of a class name opens a subarea that contains the editable subset of the data members for that class.

In Figure 27-5, 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. Changes made here are reflected immediately in the target pane.



MyDialog

Enter some text:

Butn4

Identifier: Butn4

Left: 85 Top: 84

Width: 59 Height: 20

CButton

Command: cmdOK

CControl

contrlTitle: OK

contrlValue: 0

contrlMin: 0

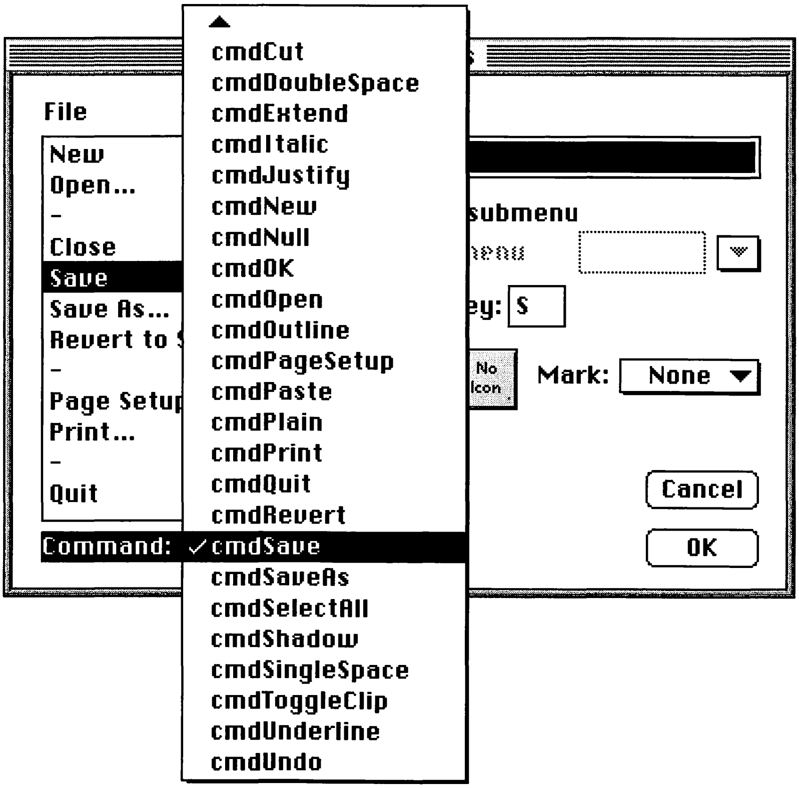
contrlMax: 1

CPane

CView

Figure 27-5 Accessing the OK button's class hierarchy

Defining and associating commands with views and menus
Visual Architect lets you assemble menus and link menu commands with actions. The interface for constructing menus is similar to that of other resource editors such as ResEdit. Visual Architect, however, shields you from many of the details required to construct menus. Linking a menu command with an action, for example, is as simple as selecting the action from a list in the **Menu Items** dialog box (see Figure 27-6).


Command: ✓cmdSave

ey: S

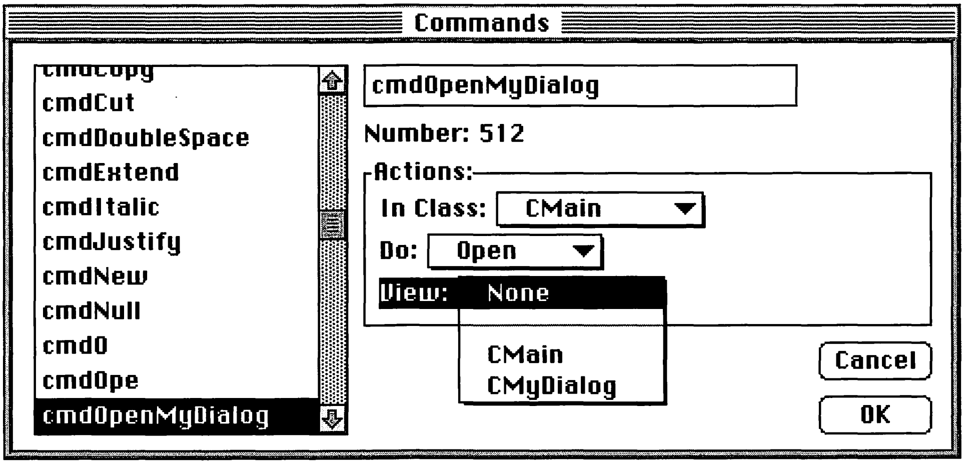
Mark: None

Figure 27-6 Linking commands via the Menu Items dialog box

To process various user actions—for example, mouse clicks or keyboard events—the THINK Class Library predefines many commonly occurring actions such as closing a window, saving a file, quitting an application, and changing text attributes. With these actions predefined, you can focus on creating the commands that are unique to your application.

When you want to implement a predefined action or a new action using Visual Architect, you need to indicate the class or classes that will handle the action. If the action of a command is to open a view, you specify that view and Visual Architect generates the necessary code. Otherwise, you indicate those classes that need to respond to the action and Visual Architect then generates an empty member function. Later, you can insert code in the member function that handles the action.

For example, Figure 27-7 illustrates how you would add a handler for the command `cmdOpenMyDialog` to class CMain. When `cmdOpenMyDialog` is handled by CMain, CMain opens the view defined by the class CMyDialog. Notice some of the predefined commands available in the scrolling list. Visual Architect also lets you associate commands with other elements that send them, such as buttons and ArrayPanes.



cmdOpenMyDialog

Number: 512

Actions:

In Class: CMain

Do: Open

View: None

Figure 27-7 Adding a command using the Commands dialog box

Note that the chain of command affects the use of Visual Architect. For example, if you specify that a button should send a particular command and that command cannot be handled by any object in the button's chain of command, then the button command does not get executed. No object can receive its message. Thus, knowing what objects are in an object's chain of command can help prevent you from introducing bugs into your code.

Trying out an application interface

With Visual Architect, you can see how the interface works before you generate source code, compile it, and run the application. When you try out a view, it is displayed exactly as it would be in a running application. Further, you can interact with the view—for example, by scrolling, resizing, and repositioning it. All the view's elements, such as pop-up and custom buttons, are active as well.

Trying out views lets you glimpse the final product of your work quickly and conveniently. As a result, the design process can proceed more rapidly.

Modifying the code generated by Visual Architect

Programming is never accomplished in one step. Typically, you design some user interface elements in Visual Architect, hand code in the Symantec Project Manager, compile, run, and inspect your project. At that point, you would return to Visual Architect to make changes and start the cycle again.

Visual Architect does not force you to live with the code it generates "as is." Much of the code it generates is well-commented C++ skeleton code. Visual Architect will not, in subsequent code-generation steps, overwrite changes you make to your code by hand.

Visual Architect facilitates and protects hand coding with an architecture based on split-level classes. Most classes defined in Visual Architect are implemented as two classes: a lower-level class, reserved for Visual Architect, and an upper-level class, reserved for custom programming. The first time Visual Architect generates source code for a graphical element, it generates code for both classes in separate files.

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 is shown here. You should not modify this code.

The upper-level class is derived from the lower-level class. To customize the skeleton code, you add member functions, additional data members, and so forth to this class. Member functions that you manually add to the upper-level class often override or expand on the corresponding lower-level class member functions. Visual Architect writes to the upper-level class 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 upper-level class.

Programming with the THINK Class Library 28

This chapter explores aspects of the THINK Class Library (TCL) that you encounter in the course of building an application. No topic is covered exhaustively here; the class descriptions in the online documentation provide the definitive reference. Nevertheless, the discussion in this chapter is sufficiently detailed to give you a sound understanding of how the library works and of how you can use it to build a robust application. The chapter assumes that you have read Chapter 27, "TCL and VA: Basic Concepts," and that you have worked through the tutorials.

Introduction

This chapter discusses the essential features of the THINK Class Library and covers some of the general topics in application building. After you have read this chapter, you should try running and modifying the demonstration applications.

Naming Conventions

The THINK Class Library adheres to the following naming conventions:

Table 28-1 Naming conventions

Name	Description
C*Name*	Name of a class
a*Name*	Formal parameter
c*Name*	Static data member
f*Name*	Flag, usually a Boolean data member
g*Name*	Global variable

Table 28-1 Naming conventions *(Continued)*

Name	Description
k*Name*	Constant, defined with #define, enum, or const
its*Name*	Data member
the*Name*	Variable, usually a local variable or data member; sometimes used for formal parameters
mac*Name*	Macintosh data structure, used either as a data member or as a local variable

You should give names to classes names that begin with letters other than uppercase “C”. Adhering to this practice avoids 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. THINK Class Library classes still contain explicit initialization and destruction member functions to maintain backward compatibility with THINK C with Object Extensions.

The old initialization function for a class C*Name* is denoted by I*Name*; the old destruction member function is always named Dispose. These member functions are now anachronisms, superseded by genuine C++ constructors and destructors. Their continued presence guarantees that existing code written for earlier versions of the THINK Class Library does not require extensive rewriting.

Writing an Application with the TCL

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 the application. The document class implements file handling for the application, and the pane classes implement the way in which information in files is displayed in document windows.

Note

Many THINK Class Library pane classes already supply the functionality needed for an application. Typically, you derive new pane classes to customize or add to the behavior of existing classes. 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 a project. This resource file is intended to contain the standard THINK Class Library resources as well as any resources you add to the project. The standard resources are located 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.

Note

For more information about resource files and the THINK Class Library, see the section "THINK Class Library Resources," later in this chapter.

Creating the application class

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

```
SetUpFileParameters
CreateDocument
OpenDocument
DoCommand
```

Your application class constructor should initialize any new data members declared by your derived class. 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 the files that will be visible in the standard **File Open** dialog box when the user chooses **Open** from the **File** menu.

CreateDocument creates a new, untitled document when you choose **New** from the **File** menu . You derive the class of the document it creates from CDocument. After creating the document, your CreateDocument function calls the document's NewFile function.

OpenDocument is similar to CreateDocument. Instead of calling the newly created document's NewFile function, your OpenDocument calls its OpenFile function. The OpenDocument function takes one parameter, an SFReply record, which contains information about the file the user has chosen to open.

DoCommand handles all application-specific commands. Typically, most commands specific to your program are best handled at the document level. Some commands, such as **New**, **Open**, and **Quit**, are already handled by the default application class, CApplication.

Creating the document class

Applications draw and display data in documents. All documents have associated windows. Most documents also have an associated file. Neither the window nor the file is created automatically.

The document class must derive from CDocument. It must define a constructor and override the following member functions:

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

If your document class defines new data members, its constructor must 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 delete the data members `itsWindow` or `itsFile`. The virtual destructor `~CDocument` does that for you.

The `DoCommand` functions of your document classes do most of the work in an 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 to its supervisor. Your document class should handle all the commands it knows about, and call `CDocument's DoCommand` when it cannot handle the commands. CDocument's DoCommand function will process all the commands it understands and will pass any commands that it does not understand to the application object's `DoCommand` function.

Your document classes's `NewFile` function is called when the user chooses **New** from the **File** menu. This function creates a window and attaches its panes. The `NewFile` function does not 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. Calling the `OpenFile` function confirms 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 one 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, the document's `DoSave` function is called. The `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, the 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 must override this function to write its data to a file.

If the application supports the **Revert** command, you should implement it in the `DoRevert` function. Your implementation might do something like closing a file without saving, then opening it 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 have created your windows and opened files, you need to display them somewhere. In the TCL, you do not 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 a 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 to which its window belongs.

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

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

The `Draw` function always draws its contents in a pane. You can assume that the port, clip region, and coordinate systems are set up correctly when `Draw` is called. See the section "Drawing in a pane" later in this chapter to learn about the drawing environment.

When you click a pane, its DoClick function is called. DoClick can either perform actions (such as drawing in a pane) or initiate processes (such as dragging an object). Some panes, such as the dialog text pane implemented by CEditText, have built-in DoClick functions that may already implement the behavior needed by your pane class.

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

Working with Panes

In the TCL, all 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 receive mouse clicks and events that affect its display such as Activate and Deactivate events.

Every pane has an enclosure that completely surrounds the pane. A pane also has a supervisor, which is 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. Typically, the supervisor is the director that owns the window or the pane.

Every pane has its own drawing environment. The rectangle that defines 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 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 objects as controls, borders, pictures, and size boxes.

Every view has a single enclosure. For example, in the window in Figure 28-1, there are eight views and seven panes (the window is not a pane):

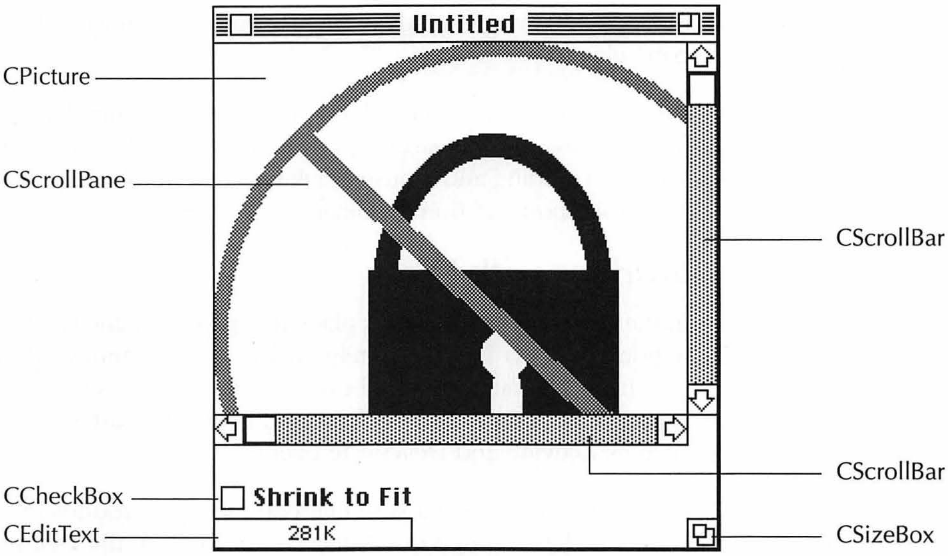


Figure 28-1 Panes in a window

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
- Window
- Frame
- QuickDraw

The desktop, some of the internal member functions, and some of the 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 also (0, 0), and each unit is a pixel. You will need the 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 when there are two 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. The coordinate system does not change if the pane moves within its enclosure; the upper-left corner is still (0, 0). The only time the origin point changes is when a panorama, which is a special kind of pane, is scrollable. For each pane, you can choose at run-time a coordinate system, long (32-bit) or short (16-bit), by calling the inherited CView function `UseLongCoordinates(fUseLong)`.

All drawing and mouse tracking is done in QuickDraw coordinates. QuickDraw is the coordinate system that the Macintosh Toolbox uses for its drawing operations. QuickDraw coordinates are valid only following a call to `Prepare`. The relationship between QuickDraw coordinates and any of the other coordinate systems changes, depending 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, which limit a pane using short coordinates to the rectangle (–32768, –32768, 32767, 32767). Long coordinates layer a 32-bit coordinate system on top of QuickDraw 16-bit coordinates. The long coordinate system lets you use a much larger coordinate area for a pane. Because 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, typically you override its `Draw` function. The THINK Class Library calls the 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 a 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 the pane uses short coordinates, the coordinate system is set up correctly. If your pane uses long coordinates, you need to transform the frame coordinates to QuickDraw coordinates before drawing. You can also use the 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 have worked with earlier versions of the TCL, this uniform use of LongRect and LongPt is the biggest and most noticeable change because it necessitates that you make 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, the frame coordinates and the QuickDraw coordinates are identical, and the values stored in a LongRect or in a LongPt will already be in QuickDraw coordinates. To use LongRect and LongPt 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 online *THINK Reference.*

Properties of panes

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

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

Table 28-2 Horizontal sizing characteristics

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 are anchored to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane scrolls with it.
`sizELASTIC`	The width of the pane grows or shrinks by the same amount as the width of the enclosing pane.

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

Table 28-3 Vertical sizing characteristics

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

Following 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 28-2, the box with the thick outline represents the window. It contains a main pane, which 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 present, regardless of how the window changes. Its dimensions do not change automatically.

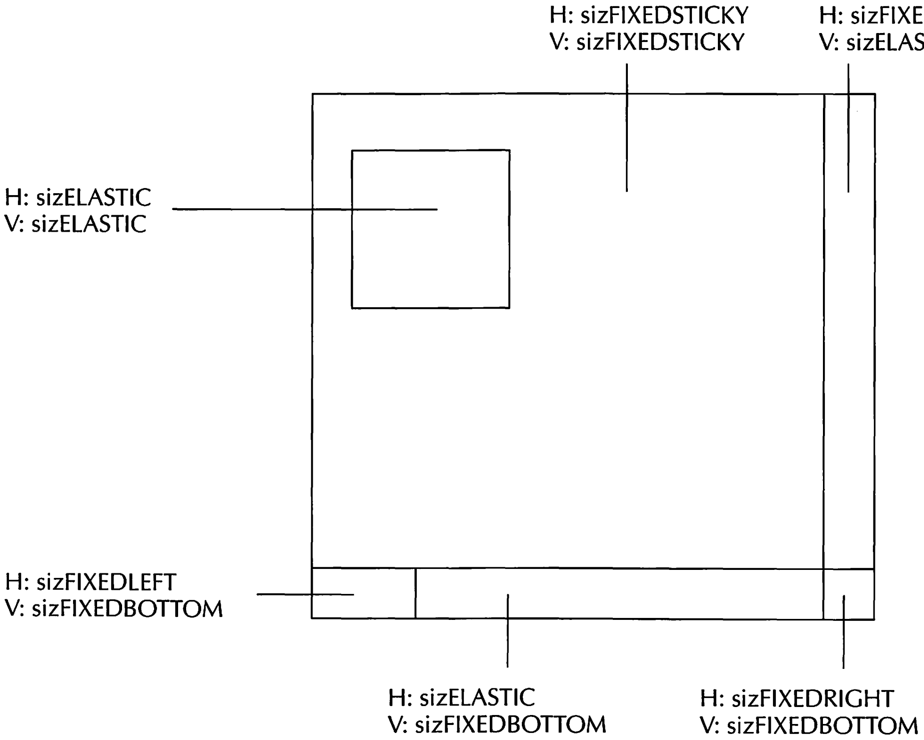


Figure 28-2 Horizontal and vertical sizing

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 varying portions of a large graphic in a pane.

For example, imagine that the only part of the sailfish in Figure 28-3 that you can see is the portion inside the frame. Currently, the frame displays the sailfish's tail; but you could also move, or scroll, the frame to include other parts, such as the head.

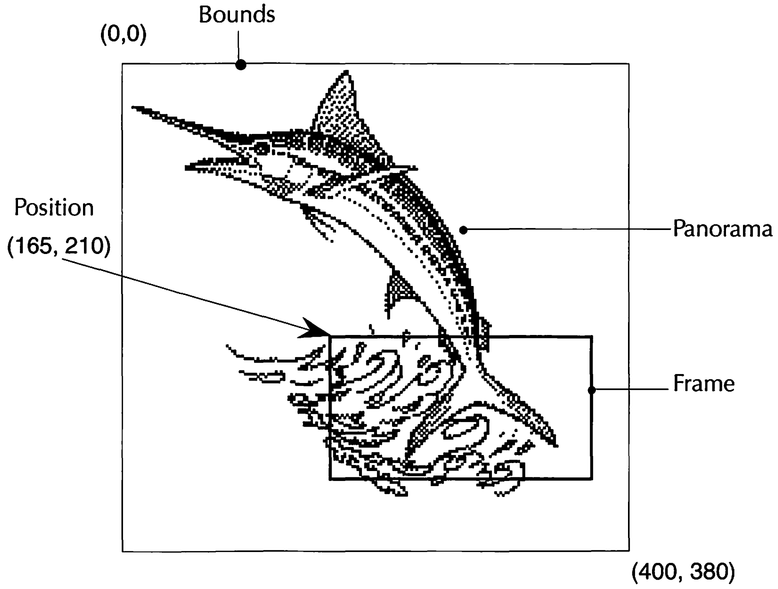


Figure 28-3 Graphic panorama and its scales

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 as you scroll. Usually, the frame moves one pixel at a time. In some applications, though, scrolling by a different measure 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 one scale for the horizontal units and another scale for vertical units. In a graphics application, each unit might be one 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. Drawing uses 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, the 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 terms of panorama units, and drawing in terms of frame coordinates. As you scroll, the origin of the frame changes. The following examples help clarify these concepts.

In Figure 28-3, both the horizontal and vertical scales are set to one 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).

Because 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 28-4, 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). Thus, if you wanted to draw a line to strike out the word "And," you would draw it from (0, 42) to (18, 42).

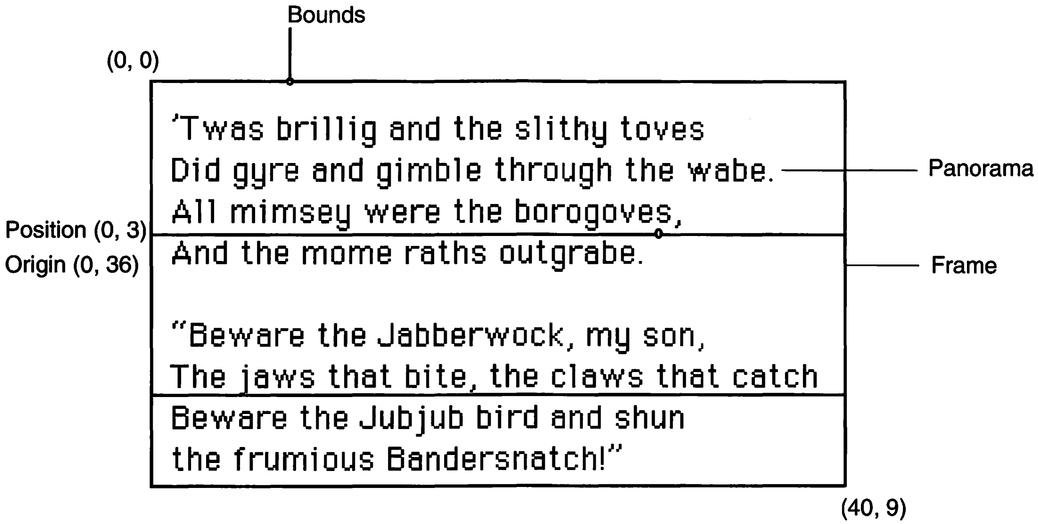

Bounds
(0, 0)
'Twas brillig and the slithy toves
Did gyre and gimble through the wabe.
All mimsey were the borogoves,
And the mome raths outgrabe.
"Beware the Jabberwock, my son,
The jaws that bite, the claws that catch
Beware the Jubjub bird and shun
the frumious Bandersnatch!"
Panorama
Frame
Position (0, 3)
Origin (0, 36)
(40, 9)

Figure 28-4 Text panorama and its scales

Scroll panes

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

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.

The scroll bars and the panorama communicate through the scroll pane. When you click a scroll bar, it informs the scroll pane, which tells the panorama how many panorama units to scroll, depending on which part of the scroll bar you clicked.

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`. An example would be the `AdjustCursor` function in CEditText.

Sometimes, you might want to use different cursors in the same pane. The `AdjustCursor` function lets you do this, but at the cost of a little more work. See the description of the CView class in the online *THINK Reference.*

Working with Menus

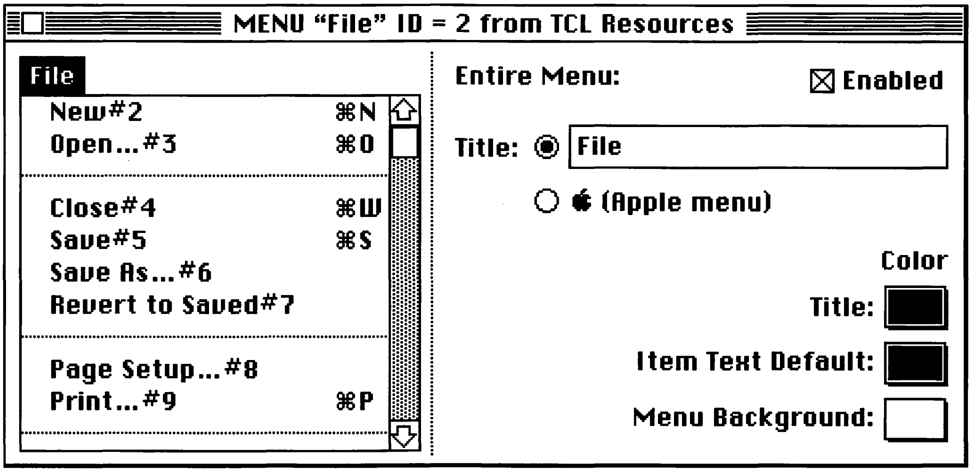
The THINK Class Library lets you identify menu items by assigning them unique command numbers. Command numbers are positive long integers in the range 0 to 2,147,483,647. Those in the range 1 to 1023 are reserved for the THINK Class Library. Command number 0 is reserved for `cmdNull`, the null command. 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 the class CBartender in the online *THINK Reference.*

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 is therefore 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 28-5 shows the **File** menu as viewed in `ResEdit`.



MENU “File” ID = 2 from TCL Resources

File

New#2 ⌘N
Open...#3 ⌘O

Close#4 ⌘W
Save#5 ⌘S
Save As...#6
Revert to Saved#7

Page Setup...#8
Print...#9 ⌘P

Entire Menu: ☒ Enabled

Title: ● File

○ (Apple menu)

Color

Title: ■

Item Text Default: ■

Menu Background: □

Figure 28-5 File menu viewed in `ResEdit`

Note

If you do not 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:

Table 28-4 Menu IDs

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

After you create all the menus that an application needs, create a resource of type `'MBAR'` with an ID of 1 to contain 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, lack command numbers associated with the items. 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 have to determine 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.

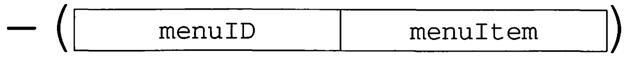


Figure 28-6 FindCmdNumber building command numbers

You can add menu items to existing menus. Menu items must be added at the end of an existing menu; otherwise, the bartender does not work properly.

Enabling 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. All items in the menu usually 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 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 is displayed 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, then enabling them again makes little sense.

Table 28-5 Menu dimming options

Dim Option	Meaning
`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.

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

Table 28-6 Menu unchecking options

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** or **Style** menus, whose items are mutually exclusive.
`FALSE`	Don't uncheck any menu items at menu selection. This is the default, because most menu items never need to be checked.

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 CApplication in the online *THINK Reference.*

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 example, your `MemoryShortage` function could delete a buffer it no longer needs. If the `MemoryShortage` function cannot 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.

Table 28-7 Reserve memory

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 and that cannot fail—even though it might require a large amount of memory. For example, saving a file is a critical operation. You can use the routine `SetCriticalOperation` to set the `inCriticalOperation` flag.

If `GrowMemory` still cannot release enough memory, and the `canFail` flag is `TRUE`, the function (or GrowMemory) 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 you have insufficient memory, even after using the rainy day fund, the application usually crashes.

You can use the routine SetAllocation to set and reset the canFail flag. In general, an 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.

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

The Undo function calls the document's DoCommand function to change the font back to the saved, previous one. Because the command goes through the regular command chain, the DoCommand function creates 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 implements whatever action you want taken when the mouse is down. The `EndTracking` function implements whatever action you want taken when the mouse is released.

For example, if you are 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, meanwhile, would erase the rectangle from its old location and redraw it in the new location.

To make the 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 is an underscore at the beginning and end of _TCL_DEBUG_. You can set the value of _TCL_DEBUG_ in the TCL precompiled header source file TCL #include.cpp.

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 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.

Table 28-8 Alert and dialog resources

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.
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."

Table 28-8 Alert and dialog resources *(Continued)*

ALRT/DITL ID	Used for
`253`	Assertion failed. Used by assertion in exception handling.
`300`	Macintosh OS error alert.

Controls

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

Table 28-9 Control resources

CNTL	Used for
`300`	Scroll bar

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'` resources. The error-handling class CError uses `'Estr'` resources to display messages. You should create an `'Estr' resource` for every error an application reports to the user.

Table 28-10 Error string resources

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

Menus

The THINK Class Library reserves the menu IDs shown in Table 28-11 for the standard menus. The **File** and **Edit** menus contain all the standard items. You can remove those that do not 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.

Table 28-11 Menu resources

Menu	Used for
1	Apple
2	File
3	Edit
10	Font
11	Size

Note

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

Table 28-12 Menu bar resource

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

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'` resources, you can use the routine `DrawSICN` to draw it in a pane. See the online *THINK Reference.*

Table 28-13 Small icon resource

SICN	Used for
200	Grow box

Strings and string lists

The THINK Class Library uses these strings for various prompts and messages. You can modify them to suit your application. The string resources are listed in Table 28-14.

Table 28-14 String resources

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`"

The string list resources listed are in Table 28-15.

Table 28-15 String list resources

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.

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.

Table 28-16 Window resource

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

Segmentation and the THINK Class Library

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

The following 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, 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, making changes in the source code may make it difficult to take advantage of future releases of the THINK Class Library from Symantec. 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 TCL, you may want to create general-purpose, reusable classes derived from TCL 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 TCL. You may not, however, distribute modified sources of TCL classes.

Visual Architect File Menu 29

This chapter is the reference for the **File** menu of the Visual Architect. It describes all commands on the menu as well as the preferences in the **Preferences** dialog box.

Commands in the File Menu

You use the **File** menu to manipulate Visual Architect resource files. This menu contains commands for printing open windows, setting preferences, and quitting Visual Architect.

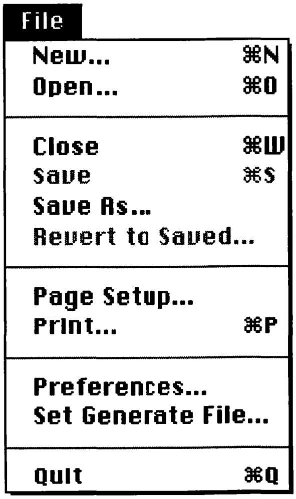


Figure 29-1 File menu

The **File** menu commands are used to perform these functions:

- Access files
- Close and save files
- Print
- Set preferences

This chapter discusses the **File** menu items by function. The **Quit** command is covered under the section on closing and saving files.

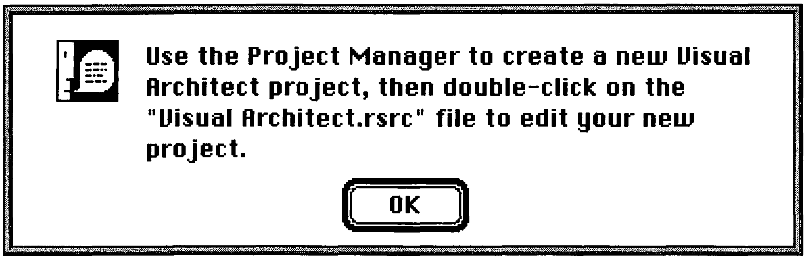
Accessing files

Two commands cover creating new files or opening existing ones.

New ⌘N

Creates a new Visual Architect resource file. This command is only functional when the Update Project on Generate preference in the **Preferences** dialog box is set off.

If that preference is set on, the **Unable to Create New File** dialog box opens when you choose **New** (Figure 29-2).



Use the Project Manager to create a new Visual Architect project, then double-click on the "Visual Architect.rsrc" file to edit your new project.

Figure 29-2 Unable to Create New File dialog box

Open ⌘O

Opens an existing Visual Architect resource file. Choosing this command opens a standard **File Open** dialog box, in which you can select the file.

You typically open the Visual Architect resource file when Visual Architect is launched, by double-clicking the resource file's icon in the Project List window in the Symantec Project Manager. However, you may want to open a Visual Architect resource file manually, for example, when you are copying resources from one Visual Architect resource file to another.

Multiple resource files can be open simultaneously in Visual Architect. The resource file that owns the frontmost window is the active resource file. This file is the one affected by commands in the Visual Architect's **Symantec Project Manager** menu.

Note

> Visual Architect lets you open any resource file (those with file extension `.rsrc`). However, Visual Architect is designed to work with resource files created in Visual Architect or in the Symantec Project Manager using a Visual Architect project model.

Closing and saving files

You use this group of commands on the **File** menu to close a file, quit Visual Architect, or save a file. You can also choose to revert to the last-saved version of a file.

Close ⌘W

Closes the active window. Choosing this command has the same effect as clicking on the active window's close box. If you try to close a resource file and the file has been modified since it was last saved, a dialog box is displayed, prompting you to save the changes, discard them, or cancel the **Close** command.

Save ⌘S

Saves the active resource file.

Save As

Saves the active resource file under another name. You may want, for example, to experiment with the new resource file while still using the original file to build applications.

Note

> By convention, the Visual Architect resource file is named `Visual Architect.rsrc`. If you use **Save As** to create a resource file for experimentation, you must save it as `Visual Architect.rsrc` to include it in a Symantec Project Manager project. At that time, the original resource file is overwritten.

Revert to Saved

Restores the last-saved version of the active resource file and discards any edits made since the last save. You are prompted to confirm the operation.

Quit ⌘Q

Quits Visual Architect. If **Confirm Save** is set on the **Preferences** dialog box, you are prompted to save any changes made to open resource files. Otherwise, changes are saved automatically.

Printing

These two commands cover page setup and printing.

Page Setup

Displays the standard **Page Setup** dialog box that lets you specify the size of the paper used for printing, whether the file should be printed landscape or portrait mode on the page, and other page setup options. See your Macintosh owner's manual for details.

Print ⌘P

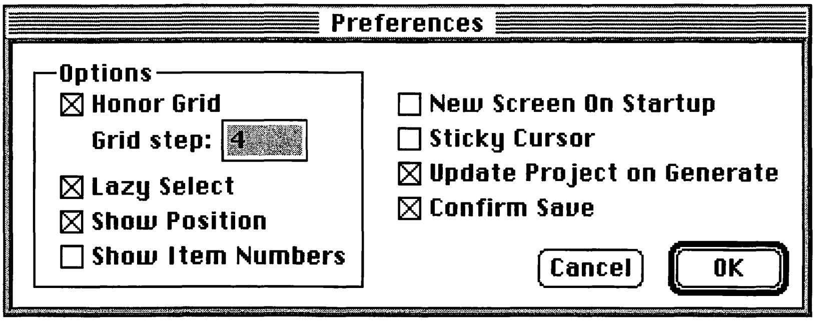
Displays the standard **Print** dialog box that lets you print the active window. You can set the number of copies to print and other print options. See your Macintosh owner's manual for details.

Setting preferences

You set preferences through the **Preferences** dialog box. You can also choose to control code generation for the active resource file. The preferences settings are saved automatically when you quit Visual Architect.

Preferences

Displays the **Preferences** dialog box.



Preferences

Options

☒ Honor Grid

Grid step: 4

☒ Lazy Select

☒ Show Position

□ Show Item Numbers

□ New Screen On Startup

□ Sticky Cursor

☒ Update Project on Generate

☒ Confirm Save

Figure 29-3 Preferences dialog box

Options area

The preferences in the Options area of the **Preferences** dialog box allow you to set the default values of the corresponding options in Visual Architect's **Options** menu. See Chapter 33, "Visual Architect Options Menu," for details.

New Screen on Startup

When this preference is set on and Update Project on Generate is set off, Visual Architect opens a new resource file on startup.

Sticky Cursor
When this preference is set on, any tool selected from the Tool palette remains "stuck" to the cursor until another tool is selected. This "sticky cursor" function is useful when you are creating multiple panes of the same type. With the preference set off, you can achieve the same effect by double-clicking tool items. (You must first drag the **Tools** menu off the menu bar to be able to do this.) If instead you single-click a tool, the cursor returns to pointer mode after the tool has been used once.

Update Project on Generate
When the preference is set on, Visual Architect sends Apple events to the Symantec Project Manager to add files, compile, and perform project maintenance.

Confirm Save
When this preference is set on, Visual Architect prompts you to save a resource file when closing it after modifications have been made. Otherwise, Visual Architect automatically saves modified resource files on closing and on generating code.

Set Generate File

Lets you set the macro file Visual Architect uses to control code generation for the active resource file. By default, Visual Architect uses the file `GenerateTCLApp`, which it looks for first in the Symantec tree and then in your project tree. The **Set Generate File** command lets you override this default use of `GenerateTCLApp`.

Note

> Chapter 35, "VA: Symantec Project Manager Menu," contains details on the use of macro files.

Visual Architect Edit Menu 30

This reference chapter provides a detailed description of the commands in the Visual Architect **Edit** menu. Descriptions are also provided of the dialog boxes opened from commands on this menu, including the **Classes**, **Data Members**, **Commands**, and **Menu Bar** dialog boxes.

Commands in the Edit Menu

The **Edit** menu contains standard editing commands (**Undo**, **Cut**, **Copy**, **Paste**, **Clear**). In addition, it contains commands for manipulating many of an application's Visual Architect objects. Figure 30-1 displays the commands on the **Edit** menu.

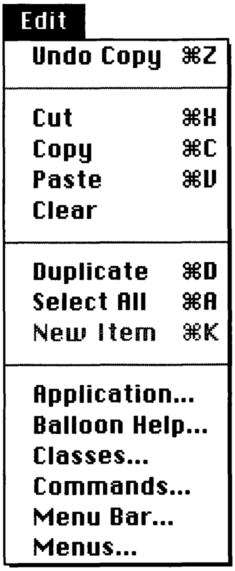


Figure 30-1 Edit menu

You can use the commands on the **Edit** menu to perform the following functions:

- Edit and manipulate code
- Edit application macros
- Add Balloon Help
- Edit classes, commands, and menus

These commands are covered in the order shown in Figure 30-1.

Editing and manipulating code

Use these commands to edit text and views.

Undo ⌘Z

Reverses the last edit operation. The command's name changes to reflect the operation you are undoing, for example, **Undo Paste**. Once you have undone an operation, the name of this command changes to **Redo**, allowing you to redo the operation that was just undone.

If there is nothing to undo in the frontmost window, this command is disabled.

Cut ⌘X

Removes the selected item and copies it to the Clipboard. This command replaces any previous contents of the Clipboard.

Visual Architect lets you cut an individual pane, a set of panes, or text. Only text can be pasted into other applications.

Copy ⌘C

Copies the selected item to the Clipboard. This command replaces any existing contents of the Clipboard.

Visual Architect lets you copy an individual pane, a set of panes, or text. Only text can be pasted into other applications.

Note

A **Cut** or **Copy** operation applied to a CPicture or CStaticText pane places the entire pane (not just the picture or text contained in the pane) on the Clipboard.

Paste ⌘V

Pastes the contents of the Clipboard into Visual Architect. If the contents of the Clipboard were cut or copied from Visual Architect, you can paste any pane, set of panes, or text. If the contents of the Clipboard were cut or copied from another application, only text and pictures can be pasted into Visual Architect.

The following rules determine the outcome of a paste operation. If a given set of conditions is not covered by one of these rules, choosing this command has no effect.

- If the Clipboard contains text and the frontmost window contains an active textbox, the Clipboard contents are pasted into that textbox.

- If the Clipboard contains text, the frontmost window is a View Edit window, and a CStaticText pane is selected, the text in the pane is replaced with the Clipboard contents.

- If the Clipboard contains text, the frontmost window is a View Edit window, and no pane is selected, a new CStaticText pane is created and the contents of the Clipboard becomes the pane's text.

- If the Clipboard contains a picture, the frontmost window is a View Edit window, and a CPicture pane is selected, the picture in the pane is replaced with the Clipboard contents.

- If the Clipboard contains a picture, the frontmost window is a View Edit window, and no pane is selected, a new CPicture pane is created and the contents of the Clipboard become the pane's picture.

Clear

Removes the selected text, pane, or set of panes, without affecting the contents of the Clipboard. You can also execute this command by pressing Clear or Delete.

Duplicate ⌘D

Creates a copy of the selected pane or panes in the frontmost View Edit window and pastes them into the same window. The new panes are positioned downward and to the right of the original pane. If no panes are selected, this command is disabled.

Select All ⌘A

Selects all text in an active textbox in the frontmost window. If that window is a View Edit window without an active text field, this command selects all panes in the window. Otherwise, the command is disabled.

New Item

Adds a new command, class, data member, menu, or menu item. The name of the command changes depending on the type of dialog box that is frontmost.

- If the **Commands** dialog box is frontmost, the command is named **New Command**. Choosing it enables you to define a new command to be added to your command list.
- If the **Classes** dialog box is frontmost, the command is named **New Class**. Choosing it enables you to define a new class to be added to your class command list.
- If the **Define Data Members** dialog box is frontmost, the command is named **New Data Member**. Choosing it defines a new data member to be added to your derived class.
- If the **Menus** or **Menu Bar** dialog box is frontmost, the command is named **New Menu**. Choosing it enables you to define a new menu to be added to your menu list. If the frontmost window is a **Menu Bar** dialog box, the new menu is also added to your menu bar.

These dialog boxes are discussed in greater detail later in this chapter.

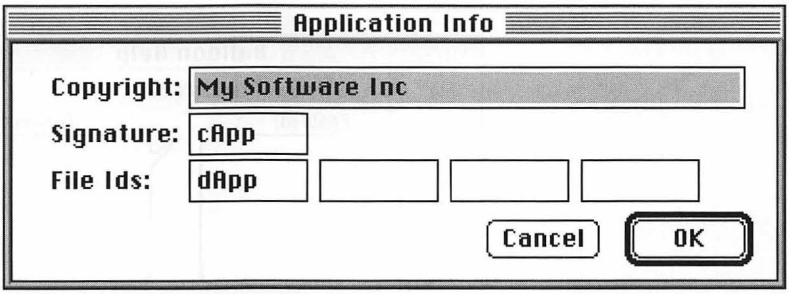
If none of these dialog boxes is frontmost, the command is disabled.

Editing application macros

You can specify values for three kinds of application macros.

Application

Opens the **Application Info** dialog box, which you use to edit values of the `copyright`, `signature`, and `docType` application macros.



Application Info

Copyright: My Software Inc

Signature: cApp

File Ids: dApp

Figure 30-2 Application Info dialog box

Note

These values should be set before generating code for the first time, because Visual Architect sets the values for the macros only once.

Copyright

You use this textbox to specify the value of the `copyright` macro during code generation, and to define the application copyright as it appears in the comments of generated code.

Signature

You can enter up to four characters in this textbox to specify the value of the `signature` macro during code generation. This macro is currently not used in the Visual Architect macro files. To define the application signature, you must change the upper-level application header file manually.

File IDs

The dialog box contains four text boxes, each holding a four-character string, for specifying the values of the `docType1`, `docType2`, `docType3`, and `docType4` macros during code generation. These macros are currently not used in the Visual Architect macro files. To define the file IDs, you must change the upper-level application header file manually.

Adding Balloon Help

Use this command to add Balloon Help to your application.

Balloon Help

Opens the Balloon Help window, which you can use to add help balloons to a view's panes (Figure 30-3). The command is available only when a View Edit window is frontmost; otherwise, it is disabled.

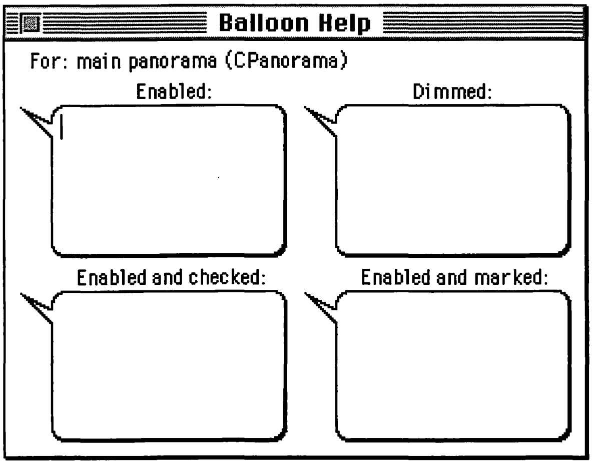


Figure 30-3 Balloon Help window

The four balloons correspond to the four possible states of a pane at run-time. In each balloon, you enter the text you want to appear when the user points to a pane in that state. (Note that, for Balloon Help to be displayed, the user must also have the Show Balloons command selected on the Balloon Help menu.) Refer to *Inside Macintosh* for details on the four balloon states.

The Balloon Help window is associated with the selected pane in the active View Edit window—or with the window itself, if no pane is selected. If you leave the window open and select another pane, the window is updated to reflect this new pane. The current pane or window object is noted at the top of the Balloon Help window.

Note

> This command cannot be used to define Balloon Help for menus created with Visual Architect. You must do this using a resource editor such as ResEdit.

Editing classes, commands, and menus

Use these commands to create and edit classes, commands, and menus.

Classes

Opens the **Classes** dialog box, which you use to create and define new derived classes as well as to delete and edit existing ones. As part of the process, you can specify the base class from which a new class is derived and define the class's data members. Visual Architect later generates C++ code for these classes.

The **Classes** dialog box is discussed in detail in the section "Classes Dialog Box" later in this chapter.

Commands

Opens the **Commands** dialog box, which you use to add, delete, and rename commands, as well as to change the handlers and actions associated with each command.

This dialog box is covered in detail in the "Commands Dialog Box" later in the chapter.

Menu Bar

Opens the **Menu Bar** dialog box, which you use to manipulate the menus in an application's menu bar. You can create menus and add menus to and remove them from a menu bar, as well as change the titles and relative positions of menus in the menu bar.

The **Menu Bar** and **Menu Items** dialog boxes are discussed in detail in the section "Menu Bar Dialog Box" later in this chapter.

Menus

Opens the **Menus** dialog box, which you use to work with the menus in an application. You can create and delete menus and change their titles, menu IDs and MDEF IDs. In addition, you can edit the commands on a menu.

The **Menus** dialog box is discussed in detail in the section "Menus Dialog Box" later in this chapter.

Classes Dialog Box

The **Classes** dialog box opens when you choose **Classes** from the **Edit** menu. The **Classes** dialog box is shown in Figure 30-4. Note that you can open the **Data Members** dialog box from this dialog box by clicking on the Define Data Members button. The latter dialog box is discussed in the section "Data Members Dialog Box," later in this chapter.

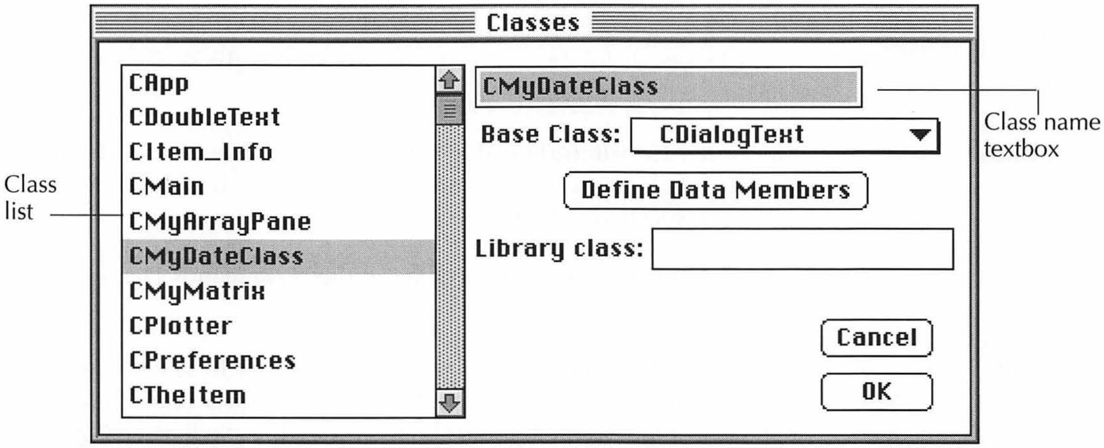


Figure 30-4 Classes dialog box

Class list
At the left of the dialog box is a scrolling alphabetic list of derived classes.

By selecting a class from this list, you can either delete it or change its name. To delete a class, press Delete. Note that application and view director classes that Visual Architect creates automatically cannot be deleted. To change a name, type the new name in the class name textbox.

Class name textbox
Any class you select from the class list is displayed in the class name textbox.

Press Return to clear the textbox before entering the name of a new class. (Alternatively, choose **New Class** from the **Edit** menu with the **Classes** dialog box open.) Select a base class from the pop-up menu and click OK to create the new class.

Note

If a derived class is in use, any change you make to it requires a corresponding change in all views containing objects of that class. Visual Architect makes these changes automatically.

If you change a class name, the new name replaces the old in the affected views. If you delete a class or change its base class, 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.

Base Class pop-up menu
This pop-up menu displays available THINK Class Library classes; you use the menu to select a base class for one of your derived classes.

Depending on your selection from the class list, the **Base Class** pop-up menu is disabled. You cannot use the menu for the application class or one of the director classes automatically created for views. Further, you cannot select one of these classes from the menu as a base class for a derived class. These classes are shown as disabled in the menu.

Unless you specify a library in the Library Class textbox, Visual Architect derives the lower-level class from the base class specified in this menu. (For more information on lower- and higher-level classes, see Chapter 35, "VA: Symantec Project Manager Menu.") The base class is used to select the appropriate macro file for code generation.

Define Data Members
Click this button to open the **Data Members** dialog box.

Library class textbox
Use this textbox to derive a class from one of your own library classes. After you type the name of the library class into this textbox, remember to add the source file for the library class to the project.

Visual Architect assumes that the library class displayed in this textbox is derived either directly or indirectly from the base class shown in the **Base Class** pop-up menu. The library name is assumed to be valid and not in conflict with any other class names.

Note

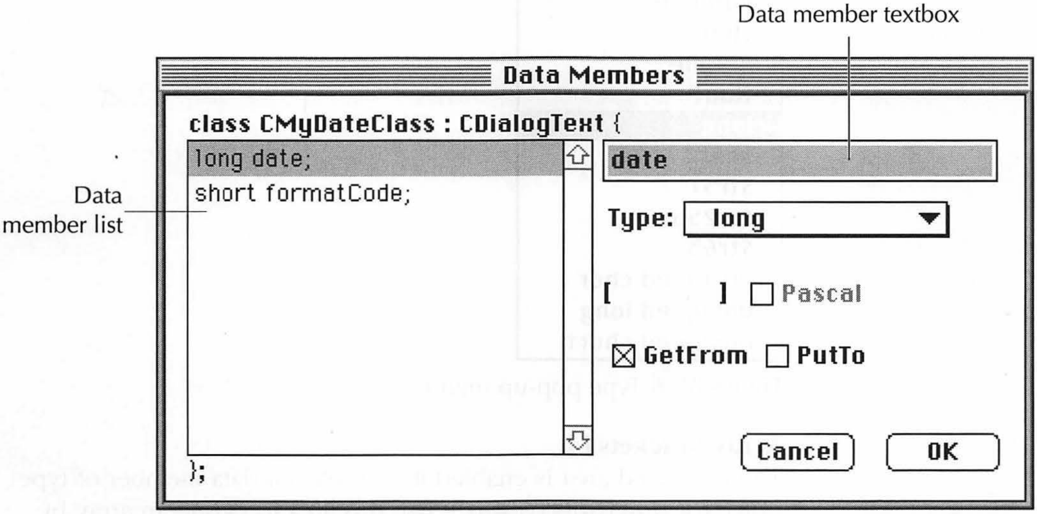
> Library classes and intermediate classes derived from the THINK Class Library base classes must not attempt to read `GetFrom` stream data members. See the electronic supplemental information for more information.

Data Members Dialog Box

From within Visual Architect, you can define data members for all derived classes (except application and director classes). Clicking the Define Data Members button in the **Classes** dialog box opens the **Data Members** dialog box (Figure 30-5).

This dialog box is not intended to provide a general-purpose data design capability. Rather, it lets you quickly parameterize user interface classes and assign preset values.

The values of data members of pane-derived classes can be set within Visual Architect through the Pane Info window (see Chapter 32, "Visual Architect Pane Menu").



```
class CMyDateClass : CDialogText {
long date;
short formatCode;
};
```

date

Figure 30-5 Data Members dialog box

Data member list
The scrolling list to the left shows the data members defined in the class, preceded by their respective types. Any data member you select from this list is displayed in the data member textbox. Data members are displayed in the list in the order in which you add them.

By selecting a data member from this list, you can either delete it or move it within the list. To delete a class, press Delete. To reorder the data members, drag the selected data member to a different position.

Data member textbox
This textbox displays the data member that is selected in the data member list.

Press Return to clear the textbox before entering the name of a new data member. (Alternatively, choose **New Data Member** from the **Edit** menu with the **Classes** dialog box open.) Select a type from the **Type** pop-up menu and click OK to create the new data member.

Type pop-up menu

This pop-up menu displays some of the types of data members possible (Figure 30-6).

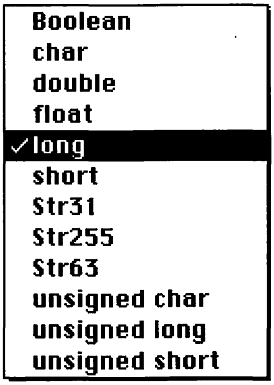


Figure 30-6 Type pop-up menu

Array brackets [ ]

The bracketed area is enabled if you select a data member of type `char` (or `unsigned char`). Use this area to define an array by typing a number between the brackets. This number indicates the length of the array.

Pascal

Set the Pascal option on if you want an `unsigned char` array to be interpreted as a Pascal string (beginning with a length byte) rather than a C string (ending with a zero byte).

Note

> Visual Architect only supports arrays of type `char` and `unsigned char` in the **Data Members** dialog box. See the introduction to the section for information on defining other data member types.

GetFrom and PutTo

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 resources. Object I/O output works by defining a `PutTo` function in each derived class. See the electronic supplemental information for more information on Object I/O.

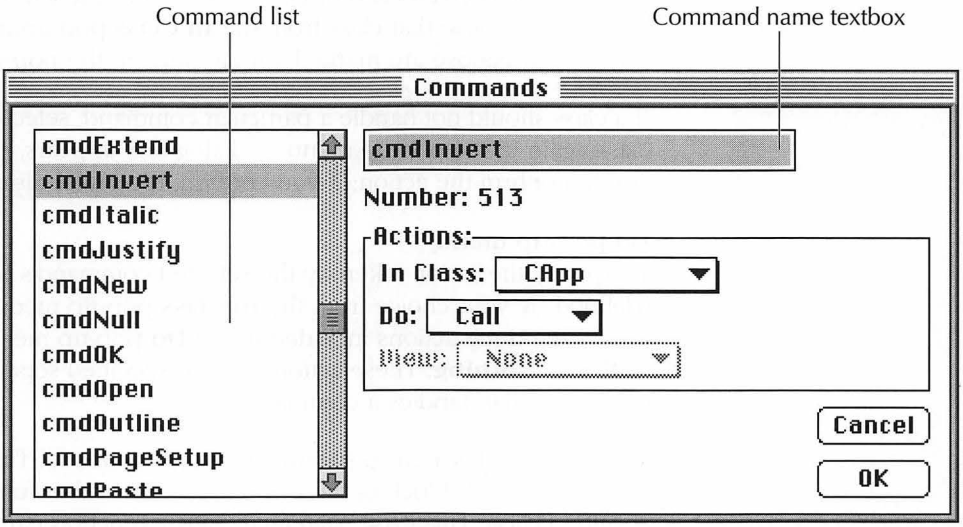
To have Visual Architect include the currently selected data member in the `GetFrom` and `PutTo` streams, set on either the GetFrom or PutTo options.

Note

> If you add or delete a data member or change its type, the old values of user-defined data members in existing objects of that class are discarded and the new values are initialized to binary zeros.

Commands Dialog Box

The **Commands** dialog box (Figure 30-7) opens when you choose **Commands** from the **Edit** menu.



Command list

Command name textbox

Commands

cmdInvert

Number: 513

Actions:

In Class: CApp

Do: Call

View: None

Figure 30-7 Commands dialog box

Command list
The command list shows all the predefined and user-defined commands.

By selecting a command from this list, you can either delete it or change its name. To delete a class, press Delete. To change a command's name, type the new name in the command name textbox. You cannot change the names of commands predefined by TCL.

Command name textbox
The command name textbox displays the name of the currently selected command from the commands list.

Press Return to clear the textbox before entering the name of a new command. (Alternatively, choose **New Command** from the **Edit** menu with the **Commands** dialog box open.)

Number
The value shown next to the Number item is the number associated with the command. Visual Architect automatically assigns numbers beginning with 512 to new commands. You never need to deal with command numbers directly.

In Class pop-up menu
To select the derived class or classes that handle a selected command, choose that class from the **In Class** pop-up menu. The classes you choose are marked with a bullet in the pop-up menu.

If a class should not handle a particular command, select the marked class again in the pop-up menu. A dialog box appears, prompting you to confirm the action; if you confirm, the bullet disappears.

Do pop-up menu
To specify the action taken by the selected command's handler (defined by your choice from the **In Class** pop-up menu), choose one of the three actions included in the **Do** pop-up menu: **Open**, **Call**, and **Nothing**. These actions can be specified separately for each class that handles a command.

Open. Generates a `case` statement for the command. This statement is in the `switch` block of the `DoCommmand` member function of the specified class. This `case` statement calls a function named `Do`*Commandname*. The `Do`*Commandname* function is generated into the lower-level of the specified class. Code generated for this function opens and selects the specified dialog box or window. Typically, you do not need to override this function in the upper-level of the specified classes.

The following is an example of the code generated into the lower-level class when you choose the **Open** action:

```
/*************************************************
 DoCommand {OVERRIDE}

   Dispatch Visual Architect-specified actions.

*************************************************/

void x_CMyDialog::DoCommand(long theCommand)

{
   switch (theCommand)
   {
      case cmdMyCommand:
         DoCmdMyCommand();
         break;
      default:
         CDialogDirector::DoCommand(theCommand);
   }
}


/*************************************************
 DoCmdMyCommand

   Respond to cmdMyCommand command.

*************************************************/

void x_CMyDialog::DoCmdMyCommand()

{
   CMyDialog*dialog;

      // Respond to command by opening a dialog

   dialog = TCL_NEW(CMyDialog,());
   dialog->ICMyDialog(this);
}
```

Call. Generates a `case` statement for the command. This command is in the `switch` block of the `DoCommand` member function of the specified class. This `case` statement calls a function named `Do`*Commandname*. The `Do`*Commandname* function is generated into the lower-level of the specified classes. Unlike the **Open** action, the `Do`*Commandname* function is empty; you must override it in the upper-level of the specified classes if you want it to do something.

The following is an example of the code generated into the lower-level class when the **Call** action is chosen:

```
/***********************************************
 DoCommand {OVERRIDE}

   Dispatch Visual Architect-specified actions.

***********************************************/

void x_CMyDialog::DoCommand(long theCommand)

{
   switch (theCommand)
   {
      case cmdMyCommand:
         DoCmdMyCommand();
         break;
      default:
         CDialogDirector::DoCommand(theCommand);
   }
}

/***********************************************
 DoCmdMyCommand

   Respond to cmdMyCommand command.

***********************************************/

void x_CMyDialog::DoCmdMyCommand()

{
      // Subclass must override this function to
      // handle the command
}
```

Note

You may find it just as convenient to leave the action of a command unspecified, and enter the code yourself as a DoCommand case in an upper-level class, thus avoiding an extra level of indirection.

Nothing. Generates a `case` statement for the command. This statement is in the `switch` block of the `DoCommand` member function of the specified class. Unlike the **Open** or **Call** actions, no statement is added to the `case`. This `case` only prevents supervisors from receiving the command. In order for the command to have any action, you must create a `case` for it in the `switch` block of the DoCommand member function of the upper-level of the specified class.

The following is an example of the code generated into the lower-level class when you choose **Nothing**:

```
/***********************************************
 DoCommand {OVERRIDE}

   Dispatch Visual Architect-specified actions.

***********************************************/

void x_CMyDialog::DoCommand(long theCommand)

{
   switch (theCommand)
   {
      case cmdMyCommand:
         break;
      default:
         CDialogDirector::DoCommand(theCommand);
   }
}
```

View pop-up menu

This menu becomes available when you choose **Open** from the **Do** pop-up menu. Choose the view that you want the command to open.

For information on command handling, see the section on this topic at the end of the chapter.

Menu Bar Dialog Box

Choose **Menu Bar** from the **Edit** menu to open the **Menu Bar** dialog box (Figure 30-8).

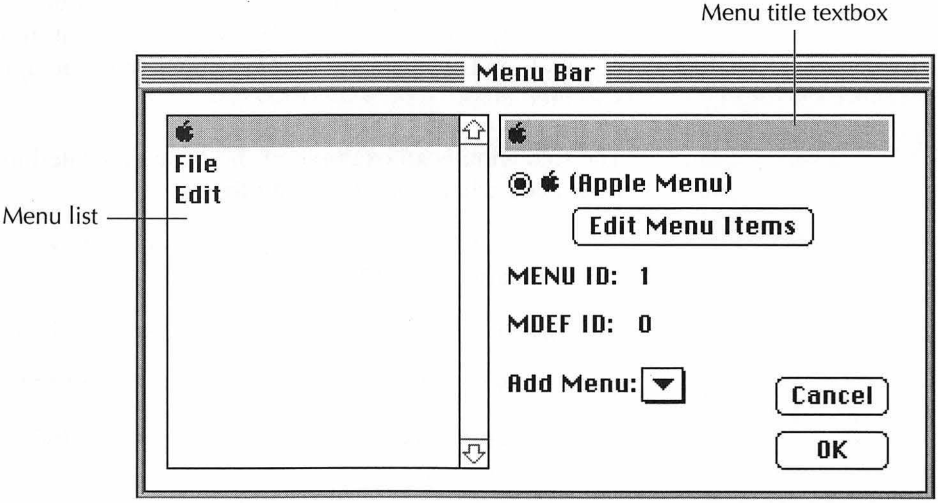


Figure 30-8 Menu Bar dialog box

Menu list
The scrolling menu list shows the menus in the order they appear in the menu bar.

By selecting a menu from this list, you can either delete it or change its position in the menu bar. To delete a menu from the menu bar, press Delete. (The menu is not deleted, just removed from the menu bar.) To change a menu's position in the menu bar, drag the selected menu to a new position. To change the name of a selected menu, type the new title in the menu title textbox.

Note

> The top-to-bottom position of menu titles in the menu list corresponds to the left-to-right position of the menus in the application's menu bar.

Menu title textbox

The menu title textbox displays the name of the currently selected menu from the menu list.

Press Return to clear the textbox before entering the name of a menu you want to create. (Alternatively, choose **New Menu** from the **Edit** menu with the **Menu Bar** dialog box open.)

Apple Menu radio button

To specify the Apple menu, click the " (Apple Menu)" radio button.

Edit Menu Items

Click this button to open the **Menu Items** dialog box, which is discussed in the subsequent section.

MENU ID

When you create or import a menu, Visual Architect automatically assigns that menu a unique menu and resource ID. You can change the menu ID for the selected menu by entering a different value in this textbox. Visual Architect automatically sets the resource ID to be the same as the specified menu ID. In this way, 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.) You seldom need to change this field.

MDEF ID

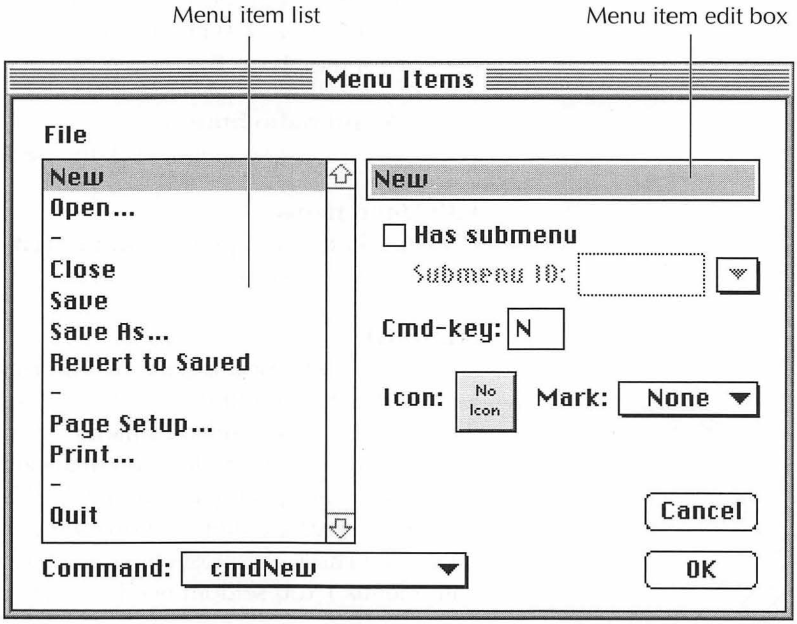
The MDEF ID field lets you use a custom menu definition procedure (MDEF) for the selected menu. The system default value is 0. You need to change this field only if you want to use a custom menu definition procedure. Refer to *Inside Macintosh* for details on creating a custom MDEF.

Add Menu pop-up menu

Use this pop-up menu to add an existing menu to the menu bar. Any menu you add in this way is displayed in the menu list.

Menu Items Dialog Box

Clicking on the Edit Menu Items button in the **Menu Bar** or **Menus** dialog box opens the **Menu Items** dialog box (Figure 30-9).



New

□ Has submenu

Submenu ID:

Cmd-key: N

Icon: No Icon

Mark: None

Command: cmdNew

Figure 30-9 Menu Items dialog box

Menu item list
The scrolling menu item list shows the menu items in order of their appearance in the menu.

By selecting a menu item from this list, you can either delete it or change its position in the menu. To delete a menu item from the menu, press Delete. To change a menu item's position in the menu, drag the selected menu item to a new position in the list.

Menu title textbox
The menu item textbox displays the name of the currently selected menu item from the menu items list.

Press Return to clear the textbox before entering the name of a menu item you want to create. (Alternatively, choose **New Menu Item** from the **Edit** menu with the **Menus** or **Menu Bar** dialog box open.) The menu item you created is added to the menu and its name is added to the menu item list.

Has Submenu and Submenu ID pop-up menu

You use the Has Submenu option to designate the selected menu item as a submenu. Setting this option on enables the Submenu ID textbox and pop-up menu, in which you specify the submenu. You can either enter a valid menu ID in the textbox or choose a menu from the pop-up menu.

Cmd-key

Enter a value in the Cmd-key textbox to specify a command key equivalent for the selected menu item. This textbox is disabled if the menu item is a submenu.

Note

> Do not use Command-key equivalents 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. Because a pop-up pane is not a candidate for the gopher and pop-up panes seldom supervise anything, these conditions cannot be met.

Icon

Click the Icon button to open the **Icon Pick** dialog box (Figure 30-10). You can use this dialog box to place an icon next to a menu item.

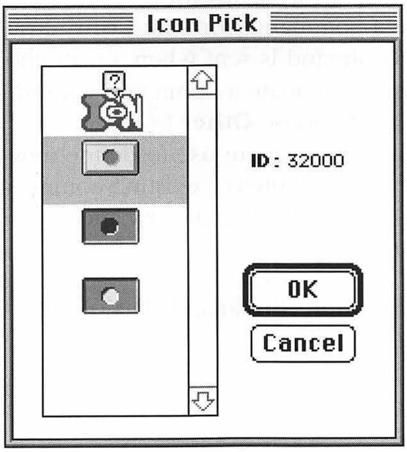


Figure 30-10 Icon Pick dialog box

The **Icon Pick** dialog box shows all icon resources that have both `'ICON'` and `'cicn'` (color icon) resources in the active resource file. Choose an icon from the list and click OK. You can place `'ICON'` and `'cicn'` resources into the resource file using a standard resource editor, such as ResEdit or Resorceror.

Mark pop-up menu
Choose a marker from the **Mark** pop-up menu to specify a default mark to be placed next to the selected menu item (Figure 30-11).

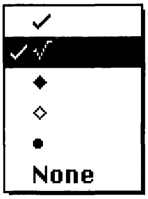


Figure 30-11 Mark pop-up menu

Note

Only the Chicago font provides all the characters listed in the Mark pop-up menu. For pop-up menus in other fonts, be sure to select a marker that exists in that font. The • (bullet) character, for example, exists in all fonts.

Command pop-up menu
Choose a command from the **Command** pop-up menu to identify which command is sent when a user chooses the menu item from the menu. To create a command without closing the **Menu Items** dialog box, choose **Other** from the list. The **Commands** dialog box opens, which you can use to define new commands as well as to change the attributes of existing commands. When you dismiss this dialog box by clicking OK or Cancel, you return to the **Menu Items** dialog box.

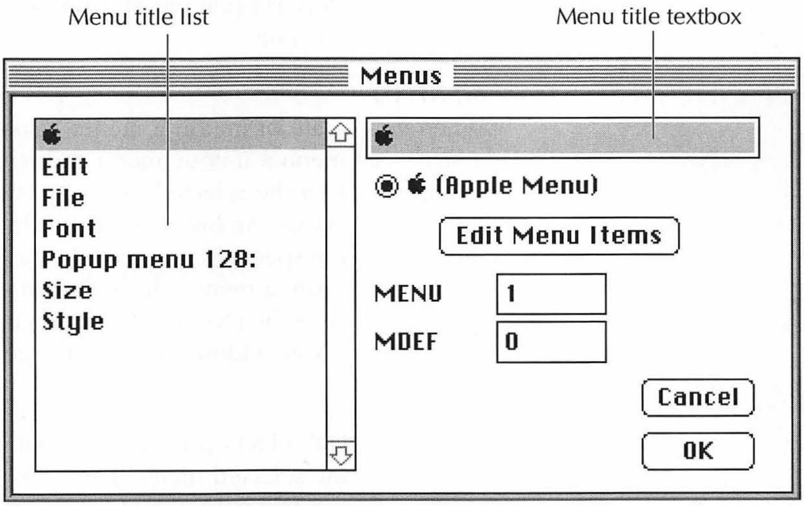
See the section "Commands Dialog Box" earlier in this chapter for more information.

Note

To define color menu items, use a standard resource editor such as ResEdit. Note that the Macintosh Human Interface Guidelines recommend coloring menus sparingly or not at all.

Menus Dialog Box

Clicking on **Menus** in the **Edit** menu opens the **Menus** dialog box (Figure 30-12).



MENU 1

MDEF 0

Figure 30-12 Menus dialog box

Menu list
The scrolling menu list shows the menus in alphabetic order.

By selecting a menu item from this list, you can either delete it or edit its title (name). To delete a menu from the resource file, press Delete. To change a menu's title in the menu list, type the new title in the menu title textbox.

Menu title textbox
The menu title textbox displays the name of the currently selected class from the menu list.

Press Return to clear the textbox before entering the name of a menu you want to create. (Alternatively, choose **New Menu** from the **Edit** menu with the **Menus** dialog box open.)

Apple Menu
To specify the Apple menu, click the " (Apple Menu)" radio button.

Edit Menu Items
Click this button to open the **Menu Items** dialog box, discussed in the previous section

MENU ID
When you create or import a menu, Visual Architect automatically assigns that menu a unique menu and resource ID. You can change the menu ID for the selected menu by entering a different value in this textbox. Visual Architect automatically sets the resource ID to be the same as the specified menu ID. In this way, 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.) You seldom need to change this field.

MDEF ID
The MDEF ID field lets you use a custom menu definition procedure (MDEF) for the selected menu. The system default value is 0. You need to change this field only if you want to use a custom menu definition procedure. Refer to *Inside Macintosh* for details on creating a custom MDEF.

Command Handling in Generated Code

This section provides information about the code you write in addition to the code generated by Visual Architect to handle commands.

AppCommands.h

Symbols for commands you define in Visual Architect are generated to the file AppCommands.h. You should include this file in the source file of upper-level classes that need to refer to your commands. In fact, an include statement is generated by Visual Architect into all upper-level class files, as shown here:

```
//#include "AppCommands.h"
// Remove comments if DoCommand overridden
```

It is commented out by default to prevent unnecessary compilation whenever AppCommands.h is changed. If you want AppCommands.h included, remove the comments.

Chain of command

The object that handles a command must normally be located at a higher position in the chain of command than the object that sends the command. For example, commands sent by menu bar menus can be handled by the gopher or any object higher than the gopher in the chain of command.

The exception to the chain of command rule is a floating window. The floating window director relays to the gopher all commands it receives and does not handle (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. A tear-off menu sends the same command, whether or not the menu is torn off.

Commands in modal dialog boxes

In modal dialog boxes, 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.

Typically, 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 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 dialog boxes should not send a cmdClose command.

Closing a modeless dialog box

The Macintosh Human Interface Guidelines recommend that a modeless dialog box be closed only by choosing the **Close** command or clicking 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.

Visual Architect View Menu 31

This reference chapter provides detailed explanations of the commands on the Visual Architect **View** menu. In addition, the four types of dialog boxes you can open by choosing **View Info** from the menu—Dialog Info, Main Window Info, Floating Window Info, and Subview Info—are described.

Commands in the View Menu

You use the **View** menu commands to manipulate the views you create in Visual Architect. These commands let you create, edit, and try out your views. Figure 31-1 displays the commands in the **View** menu.

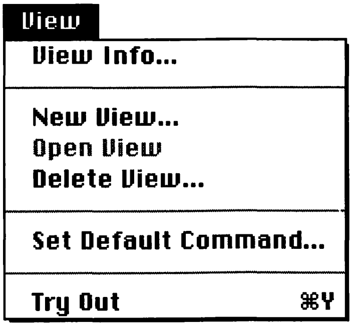


Figure 31-1 View menu

You can use these commands to perform the following functions:

- Examine and edit attributes of views
- Create, open, and delete views
- Set default commands
- Preview views

This chapter discusses the **View** menu commands by function in the order shown above. The four types of views opened using the **View Info** command are described in detail at the end of the chapter, as are the different types of views you can insert using Visual Architect.

Examining and editing attributes of views

This command allows you to change the fixed set of attribute values of an existing view.

View Info

Opens a **View Info** dialog box, which you use to examine and edit the general attributes of the view whose View Edit window is frontmost. If no View Edit window is frontmost, the command is disabled. To open a View Edit window for a view, choose **Open View**.

There are four different types of **View Info** dialog boxes. The one that opens depends on the type of view.

Table 31-1 View Info dialog boxes

Dialog box type	**View type**
Dialog Info	Dialog, Modal Dialog, New...Dialog, Splash Screen, Window
Main Window Info	Main Window
Floating Window Info	Floating Window, Tear-off Menu
Subview Info	Subview

Note

> Although some view types share a common **View Info** dialog box, the title for each **View Info** dialog box displays the name of the view it represents. For example, the **View Info** dialog box for Modal Dialog views is titled **Modal Dialog Info**.

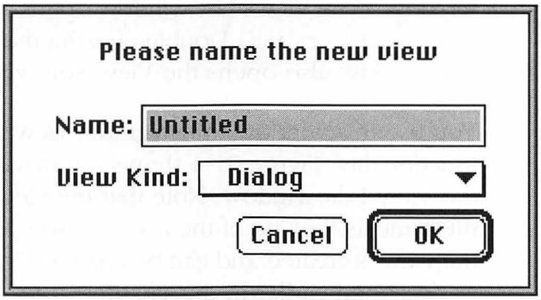
These four types of dialog box are described later in the chapter.

Creating, opening, and deleting views

This set of commands lets you create a new view, delete an existing view, or display an existing view and change its visual components.

New View

Opens the **New View** dialog box, which you use to create a new view for an application. In this dialog box, you are prompted to name the new view and specify the view's type (Figure 31-2).



Please name the new view

Name: Untitled

View Kind: Dialog

Figure 31-2 New View dialog box

Name

The name you specify in this textbox 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 already in the project, you will be prompted to specify another name.

View Kind pop-up menu

You choose the view type from the **View Kind** pop-up menu, as shown in Figure 31-3.

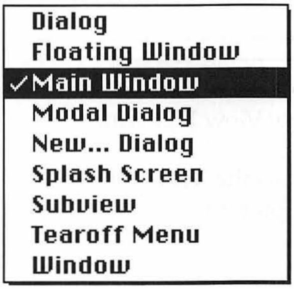


Figure 31-3 View Kind pop-up menu

Choose the view type that best fits the role of the window or subview you want to design. Then click OK to create the new view. A View Edit window is displayed for the view, and the view's name is added to the **Windows** menu. See the **Open View** command for information on the View Edit window. See the section "View Types" at the end of the chapter for more information on view types.

Open View

Opens a View Edit window for the view selected in the Views List window. (The Views List window opens when you click on `Visual Architect.rsrc`.) Double-clicking the view's name in the Views List window also opens the View Edit window for that view.

You use this window to design the view's appearance and functionality. Figure 31-4 shows the name of the view as the title of the View Edit 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, as described earlier in this chapter.

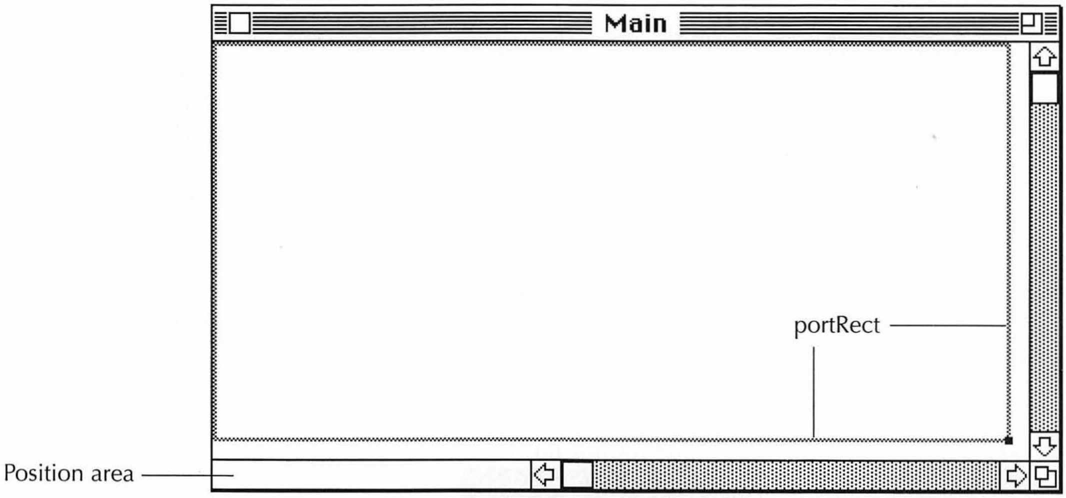


Figure 31-4 Empty View Edit window

Panes are added to the View Edit window using the Tool palette, described in Chapter 34, "Visual Architect Tools Menu."

portRect
The thick gray rectangle outline extending from the upper-left corner to the lower-right corner of the View Edit window is the portRect. The portRect defines the size of the user window or subview you are editing. You can make the user window larger or smaller by dragging the sizing handle in the lower-right corner. To see a window exactly as it will look to the user, choose **Try Out** from the **View** menu (Command-Y) at any time.

Note that the drawing area extends beyond the portRect; as a result, you can edit objects that are outside the user window. This is 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 are displayed inside the portRect, showing the space occupied by these 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 31-4. When the Show Position option is set on (see Chapter 33, "Visual Architect Options Menu"), the coordinates of the currently selected object or objects are shown in this area. These coordinates are useful for resizing the portRect.

Delete View

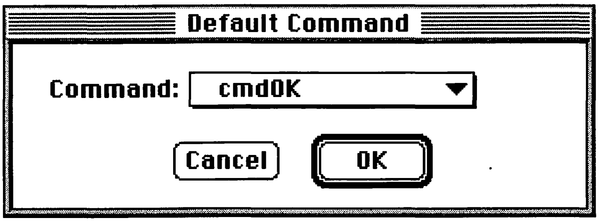
Lets you delete the view associated with the frontmost window. If the frontmost window does not belong to a view, the view whose name is selected in the Views List window is deleted. A dialog box opens, prompting you to confirm the delete operation.

Setting default commands

This command determines the outcome of Enter or Return being pressed.

Set Default Command

Opens the **Default Command** dialog box, which you use to set the default command for the view whose View Edit window is frontmost (Figure 31-5). The default command for a view is the command sent when the user presses Return or Enter—that is, the command sent by the CButton item with the double border.


Default Command

Command: cmdOK

Figure 31-5 Default Command dialog box

Choose the default command from the **Command** pop-up menu and click OK.

Previewing views

This command lets you display the visual structure of a created view.

Try Out ⌘Y

Opens a preview window for the view associated with the frontmost window. There, you can preview a view to verify its look and feel in the running application. All panes in the preview window are active and function in the same way as in the application. For example, radio buttons obey button group assignments, pop-up menus work, and scrolling text panes allow text entry and scrolling.

With this feature, you can test a view without having to go through a complete development cycle of generating code with Visual Architect, updating the project and building the application in the Symantec Project Manager, and running the application.

Note

Although panes are active, no commands other than `cmdOK` and `cmdCancel` are handled in Try Out mode.

To close the preview window, do one of the following:

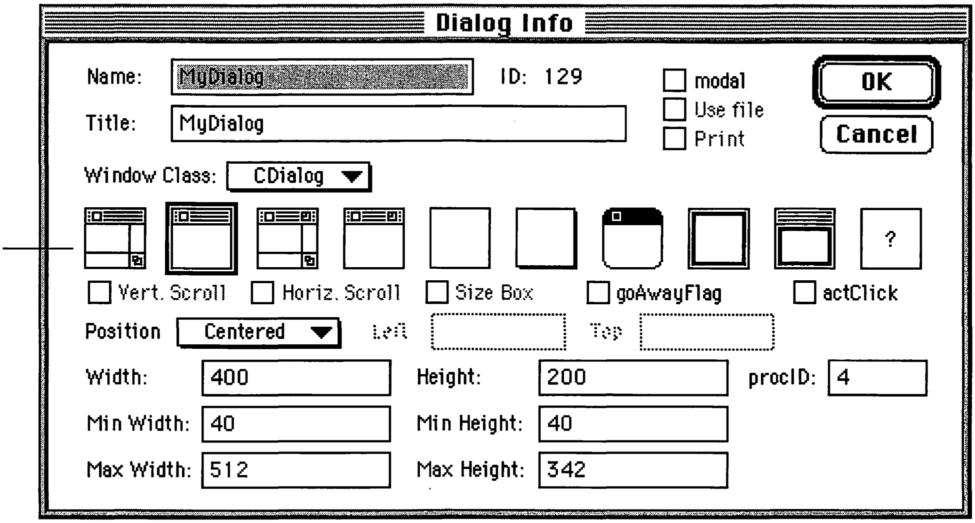
- Click the close box for the window, if one exists.
- Click a button that, in the running application, closes the window.
- Choose **Close** from the **File** menu (Command-W).

Types of View Info Dialog Boxes

Choosing the **View Info** command opens one of four types of dialog boxes. The type of dialog box that opens depends on the type of view whose View Edit window is frontmost.

Dialog Info dialog box

The **Dialog Info** type of dialog box (Figure 31-6) is used for Modal Dialog Info, New...Dialog Info, Splash Screen Info, and Window Info views.



Dialog Info

Name: MyDialog ID: 129

Title: MyDialog

□ modal
□ Use file
□ Print

Window Class: CDialog ▼

□ Vert. Scroll □ Horiz. Scroll □ Size Box □ goAwayFlag □ actClick

Position Centered ▼ Left Top

Width: 400 Height: 200 procID: 4

Min Width: 40 Min Height: 40

Max Width: 512 Max Height: 342

Window types

Figure 31-6 Dialog Info dialog box

Name

This textbox displays the name of the view, which is also the name of the view resource in the Visual Architect resource file. Note that if you change the view name displayed:

- The names of all the data members that represent the view's panes change, because their names are constructed using the view name (for example, `CButton *MyView_Button1`).
- A new *viewname*`items.h` file is generated because this file's name incorporates the view name. The old file, rendered obsolete, should be discarded.
- New upper- and lower-layer class files are generated for the view because the files' names incorporate the view name. The old files, rendered obsolete, should be discarded.

As a result of a change in view name, any upper-layer files that refer to the view's panes by name or include the old items file no longer compile. You must change these references manually.

ID

The view's resource ID in the Visual Architect resource file is displayed next to the Name textbox. This ID is set automatically by Visual Architect when the view is created; you cannot change the ID.

You can use the procID field to identify a `'WDEF'` window definition resource that you have added to the active resource file manually, using a standard resource editor, such as ResEdit or Resorceror. Refer to *Inside Macintosh* for details on `'WDEF'` and procID.

Title

This textbox displays the title of the window or dialog box as it is shown in the application.

Modal

When this option is set on, the view is modal. This option only applies to Dialog views.

Window Class
Use this pop-up menu to specify the CWindow- or CDialog-derived class for the view. You can choose CWindow, CDialog, or any CWindow-derived class you have defined through the **Classes** dialog box.

Window Types
Icons for the ten window types are displayed below the **Window Class** pop-up menu. Numeric procIDs, which are used to define these types, can be set using these icons. Refer to *Inside Macintosh* for details on procIDs.

Note that if you select the icon with the question mark, you must supply the procID.

Vert. Scroll
Setting this option on provides the view with a vertical scroll bar. In this case, the main panorama of the view is enclosed in a CScrollPane.

Horiz. Scroll
When this option is set on, the view has a horizontal scroll bar. In this case, the main panorama of the view is enclosed in a CScrollPane.

Size Box
When this option is set on, the view has a size box. In this case, the main panorama of the view is enclosed in a CScrollPane.

goAwayFlag
When this option is set on, the view has a goAwayFlag. This flag is not relevant if the view is a modal dialog box.

actClick
Typically, if the user clicks a view in the background, the view is activated (brought to the foreground) but the pane that was clicked does not receive the click message. If actClick is set on, the pane does receive the click message.

Position
Use this pop-up menu to determine whether the view initially is centered, staggered, or placed with its upper-left corner at a fixed position.

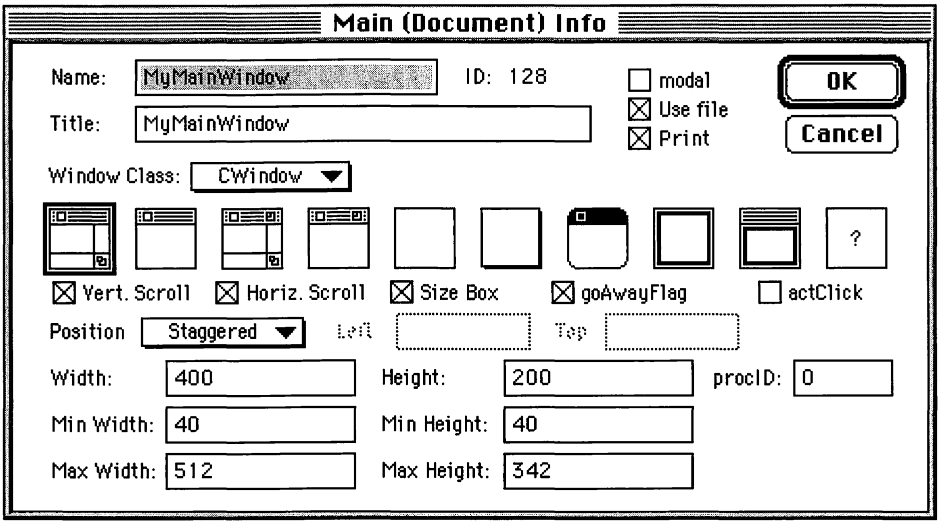
Left, Top
These textboxes are enabled if you choose **Fixed** from the **Position** pop-up menu. Type in values for the upper-left corner of the view.

Width, Height
Use these textboxes to establish the initial width and height of the view.

Min Width, Min Height, Max Width, Max Height
Use these textboxes to determine the dimensions for the view's sizeRect.

Main Window Info dialog box

The Main Window Info type of dialog box is used for a single type of view: Main Window. An example of the **Main Window Info** dialog box is shown in Figure 31-7.



Main (Document) Info

Name: MyMainWindow ID: 128

Title: MyMainWindow

☐ modal
☒ Use file
☒ Print

Window Class: CWindow

☒ Vert. Scroll ☒ Horiz. Scroll ☒ Size Box ☒ goAwayFlag ☐ actClick

Position Staggered Left Top

Width: 400 Height: 200 procID: 0

Min Width: 40 Min Height: 40

Max Width: 512 Max Height: 342

Figure 31-7 Main Window Info dialog box

Most of the options in this dialog box are the same as those displayed in the **Dialog Info** dialog box (see the previous section). The options that are different or are used differently are described in the following sections.

Modal
Because the Main Window view typically should be modeless, set this box option off.

Use File
Setting this option on allows the Main Window view to support open, save, and revert file functions. In this case, the director class is derived from CSaver. When it is set off, the Main Window view has no associated file and the director class is derived from CDocument.

Note

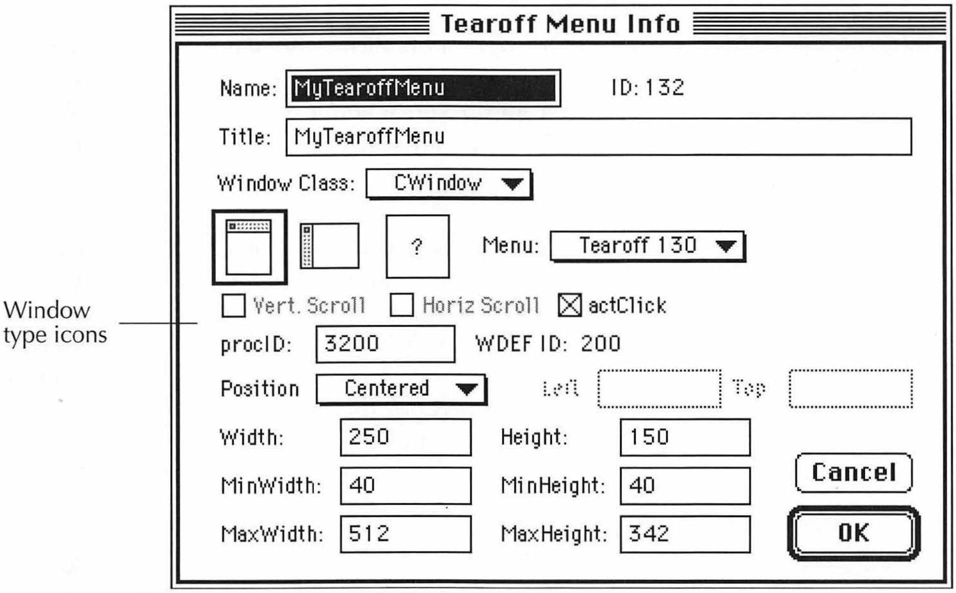
> You can derive your document class from a THINK Class Library class named CSimpleSaver. As its name suggests, this class is easier to use than CSaver; it does not use Object I/O. To use CSimpleSaver, leave the Use File option off, and, in the **Classes** dialog box, set the library class of your document class to CSimpleSaver.

Print
When this option is set on, the document is initialized as printable. Refer to the information on the classes CDocument and CPrinter in the online *THINK Reference* for details.

Window Class
Typically, a Main Window view's window object is derived from CWindow. Use the **Window Class** pop-up when you have defined a CWindow-derived class you want to use for a view. A Main Window view can also use a CDialog-derived window, but its director class is always derived from CDocument, not from CDialogDirector.

Floating Window Info dialog box

The **Floating Window Info** dialog box type is used for Floating Window and Tear-off Menu views. The difference in dialog box layout for the two views is the **Menu** pop-up menu. This menu is present only in the **Tear-off Menu Info** dialog box, as shown in Figure 31-8.



Tearoff Menu Info

Name: MyTearoffMenu ID: 132

Title: MyTearoffMenu

Window Class: CWindow

Menu: Tearoff 130

□ Vert. Scroll □ Horiz Scroll ☒ actClick

procID: 3200 WDEF ID: 200

Position Centered Left Top

Width: 250 Height: 150

MinWidth: 40 MinHeight: 40

MaxWidth: 512 MaxHeight: 342

Cancel OK

Figure 31-8 Tear-off Menu Info dialog box

Most of the options in these dialog boxes are the same as those in the **Dialog Info** dialog box discussed previously. Following is a discussion of the items that are different.

Window Type

The window type icons represent the procID for the window. The standard floating window `'WDEF'` supports a drag bar at the top of the left side of the window. Click one of the first two icons to select the floating window `'WDEF'`. The procID corresponding to that window is then displayed in the procID box. When you click the icon with the question mark, you must supply the procID yourself.

Menu

The **Menu** pop-up menu lets you choose the menu that will be displayed in the Tear-off Menu view.

WDEF ID

The resource ID of the `'WDEF'` is displayed next to the procID textbox. This value is set automatically according to the following formula:

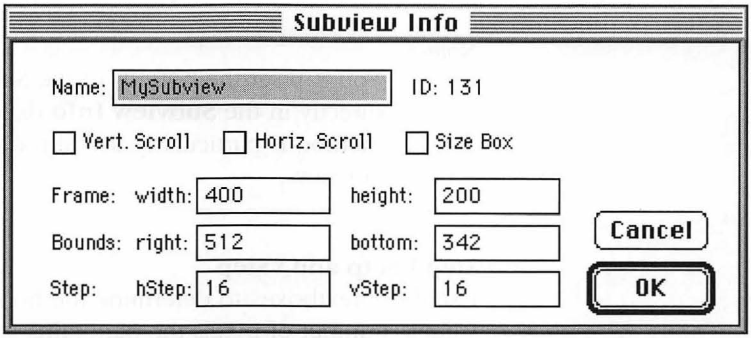
```
WDEF ID = (procID - variation code) / 16
```

Refer to Window Manager documentation in *Inside Macintosh* for more information on the `'WDEF'` ID.

Because the Floating Window and Tear-off Menu views cannot use files, be modal, or print, the check boxes for these options are not present in these **View Info** dialog boxes.

Subview Info dialog box

The Subview Info type of dialog box is only used for Subviews. An example of this dialog box is shown in Figure 31-9.



Subview Info

Name: MySubview ID: 131

□ Vert. Scroll □ Horiz. Scroll □ Size Box

Frame: width: 400 height: 200

Bounds: right: 512 bottom: 342

Step: hStep: 16 vStep: 16

Figure 31-9 Subview Info dialog box

The subview is not a window but a panorama (with an optional scroll pane). (Refer to Chapter 28, "Programming with the THINK Class Library," for a description of panoramas.) The options in the dialog box are described in this section.

Name

This textbox displays the name of the subview. See the description of the Name option in the section "Dialog Info dialog box" earlier in this chapter for general notes on changing a view name.

ID

The unique ID for this view is shown next to the Name textbox.

Vert. Scroll

When this option is set on, the subview has a vertical scroll bar. In this case, the subview's main panorama is enclosed in a CScrollPane.

Horiz. Scroll

When this option is set on, the subview has a horizontal scroll bar. In this case, the subview's main panorama is enclosed in a CScrollPane.

Size Box
When this option is set on, the subview has a size box. In this case, the subview's main panorama is enclosed in a CScrollPane.

Frame Width and Height
Use these textboxes to determine the width and height of the subview.

Bounds Right and Bottom
Indicate the bottom-right corner of the panorama's bounds rectangle in these textboxes. 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 hSetp and vStep
Use these textboxes to determine the horizontal and vertical step values (number of pixels the panorama scrolls in a single step).

View Types

You can choose from nine types of views from the **View Kind** pop-up menu.

Dialog

A Dialog view may be used in any context, but typically it is used to present and gather information. Dialog views are preset to modeless. These views use a CDialog window and are implemented using a CDialogDirector-derived director.

Floating Window

A Floating Window view is drawn in front of all nonfloating views. This type of view is often used as a palette—for example, as a collection of drawing tools, colors, or patterns from which the user can select. A Floating Window view is never activated or deactivated. A CSelector-derived panorama displays the palette and monitors selections. At run-time, all Floating Window views 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.

Note

Do not use the Floating Window view to implement a tear-off menu. Instead, you should use the Tear-off Menu view described later in this chapter.

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. By default, a Main Window view is created when the user chooses **New** from the application's **File** menu.

A document-based application typically defines just one Main Window view, but it may define more if it edits more than one document type. In any case, multiple Main Window views can be open simultaneously, for example, when the user has multiple documents open. A Main Window view uses a CWindow window by default, but can instead use a CWindow-derived or CDialog-derived window. A Main Window view is always implemented using a CDocument-derived director. If you set the Use File option on in the **Main Window Info** dialog box for the view, the document class is derived from CSaver.

Modal Dialog

A Modal Dialog view is essentially the same as a Dialog view, but it 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

You use a New Dialog view when you have more than one Main Window view. A New Dialog view (which appears as **New... Dialog** on the **View Kind** pop-up menu) is a Modal Dialog view used to select the type of document to create when **New** is chosen from the **File** menu. The New Dialog view uses a CDialog-derived window and is implemented using a CDialogDirector-derived director.

Splash Screen

A Splash Screen view is essentially the same as a Dialog view, but it is displayed as soon as possible after the application launches and is removed as soon as initialization completes. A Splash Screen view 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 is not a window. Instead, it is a panorama. Using a subview, you can define a pane independently from the windows that display it. This is a useful option when you want to use the same pane in more than one window or edit a window with a deeply nested View hierarchy. A subview can scroll and can contain any other pane types, including other subviews.

Creating a subview does not automatically define a derived class because a panorama-derived class is not strictly necessary in order to use a subview if the panes in the panorama provide an essential part of the user interface.

This is in contrast to other views, for which derived classes are defined automatically. However, if you want a subview's panorama to play an active role in the user interface (for example, receive commands), then you must derive your own panorama class and use it for the subview's panorama. To do so:

1. Use the **Classes** dialog box to define your own derived class of CPanorama, as described in Chapter 30, "Visual Architect Edit Menu."

2. In the View Edit window for the subview, with no panes selected, choose **Class** from the **Pane** menu and choose your derived class from the submenu, as described in Chapter 32, "Visual Architect Pane Menu."

Tear-off Menu

A Tear-off Menu view is essentially the same as a Floating Window view, but it is used to implement a tear-off menu. A Tear-off Menu view uses a CWindow window and is implemented using a CTearOffMenu-derived director.

Window

A Window view is a plain director view, as distinguished from the Main Window view, which is a document view. A Window view is used when a document needs more than one window to display its contents. The primary document view is designated as a Main Window view, while all other document-related views are Window views. For this arrangement to work, the Window view must be supervised by the Main Window view. A Window view uses a CWindow window and is implemented using a CDirector-derived director.

Visual Architect Pane Menu ◆ 32

This chapter provides a detailed explanation of the commands in the Visual Architect **Pane** menu. You use these commands to change the characteristics of panes in your views. In addition, the options on the Pane Info window and ScrollPane Info window are described.

Commands in the Pane Menu

The **Pane** menu commands let you create and edit the panes in Visual Architect views. You can examine and change data members in a class hierarchy; specify font, size, and style for displayed text; and set a color for displaying a pane. You can also specify the arrangement of panes on the screen. Figure 32-1 displays the commands on the **Pane** menu.

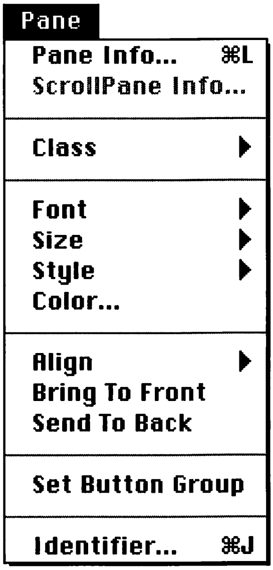


Figure 32-1 Pane menu

The **Pane** menu commands are used to perform the following functions:

- Edit data members in a class hierarchy
- Select classes for a pane
- Set display options for a pane
- Arrange panes in the View Edit window
- Group radio button panes
- Change the identifiers for panes

This chapter discusses the **Pane** menu commands by function in the order listed above.

Editing data members in a class hierarchy

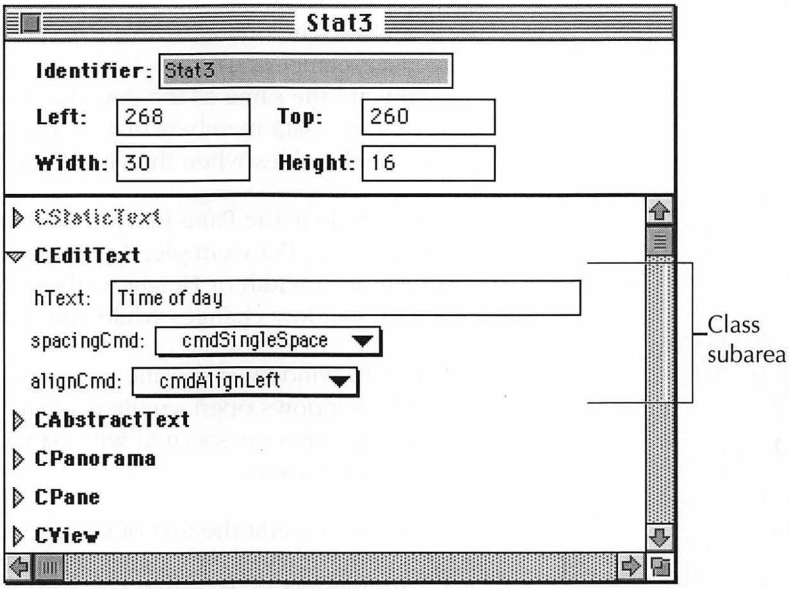
The following two commands open the Pane Info and the ScrollPane Info windows.

Pane Info **⌘L**

Opens a Pane Info window for the currently selected pane in the frontmost View Edit window. In this window, you can examine and edit data members in the class hierarchy. If no View Edit window is frontmost or if no pane is selected, the command is disabled. The Pane Info windows are different for each pane class, but are organized similarly for all pane classes (Figure 32-2).

Note

> You can also open the Pane Info window by double-clicking a pane in the View Edit window.



Stat3

Identifier: Stat3

Left: 268 Top: 260

Width: 30 Height: 16

CStaticText

CEditText

hText: Time of day

spacingCmd: cmdSingleSpace

alignCmd: cmdAlignLeft

CAbstractText

CPanorama

CPane

CView

Figure 32-2 Pane Info window

The title of the Pane Info window identifies the pane. You can edit a pane's identifier in the Identifier textbox at the top of the Pane Info window or with the **Identifier** dialog box, as described in the section "Changing the identifiers for panes," later in the chapter.

The Left, Top, Width, and Height textboxes at the top of the Pane Info window let you change the position and size of the pane in relation 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 most deeply 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 in Visual Architect.)

The small triangles next to the class names let you access the contents of each class. The triangles exist in two states: closed, when they point to the right, and open, when they point down. Clicking a closed triangle opens a subarea below the class name and reveals the contents of that class. Clicking an open triangle closes the subarea and hides the contents of that class.

Each class subarea contains the editable subset of the data members for that class. The online *THINK Reference* contains definitions of these data members. The labels shown in the Pane Info window generally are the same as the data member names in the THINK Class Library. Data members that do not appear in the subarea are given default values when the view is run.

Changes made in the Pane Info window are reflected immediately in the target pane. For example, if you type a value in a CPane class data member's Width or Height textbox, the size of your pane in the View Edit window changes while you are typing.

A Pane Info window is specific to a pane, so you can have multiple Pane Info windows open simultaneously. When you close a view, Pane Info windows associated with panes in that view are automatically closed.

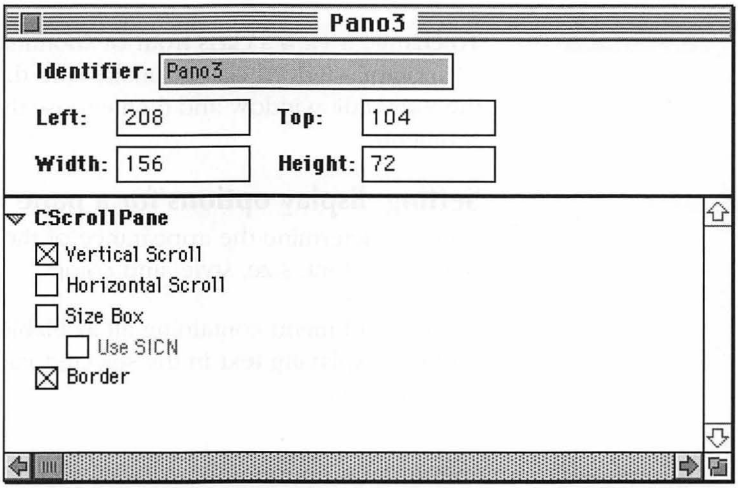
You can directly edit the text of static text, edit text, push button, radio button, or check box panes without using the **Pane Info** dialog box. Simply select the pane and press Return or Enter.

You can edit or type text in the pane, up to 32K characters. Press Return to add new lines. When you are finished editing, 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 both vertically and horizontally. Static text panes more than one line high retain their original shape. To change the shape, reselect the pane and drag the knob in the lower-right corner to the desired shape. Dialog text panes of any size retain their shape unless you extend the pane by typing more text than the box can hold. You can make a dialog text pane larger by entering text in it.

Static and dialog text panes are edited with the wholeLines attribute on (in the CAbstractText subarea), so there is no extra space at the bottom of the pane.

ScrollPane Info Opens a ScrollPane Info window for the currently selected pane in the frontmost View Edit window. If the pane is not a List/Table or a Panorama pane, it does not have an associated scroll pane. In this case, the command is disabled. The ScrollPane Info window (Figure 32-3) is similar to the Pane Info window described above.


Pano3

Identifier: Pano3

Left: 208 Top: 104

Width: 156 Height: 72

CScrollPane

☒ Vertical Scroll

□ Horizontal Scroll

□ Size Box

□ Use SICN

☒ Border

Figure 32-3 ScrollPane Info window

Only CScrollPane class data members can be edited in the ScrollPane Info window. Refer to online *THINK Reference* for definitions of these data members.

Selecting classes for a pane

You select a class for a pane by choosing from among the options presented in the **Class** submenu.

Class Opens a submenu containing the classes available either for the currently selected pane in the frontmost View Edit window or for the View Edit window itself, if no pane is selected (Figure 32-4). You can then select from the options provided.

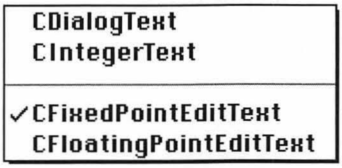


Figure 32-4 Class submenu

Classes listed above the divider are the appropriate THINK Class Library classes for the pane. Classes listed below the divider are your own classes for which the classes shown above the divider are the base classes. A checkmark, placed next to the class when you choose it from the submenu, indicates the class from which the pane is derived.

To change a view's class from CPanorama to one of your own CPanorama-derived classes, make sure that no panes are selected in the View Edit window and then choose the new class from the **Class** submenu.

Setting display options for a pane

You can determine the appearance of the text in a selected pane by selecting a font, size, style, and color.

Font

Opens a submenu containing all available fonts. You can choose a font for displaying text in the selected pane in the frontmost View Edit window.

Note

Font changes apply to push buttons, check boxes, and radio buttons, as well as to text panes.

Size

Opens a submenu containing the available sizes for the current font. You can choose a font size for displaying text in the selected pane in the frontmost View Edit window.

Style

Opens a submenu containing the available font styles. You can choose a font style for displaying text in the selected pane in the frontmost View Edit window.

Color

Opens the standard Macintosh **Color Picker** dialog box, in which you choose a color for displaying text in the selected pane in the frontmost View Edit window. Consult your Macintosh documentation for details on using this dialog box.

Arranging panes in the View Edit window

The following commands let you position panes as well as place them at the beginning or end of the drawing order.

Align

Opens the **Align** submenu (Figure 32-5). The commands on this menu let you align the selected panes in the frontmost View Edit window.

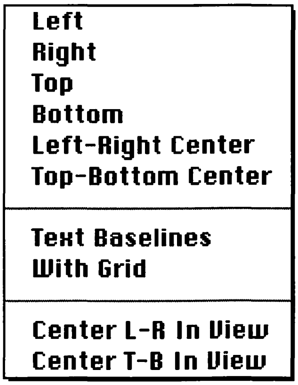


Figure 32-5 Align submenu

Left, Right, Top, Bottom

Reposition the selected panes so that their left, right, top, or bottom sides, respectively, are aligned.

Left-Right Center
Top-Bottom Center

Reposition the selected panes so that their horizontal or vertical midlines, respectively, are aligned.

Text Baselines

Repositions static text and dialog text panes so that the baselines of the first lines of text are aligned.

With Grid

Repositions the panes onto a grid whose size is defined in the **Preferences** dialog box. See Chapter 33, "Visual Architect Options Menu," for details on using a grid to position panes.

Center L-R In View
Center T-B In View

Reposition the panes so that their horizontal or vertical midpoints, respectively, are centered horizontally or vertically in the view.

Besides **Align**, the other two commands on the **Pane** menu for arranging panes are **Bring To Front** and **Send To Back**.

Bring To Front

Brings the selected pane to the end of the drawing order in the frontmost view. The drawing order of overlapping panes is the order in which they are drawn by the computer, even if the panes currently overlap as a result of dragging or using this command. The frontmost pane is thus the one that was last drawn or last dragged over the others by the user.

Send To Back

Sends the selected pane to the beginning of the drawing order in the frontmost view. See the preceding discussion of **Bring To Front** for a definition of drawing order.

Note

> The drawing order of panes affects not only their appearance in the view, but also whether they receive mouse clicks. For example, if a check box pane is behind a rectangle pane, it is not possible for the user to change the check box pane's value in the running application.

Grouping radio button panes

You can group radio buttons in views with this command.

Set Button Group

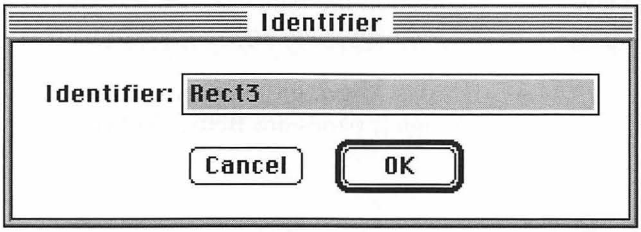
Lets you associate the selected radio button panes into a single group. Only one radio button in a group can be on at a time; clicking a button turns the currently selected button off. You can define as many button groups in a view as you want.

Changing the identifiers for panes

You can change the identifier for panes with this command.

Identifier ⌘J

Opens the **Identifier** dialog box (Figure 32-6). Here, you can change the identifier for the selected pane in the frontmost View Edit window.


Identifier

Identifier: Rect3

Figure 32-6 Identifier dialog box

Type the new identifier for the pane in the Identifier textbox and click OK. You can also change the identifier for a pane in the Pane Info window, as described in the section "Editing data members in a class hierarchy," earlier in this chapter.

Visual Architect Options Menu 33

This chapter explains the commands in the Visual Architect **Options** menu. These commands are used to customize the Visual Architect's View Editor.

Commands in the Options Menu

You use the **Options** menu commands to set the behavior of the View Editor in Visual Architect. Customization options include grid size, pane positioning, display button groups, and pane item numbers. Commands in the **Options** menu are shown in Figure 33-1.

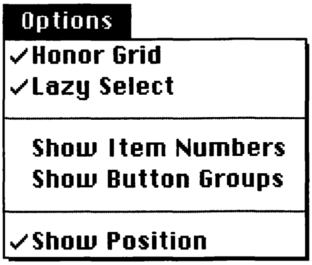


Figure 33-1 Options menu

These commands are used to perform the following functions:

- Position and select frames
- Display pane and button information

This chapter discusses the **Options** menu commands by function in the order listed above.

When you launch Visual Architect, all options are set to default settings. Most of these can be set in the **Preferences** dialog box, which is accessed by choosing **Preferences** from the **File** menu. After launch, you can use the **Options** menu to change the values from those set by default to other values suitable to the specific project.

Positioning and selecting panes

You can choose to have panes snap automatically to positions on a grid. You can also choose options for selecting panes.

Honor Grid

Forces the top-left and bottom-right coordinates of newly drawn or resized panes in the View Edit window to be even multiples of the grid step size. When the command is disabled, panes can be placed at any pixel position in the window.

You can set the grid step size to any positive power of two using the **Preferences** dialog box.

Even when **Honor Grid** is enabled, a pane can be positioned off the grid in any of the following ways:

- Creating the pane before enabling **Honor Grid**.
- Using the arrow keys with the Command key held down to move a pane in single-pixel steps.
- Entering numeric values in either the `hEncl` or the `vEncl` data member textbox of a pane in the Pane Info window. This moves the pane to that exact position. (See Chapter 32, “Visual Architect Pane Menu.”)

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

Allows you to select a pane by dragging the selection rectangle so it intersects the pane’s frame. If the command is disabled, you can select a pane only if the selection rectangle completely encloses the pane’s frame (similar to selecting in MacDraw™).

One reason to use **Lazy Select** is that a pane’s frame size is not always obvious—the frame may be larger than the drawing inside. However, it is easier to select small panes inside large panes with **Lazy Select** disabled.

Displaying pane and button information

Use these three commands to display pane and button information in the View Edit window.

Show Item Numbers

Displays the number of each pane in the drawing order in a small box in the upper-right corner of the pane's frame. If the command is disabled, pane numbers are not shown.

Panes with lower numbers are drawn earlier than panes with higher numbers. This makes them appear behind panes that are drawn later. The drawing order is the same as the tab order in Dialog views. The tab order is the order in which panes are selected when you press the Tab key to move between them.

To change a pane's drawing order number, choose the **Bring To Front** or **Move To Back** commands 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. Enabling **Show Item Numbers** disables **Show Button Groups**.

Show Button Groups

Displays the button group of each button 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. Enabling **Show Button Groups** disables **Show Item Numbers**.

Show Position

Displays the coordinates and color of the currently selected pane or panes, or the coordinates of the portRect. (The portRect is the rectangle that surrounds the portion of the View Edit window that will appear in the window of the built application. See Chapter 14, "Tutorial: Beeper," for an example.) The coordinates are shown in the Position area in the lower-left corner of the View Edit window.

Visual Architect Tools Menu

34

This reference chapter provides detailed descriptions of the tools on the Tool palette, which is displayed by clicking on the **Tools** menu in Visual Architect. You use these tools to add different types of panes to views—including text, pop-up menu, multistate button, scrollable, and graphic panes as well as subpanes.

Introducing the Tools Menu

You use the Tool palette to add panes to the view whose View Edit window is frontmost. (See Chapter 31, "Visual Architect View Menu," for information on the View Edit window.) Whenever the frontmost window is not a View Edit window, the palette is temporarily hidden (and the **Tools** menu is disabled).

By selecting a tool from the Tool palette, you can drop or draw a pane object of the corresponding type into a view. The palette is a tear-off menu, so you can drag it to another position on the screen, and it remains visible unless you close it using the close box.

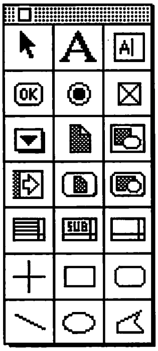


Figure 34-1 Tool palette

This chapter begins with descriptions of the palette tools, starting with the Select tool at the upper left. The following sections discuss three things: using these tools to create panes, pasting to the Clipboard, and adjusting panes.

Tool descriptions

The class of the pane each tool creates is indicated in parentheses.

Select
Choosing the Select tool changes the cursor to an arrow, the standard Macintosh selection cursor. To select a single pane, click on it. To select multiple panes, click them while holding down the Shift key. Alternatively, click an empty part of the drawing area and, holding down the mouse button, drag the cursor until the selection rectangle encompasses the panes you want to select.

As described in Chapter 33, "Visual Architect Options Menu," Visual Architect offers two selection modes: Normal and Lazy Select.

Static Text and Dialog Text
The Static Text and Dialog Text tools create static (CStaticText) and editable (CDialogText) text panes. When using the Static Text tool, click in the View Edit window to position the page. A blinking insertion point appears, letting you enter the text. When using the Dialog Text tool, click or click and drag in the View Edit window to create the pane. For both types, you can edit the pane once it has been created by selecting it and pressing Enter (Return). This changes the cursor to an I-beam, which you then click within the text to position the cursor.

Terminate text editing by either 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, as described in Chapter 32, "Visual Architect 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. You probably will have to override one or more CScrollBar member functions to create another kind of slider.

Pop-up Menu

The Pop-up Menu tool creates standard pop-up menu panes (CStdPopupPane), which are used to display menus. See Chapter 30, "Visual Architect Edit Menu," for information on using menus.

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. You can utilize any 'PICT, 'ICON', or 'cicn' resources that are in the active resource file. Place these resources into the resource file using a standard resource editor, such as ResEdit or Resorceror. .

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 they can be color-highlighted or framed. Icon and picture buttons can be configured to act as push buttons, radio buttons, or check boxes. You can display any 'PICT, 'ICON', or 'cicn' resources that are in the active resource file. Place these resources into the resource file using a standard resource editor, such as ResEdit or Resorceror.

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 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 and can contain any number of other panes.

Panorama
The Panorama tool creates generic scrollable panoramas (CPanorama). You use this tool to set aside subareas 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 in Chapter 32, "Visual Architect Pane Menu." By default, a CEditText panorama is used.

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 in Chapter 32, "Visual Architect Pane Menu." If a panorama has no scroll bars or size box, the CScrollPane is not present in the view resource or in the view at run-time.

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.

Graphic panes are not just drawings; any one can act as a button of any kind. One good way to use these graphic buttons is by drawing them over picture panes and making them invisible by setting the pen size to 0. You can choose any arbitrary size and shape for these buttons, thus making different parts of a picture clickable. By setting the `clickCmd` for shape buttons using the **Pane Info** dialog box (described in Chapter 32, "Visual Architect Pane Menu"), each different picture part can respond differently to user clicks.

Creating panes

Typically, after you use a tool once, the tool reverts to the Select tool. If you double-click a tool, however, the tool "sticks on," meaning that you can use it multiple times without reselecting. When a tool is stuck on, you unstick it by selecting another tool. Use the **Options** menu to have tools stick on after only single-clicking them. (See Chapter 33, "Visual Architect Options Menu," for details.)

Except for the Static Text tool, you create panes of arbitrary size by selecting the appropriate tool and clicking and dragging in the View Edit window. Alternatively, you can create a new pane of a default size by clicking the mouse in the window.

When you position the Static Text tool over an activated View Edit window, 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 created.

With any other tool, Visual Architect displays the crosshair cursor.

You can use the Honor Grid option to constrain the positioning of panes in the View Edit window. Enabling this option makes alignment of panes much easier. See Chapter 33, "Visual Architect Options Menu," for details.

The drawing and tab order of panes initially is set to the order in which they were created. To change the relative order of panes, choose **Bring To Front** or **Send To Back** from the **Pane** menu. To view the relative order of panes, choose **Show Item Numbers** from the **Options** menu.

Using the Clipboard

In addition to using the Tool palette, you can create static text or picture panes by pasting `PICT` or `TEXT` data from the Clipboard. See Chapter 30, "Visual Architect Edit Menu," for details.

Using arrow keys to adjust panes

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 depressed, 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.

VA: Symantec Project Manager Menu 35

Programming means generating code. The Symantec Project Manager provides powerful menus, macros, and macro development tools to help you generate code in a manner that suits your work process.

This chapter looks at the **Symantec Project Manager** menu, as well as such subjects as how to preserve manually edited code through generation cycles and how to customize the code generation macros. This chapter also contains information on the structure of generated code and the macro files used by Visual Architect for generating code.

Commands in the Symantec Project Manager Menu

The **Symantec Project Manager** menu commands let you generate your Visual Architect source code files. In addition, you can control some functions of the Symantec Project Manager from within Visual Architect using this menu.

The **Symantec Project Manager** menu in Visual Architect contains commands that directly affect the Symantec C++ project to which your Visual Architect resource file belongs. This linking of Visual Architect and the Symantec Project Manager, which occurs through Apple events, only works properly if:

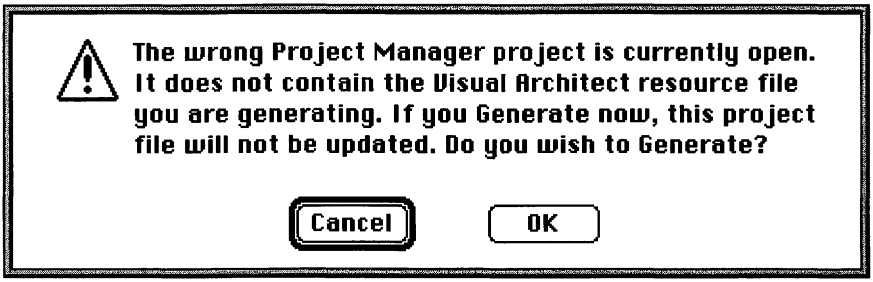
- The Update Project on Generate preference in the Visual Architect **Preferences** dialog box is set on.

- The Symantec Project Manager is running and the project to which the Visual Architect resource file belongs is open.

Note

Do not use the commands in the Visual Architect **Symantec Project Manager** menu unless these two conditions are met. If you do, your Symantec Project Manager project will not be updated properly.

If the update preference is set on but the project you need is not open, the **Update Project Warning** dialog box opens (Figure 35-1) when you choose a command from the **Symantec Project Manager** menu.



The wrong Project Manager project is currently open. It does not contain the Visual Architect resource file you are generating. If you Generate now, this project file will not be updated. Do you wish to Generate?

Figure 35-1 Update Project Warning dialog box

Click OK to proceed with code generation, though using Visual Architect in this mode is not recommended.

Figure 35-2 displays the commands on the **Symantec Project Manager** menu.

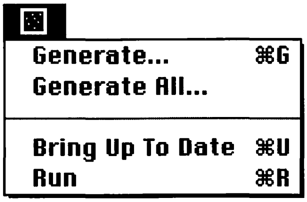


Figure 35-2 Symantec Project Manager menu

You use the commands on the Symantec Project Manager menu to perform the following functions:

- Generate code
- Update a project and run an application

This chapter covers the **Symantec Project Manager** menu commands by function in the order listed in Figure 35-2.

Generating code

Code generation in Visual Architect is controlled by a macro file. By default, Visual Architect looks for a macro file named `GenerateTCLApp` in the Symantec Project Manager tree and then in your project tree.

`GenerateTCLApp` contains macros that indicate to Visual Architect the 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.

To change the macro file Visual Architect uses to generate code, choose **Set Generate File** from the **File** menu, then select the macro file with the standard **File Open** dialog box, as described in Chapter 29, "Visual Architect File Menu."

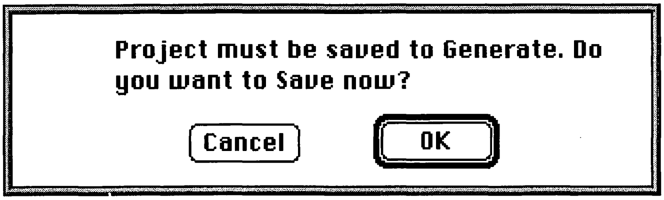
If the Update Project on Generate preference is not set in the **Preferences** dialog box or if the project that owns the resource file is not currently open in the Symantec Project Manager, you are prompted to provide the macro file to use for generating code each time you choose **Generate**.

Generate

Generates the necessary C++ code to implement the user interface elements in the resource file that owns the frontmost window. **Generate** is disabled if no changes have been made to the resource file since the last **Generate** or **Generate All**.

This command generates code only for those files that need updating as a result of changes made to the Visual Architect resource file. **Generate** does not affect generate-once files that have already been generated; see the section "Code-Generating Process," later in this chapter for information on generate-once files.

If you have set the Confirm Saves preference on in the **Preferences** dialog box and there are unsaved changes to your resource file, the **Save Now** dialog box opens (Figure 35-3) when you choose this command.

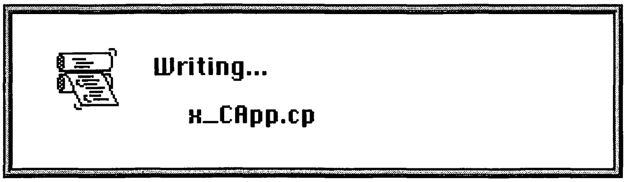


Project must be saved to Generate. Do you want to Save now?

Figure 35-3 Save Now dialog box

Click OK to have Visual Architect save the resource file and continue with the **Generate** command or click Cancel to cancel **Generate**. If the Confirm Saves preference is not set on and there are unsaved changes to your resource file, Visual Architect saves the resource file automatically, then proceeds with code generation.

While code generation is underway, you are informed of its progress in the **Code Generation Progress** message box (Figure 35-4).



Writing...

x_CApp.cp

Figure 35-4 Code Generation Progress message box

This dialog box shows the name of each file as it is being processed, as well as the action being applied to the file. To stop generation in progress, press Command-Period. Stopping generation does not remove files already generated.

If the Update Project on Generate preference is set in the **Preferences** dialog box and the project that the resource file owns is currently open in the Symantec Project Manager, the project is updated after code generation is complete. Generated files not already in the Symantec Project Manager project are added to the project, and generated files already in the project are marked for recompilation.

Generate All

Generates code for all source files, not only those that need updating as a result of changes made in Visual Architect. As with **Generate**, no code is generated for generate-once files that have already been generated. You should choose **Generate All** the first time you generate code for a resource file.

If you need to regenerate any generate-once files, remove them from the `Source` folder in your project folder, then choose **Generate All**.

Updating projects and running applications

In addition to generating code, you can use the **Symantec Project Manager** menu to update your project and run your application.

Bring Up To Date

Performs the same functions as the **Generate** command, then sends a **Bring Up To Date** command to the Symantec Project Manager. This command lets you generate source code and bring a project up-to-date without having to switch to the Symantec Project Manager.

Bring Up To Date is disabled if the Update Project on Generate preference in the Visual Architect **Preferences** dialog box is set off.

Run

Performs the same functions as the **Generate** command, then sends a **Run** command to the Symantec Project Manager. This command lets you run a project's application without having to switch to the Symantec Project Manager.

Run is disabled if the Update Project on Generate preference in the **Preferences** dialog box is set off.

Code-Generating Process

Visual Architect supplies a set of macro files capable of generating a complete THINK Class Library-based application. Before you generate code, the copyright field of the **Application Info** dialog box should be filled out and a Main view defined. To do this, select the **Application** command from the **Edit** menu.

The following sections explain the code generation strategy followed by `GenerateTCLApp` and the other macro files in the `Macros` folder supplied with Visual Architect. See the section "Generating code," earlier in this chapter.

Preserving code during regeneration

Each time you make changes to views, classes, and commands in Visual Architect, new files must be generated and 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 manually modify these files to add additional behavior.

To preserve code added 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 base class at the bottom, the lower-level class above it, and the upper-level class on top.

Structure of generated code

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 three-tier hierarchy, as shown in Figure 35-5.

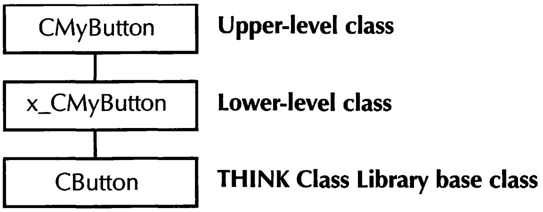


Figure 35-5 Example of a multi-level class hierarchy

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 when 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 add your own code to the generated functions. This split-level approach adds only a few bytes to program size with minimal execution overhead, but it protects the generated code from 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 `*viewname*Items.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 code from the effects of most iterative design changes, you should note the following:

- If your handwritten code uses a symbolic name for a pane, and you rename or delete the pane without making the appropriate changes in your code, an error results. Typically, the compiler warns you of this type of error.

- 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 must copy your code from the old upper-level files to the new ones.

Files generated for an application

The files in the table below are generated for every application, regardless of size. The name of a lower-level file for a derived class consists of x_ (x underscore) followed by the name of the corresponding upper-level files.

Table 35-1 Code files generated by Visual Architect

File name	Description
x_*appclass*.cp	Lower-level source code for the application class
x_*appclass*.h	Header file for the lower-level application class
appclass.cp	Upper-level source code for the application class
appclass.h	Header file for the upper-level application class
*viewname*Items.h	Header files defining symbols for panes in each view

Table 35-1 Code files generated by Visual Architect *(Continued)*

File name	Description
AppCommands.h	Header file that defines symbols for the commands defined in Visual Architect and that is regenerated each time
References.cp References.h	Files defining a ReferenceStdClasses function that force a reference to every THINK Class Library class that can do Object I/O

Inside Macro Files

This section describes how to modify and write macro generation files. If you use only the macro files supplied with Visual Architect, you can skip this section.

Macro files that drive code generation are text files. The macro files shipped with Visual Architect were created in Symantec Project Manager. You can use any text editor that can create files of the file type TEXT.

Because generated macro files are ordinary text files, they are easy to customize. To customize them, you need to know the Visual Architect macro language, which is described in the remainder of this section.

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 and expression. A statement macro begins with the first non-whitespace character on a line and occupies the entire line. The line does not appear in the output file; the macro and its line 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 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.

To use the $ character as ordinary text, you must double it, as follows:

```
CopyPString("\p$$2.99", str);
```

This line copies the Pascal string "$2.99". Note that doubling works inside C++ macros as well.

Statement macros

Each statement macro begins with a keyword immediately following the leading $, which identifies the type of statement. You must use one of the keywords listed below:

```
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 have fewer than 31 alphabetic or numeric characters. The underscore character "_" is considered an alphabetic character.

The expression can consist of any combination of variable names, constants, and operators that are permissible in an expression macro. Visual Architect evaluates the expression as if it were an expression macro and assigns the result to the macro variable. See the section "Expression macros," later in this chapter.

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. The use of `pop` statements is described at the end of this section.

Certain variables are predefined by Visual Architect to hold character strings, records, or arrays. If you define a variable with the same name as a predefined variable, that predefined variable is hidden (and is inaccessible) until the variable you defined is popped. In general, you should choose variable names that do not conflict with those listed in the section "Predefined variables," 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 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—for example, `windows`—are plural. For each array variable, the array element name (such as `window`) is singular. 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 negate 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 negate 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 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 at the end of this section.

if statement

Use an if statement to conditionally expand one or more subsequent lines. The format of the if statement is:

```
$if expression$
```

If the evaluation of the expression results in a value other than zero or 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 sometimes the two are used to prevent 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 operators that, except for automatic string-to-long coercion, are defined like their C and C++ language counterparts. These operators include:

```
+ - * / << >> ~ & | ^ && || < <= > >= == != !
```

In addition, the dot operator (`.` operator) is used to select a record item.

Expressions may be placed 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 the variables listed in the table below. For definitions of records or arrays that have record elements, see the section "Record types," later in this chapter.

Table 35-2 Predefined variables

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 application's menu bar
`classes`	An array with one class record for each class displayed in the **Classes** dialog box
`classeschanged`	`1` if **Generate All** is chosen or if any classes changed since the last generate; otherwise, `0`
`commands`	An array with one command record for each command you define in the **Commands** dialog box

Table 35-2 Predefined variables *(Continued)*

Variable name	**Predefined as**
`commandschanged`	1 if **Generate All** is chosen or if any commands changed since the last generate; otherwise, 0
`copyright`	The copyright text you specify in the **Application Info** dialog box
`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
`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`)

Table 35-2 Predefined variables *(Continued)*

Variable name	Predefined as
`time`	The current time
`views`	An array with one view record for each view you define (all window views and subviews)
`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

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 subsections define each record type separately:

App record
An app record contains application information identified in the **Application Info** dialog box.

Table 35-3 Application record description

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 type the application can create
`docType2`	The second document type the application can create
`docType3`	The third document type the application can create
`docType4`	The fourth document type the application can create
`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)

Action record
An action record describes a command action defined in the **Commands** dialog box. Each action defines a handler class, what effect the command brings about when called and, in the case of an Open action, the class of the view that is opened.

Table 35-4 Action record description

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

Table 35-4 Action record description *(Continued)*

Record element	Description
`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

Class record

A class record describes each class displayed in the **Classes** dialog box.

Table 35-5 Class record description

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 the 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

Command record

A command record describes each command you define in the **Commands** dialog box.

Table 35-6 Command record description

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

File type record

A file type record contains a single, four-character file type, for example, `TEXT`.

Table 35-7 File type record description

Record element	Description
`type`	The four-character type

Menu record

A menu record describes each menu you define in the **Menus** or **Menu Bar** dialog box.

Table 35-8 Menu record description

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

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:

```
$member.type$  $member.name$ $member.elem$;
```

Table 35-9 Member record description

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 the member to be included is generated in the 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 the member is to be included in the 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

Menu item record

A menu item record describes each menu item you define in the **Menu Items** dialog box.

Table 35-10 Menu item record description

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 35-10 Menu item record description *(Continued)*

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

Pane record

A pane record describes each pane in a view.

Table 35-11 Pane record description

Record element	Description
active	1 if pane is initially active; 0 if not
autoRefresh	1 if autoRefresh is set for the 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 box 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

Table 35-11 Pane record description *(Continued)*

Record element	Description
visible	The pane's visible value
vSizing	The pane's vSizing value
wantsClicks	The pane's wantsClicks value
width	The pane's width value

Point record

A point record contains a point.

Table 35-12 Point record description

Record element	Description
h	Horizontal coordinate
v	Vertical coordinate

Rectangle record

A rectangle record contains a rectangle.

Table 35-13 Rectangle record description

Record element	Description
bottom	Lower vertical coordinate
left	Upper horizontal coordinate
right	Lower horizontal coordinate
top	Upper vertical coordinate

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, not subviews.

Table 35-14 View record description

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 box
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 35-14 View record description *(Continued)*

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 35-14 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)

Symantec C++

Appendixes

Part Six

A Linker Error Messages

B Debugger Error Messages

Linker Error Messages

A

This appendix is a guide to the error messages that can be generated by the Symantec Linker. Error messages that can occur when using the Debugger are documented in Appendix B, "Debugger Error Messages." Error messages generated by Symantec C, C++, or Rez are documented in the *Symantec C++ Compiler Guide.*

An error message typically consists of the filename and line number followed by a description of the error:

```
File "Sillyballs.cp"; Line 86
Error:   function 'NexBall__F' has no prototype
```

The following Linker error messages are arranged in alphabetic order.

Entry point (main) not found
You have not defined a `main()` anywhere in your project. Make sure that you have included the PPCRuntime library in your project and have defined a `main()` in your project.

Error creating or writing output file
There was a system error with the output file. Make sure that you have enough disk space and you have write access to the target folder.

Error opening or processing input file
There was a system error with the input file. Make sure that the file was created without errors and is not corrupt.

Internal error
There is a bug in the Linker. Report this to Technical Support.

Invalid input file format
There is an error in the format of the input file. Make sure that it is the correct file type, was created without any errors, and is not corrupt.

Invalid parameter
There is a bug in the Linker. Report this to Technical Support.

Linker database version doesn't match code
You are using an old project. This project should have been converted automatically. Report this to Technical Support.

Memory error
There is not enough memory to continue. Increase the size of the partition for the Symantec Project Manager or quit other applications.

Module not found
This is an internal return code used by the Debugger.

Multiply defined symbol: x (file 1,...,file n)
A function, class object, or global data item is defined in more than one source file or library.

Project database error
The Project file is corrupt. Try choosing **Remove Objects**, then **Build Project** in the **Build** menu to rebuild. Report this to Technical Support.

Recovery error
The Linker discovered some corruption of its database while attempting to recover from an error (usually a memory error). Try choosing **Remove Objects**, then **Build Project** in the **Build** menu to rebuild. You also may want to increase the partition size of the Symantec Project Manager before rebuilding.

Required TOC is larger than 64K bytes
The application or shared library has too many functions or global variables to fit in the TOC. This is a limitation of the PowerPC architecture. Split the application or shared library into shared libraries.

Routine not found at address
This is an internal return code used by the Debugger.

Segment not found
This is an internal return code used by the Debugger.

Shared library has no 'cfrg'
A shared library in your project lacks an essential resource. Make sure that you are linking to a properly constructed shared library.

Symbol not found
This could be the result of a corrupt Linker database. Try choosing **Remove Objects**, then **Build Project** in the **Build** menu to rebuild. Report this to Technical Support.

Undefined symbol: x (file 1,...,file n)
There is an undefined symbol. A function, class object, or global data item is missing. Make sure that the project has all the necessary libraries. If it does, either there is a missing source file or the program is incomplete and the missing symbol should be defined. The name of the missing symbol is a good clue, that is, `missing symbol 'printf'` means `PPCANSI.o` is not in the project.

Debugger Error Messages ◆ B

This appendix is a guide to the error messages that can occur when using the Symantec Debugger. Error messages generated by the Linker are documented in Appendix A, "Linker Error Messages." Error messages generated by Symantec C, C++, or Rez are documented in the *Symantec C++ Compiler Guide.*

The Debugger can report errors in three different places.

- In the Data pane, errors may result from entering an expression.

  Messages that appear in the Data pane appear bracketed by two hollow diamonds ◊◊ on each side:

  ```
  ◊◊ expression expected ◊◊
  ```

- In the Control palette, errors may result when an exception occurs.

  The Debugger uses the Control palette for exception messages; if the Control palette is closed, then it will use a dialog box. For more information about machine exceptions, see the description of exceptions in the manual for the 601 processor (Motorola press document MPC601UM/AD).

- In a modal dialog box, errors may result when an exception occurs.

The messages that occur while debugging can be generated by the compiler or Linker, or they can be exception errors.

The following Debugger error messages are arranged in alphabetic order.

Exception access fault
Your program attempted to write to or read from an address that doesn't exist or is in a protected memory space.

Floating point exception
Your program attempted to execute an invalid floating point operation when floating point exceptions are enabled. Under the Macintosh operating system, they are disabled by default.

Illegal instruction
Your program attempted to execute an unknown instruction. This can happen if you try to call a member function of an invalid C++ object.

Note

> Unlike the 68000 microprocessor, the 601 microprocessor doesn't report a divide by zero error.

Index ◆

Entries in **boldface** are menu commands or dialog boxes. Entries in `typewriter face` are functions, methods, variables, keywords, or files.

Symantec Service and Support Solutions

Symantec offers a variety of technical support and customer services to meet your needs. Our Technical Support department offers several different levels of support to assist you with specific questions you may have in using our software. Our Customer Service department will inform you what Symantec has to offer and how to get what's available.

Registering Your Symantec Product

To register your Symantec product, complete the registration card included with your package and drop the card in the mail. You can also register via your modem during the installation process if your Symantec software offers this feature. In addition, you can use the toll-free fax number listed below to register your product.

If your address changes, you can mail or fax your new address to Customer Service. Please send it to the attention of the Registration Department.

Symantec Corporation
Attn: Registration Dept.
175 W. Broadway
Eugene, OR 97401

(800) 800-1438 Fax

Technical Support

Symantec's Technical Support department offers expanded support options designed for your individual needs and to help you get the most out of your software investment.

The phone numbers listed on the back of this manual are for support in North America. If you are outside of the United States or Canada, please call the local Symantec office or distributor in your area, or refer to the international offices provided at the end of this section.

Symantec now offers different types of technical support services for you to choose from, which are described below. You are given StandardCare Support by purchasing the product and then can choose from Symantec's PriorityCare and PremiumCare services to extend your level of support.

StandardCare Support

All registered users of Symantec products are entitled to these services at no charge:

- Unlimited calls for 90 days (from the date of the first call) for installation assistance, configuration, and general usage questions.

- Unlimited technical assistance via CompuServe and America Online. These forums offer electronic access to our technical support staff, libraries of sample files, technical notes, and bulletins. You will also find a rich interaction and information exchange with other users of Symantec software.

- Unlimited use of Symantec's Bulletin Board System (BBS). This download BBS is kept updated with sample files and product technical notes for quick and easy electronic access.

- Unlimited access to company information via the Internet. With an Internet browser program such as Mosaic, Cello, or Netscape, you get the latest company news by entering Symantec's Internet address: HTTP://WWW.SYMANTEC.COM.

- Unlimited use of Symantec's automated fax retrieval system for instant printouts of technical notes, bulletins, product literature, and general information by fax.

- StandardCare Support is available Monday through Friday, 7:00 a.m. to 4:00 p.m. Pacific Time.

For your first 90 days of free technical support, refer to the (503) phone number on the back of this manual.

PriorityCare Support

All registered users of Symantec products are entitled to these services on a "pay-as-you-go" basis:

- The PriorityCare 800-number is charged to your VISA, MasterCard, or American Express on a per incident basis.
- The PriorityCare 900-number is charged to your telephone bill on a per minute basis. (As of this writing, the equivalent 900-number service is not available in Canada.)
- Average hold time will be kept to a minimum.
- PriorityCare Support is available Monday through Friday, 6:00 a.m. to 5:00 p.m. Pacific Time.

To use the PriorityCare 800- and 900-number services, refer to those numbers on the back of this manual.

PremiumCare Support

All registered users of Symantec products are entitled to these services on an annual subscription basis:

PremiumCare Gold Support provides:

- Unlimited calls on a toll-free 800 line.
- Average hold time will be kept to a minimum.
- PremiumCare Gold Support is charged on an annual subscription basis.
- PremiumCare Gold Support is available Monday through Friday,
  6:00 a.m. to 5:00 p.m. Pacific Time.

PremiumCare Platinum Support provides:

- Unlimited calls on a toll-free 800 line.
- Average hold time will be kept to a minimum.
- A Support Center Manual with troubleshooting, installation, configuration, and usage information.
- Quarterly updates of technical notes and bulletins.

- Instant access to senior support staff.
- Automatic updates of inline software revisions. (Inline software revisions do not include version upgrades.)
- After hours and weekend support is also available to PremiumCare Platinum customers for an additional fee.
- PremiumCare Platinum Support is charged on an annual subscription basis per product family. The annual fee is for two subscribers; other subscribers can be added on a per person basis.
- PremiumCare Platinum Support is available Monday through Friday, 6:00 a.m. to 6:00 p.m. Pacific Time.

To order PremiumCare Gold or Platinum support, please contact Customer Service or your Symantec sales representative.

Electronic Support

Technical information is available 24 hours a day on electronic bulletin board systems. Symantec provides access to its own Symantec bulletin board system (BBS), and maintains the Symantec forums on CompuServe, America Online, and Applelink.

Symantec BBS

The Symantec BBS provides a Customer Service forum, shareware and public-domain software, "Frequently Asked Questions" (FAQs), and support forums where you can exchange tips and information with other end users. Settings for the Symantec bulletin board are: 8 data bits, 1 stop bit; no parity.

300-, 1200-, and 2400-baud modems	(503) 484-6699 (24 hrs.)
9600-, and 14,400-baud modems	(503) 484-6669 (24 hrs.)

CompuServe

To access the Symantec forums on CompuServe, type:

`GO SYMANTEC` at any ! prompt.

For additional information, or to subscribe in the United States and Canada, call CompuServe at (800) 848-8199. Check with CompuServe for data communications settings.

America Online

To access the Symantec bulletin board on America Online, type keyword:

```
SYMANTEC
```

For additional information, or to subscribe in the United States and Canada, call America Online at (800) 227-6364. Check with America Online for data communications settings.

In other regions, contact them directly for information on how to obtain an account.

Applelink

You can exchange information and ideas with other users of Symantec products on the Applelink bulletin board. You can also ask technical support questions and report bugs. To send us email, use the address: d0152.

You can download product updates from the software library. Our forum is located in "Third Parties:Third Parties (P-Z):Symantec Solutions".

Internet

To ask technical support questions or to report a possible bug, send email to:

```
support@devtools.symantec.com
```

Automated Fax Retrieval System

Symantec's automated fax retrieval system can be used 24 hours a day to receive product information on your fax machine. You can call from any touch tone phone to receive an index listing of both Technical Support and Customer Service documents available, then have any of these specific documents faxed to you.

To receive technical application notes and samples of "how tos," call our Technical Support fax retrieval number.

You can receive general product information, data sheets, and product upgrade order forms from our Customer Service fax retrieval number.

Technical Support:	(503) 984-2490
Customer Service:	(800) 554-4403

Customer Service

Symantec's Customer Service department builds and maintains long-lasting customer relations through consistent, expert service. Our Customer Service department is available to help you:

- Order an upgrade.
- Subscribe to the technical support solution of your choice.
- Fulfill your request for product literature or demonstration disks.
- Find out about dealers and consultants in your area.
- Replace missing or defective pieces (disks, manuals, etc.) from your package.
- Let us know if the address on your registration card has changed.

Replacing a CD-ROM

If you need to replace a defective CD-ROM that is still under warranty, please use the CD-ROM Replacement form at the end of this manual, or contact Customer Service.

To receive a refund for a product purchased through a reseller, please visit or contact the authorized dealer from whom you purchased the product.

If you ordered the product or upgrade directly from Symantec and wish to receive a refund in accordance with our own 60-day money back guarantee, please contact our Customer Service department within 60 days of purchase, at 1-800-441-7234. One of our Customer Service Representatives will be happy to give you a return authorization number and instructions for returning the product.

For specific questions about how to use your Symantec software, please contact Technical Support.

Customer Service Locations

USA	Symantec C orporation	(800) 441-7234 (USA & Canada)
	175 W. Broadway	(503) 334-6054 (all other locations)
	Eugene, OR 97401	Fax (503) 334-7400

International Locations

Symantec provides technical support and customer service worldwide. If you're in a country outside of the United States or Canada, please contact the local distributor or Symantec office nearest to you, or our world headquarters.

World Headquarters	
Symantec Corporation	Tel. 1 (408) 253-9600
10201 Torre Avenue	
Cupertino, CA 95014	
U.S.A.	
European Headquarters	
Symantec Europe	Tel. (31) (71) 353 111
Kanaalpark 145	Fax (31) (71) 353 150
2321 JV Leiden	
The Netherlands	
Technical Support - Dutch	Tel. (31) (71) 353 184
French	Tel. (31) (71) 353 180
German	Tel. (31) (71) 353 181
English	Tel. (31) (71) 353 182 Fax (31) (71) 353 153
BBS	Tel. (31) (71) 353 169
Automated fax retrieval	Tel. (31) (71) 353 255
Australia	
Symantec Australia Pty. Ltd.	Tel. (61) (2) 879 6577
408 Victoria Road	Fax (61) (2) 879 6805

Gladesville, NSW 2111
Australia

Technical Support	Tel. (61) (2) 879 6577
	Fax (61) (2) 879 6594
BBS	Tel. (61) (2) 879 6322
DOS/Win Antivirus recording	Tel. (61) (2) 879 7362
Mac Antivirus recording	Tel. (61) (2) 879 6968

Symantec C++ 8.0 for Power Macintosh
CD-ROM Replacement

CD-ROM REPLACEMENT: After your 60-Day Limited Warranty, if your CD-ROM becomes unusable, fill out Sections A & B and return 1) this form, 2) your damaged CD-ROM, and 3) your payment (see pricing below, add sales tax if applicable), to the address below to receive a replacement CD-ROM. *DURING THE 60-DAY LIMITED WARRANTY PERIOD, THIS SERVICE IS FREE.* You must be a registered customer in order to receive a CD-ROM replacement.

SECTION A – CUSTOMER INFORMATION

Name ______________________________

Company Name ______________________________

Street Address *(No P.O. Boxes, Please)*______________________________

City ______________________________ State ______Zip/Postal Code______________

Country* ______________________________ Daytime Phone______________

Software Purchase Date ______________________ Version ______________

*This offer limited to U.S. and Canada. Outside North America, contact your local Symantec office or distributor.

SECTION B – CD-ROM INFORMATION

Briefly Describe the Problem: ______________________________

____

CD-ROM Replacement Price	$ 5.00
Sales Tax (See Table)	$ _____
Shipping & Handling	$ 9.95
TOTAL DUE	$ _____

Please add sales/use tax for the following states: AZ, CA, CO, CT, DC, FL, GA, IL, IN, IA, KS, LA, ME, MD, MA, MI, MN, MO, NC, NJ, NY, OH, PA, SC, TN, TX, VA, WA, WI, and Canada (GST)

FORM OF PAYMENT** (Check One)

❑ Check (Payable to Symantec) Amount Enclosed $__________ ❑ Visa ❑ MasterCard ❑ American Express

Credit Card Number ______________________ Expires ______________

Name on Card *(Please Print)* ______________ Signature ______________

**U.S. Dollars. Payment must be made in U.S. dollars drawn on a U.S. bank.

MAIL YOUR CD-ROM REPLACEMENT ORDER TO:

Symantec Corporation
Attention: CD-ROM Replacement
P.O BOX 10849
Eugene, OR 97440-2849

Please allow 2-3 weeks for delivery.

SYMANTEC.

SYMANTEC.

Technical Support

For specific technical questions about Symantec C++,
please call our technical experts by choosing one of the three support options below.
For information on Symantec's broad range of service and support programs,
see the Service and Support Solutions section in this manual.

StandardCare Support
503-465-8470 (No charge for 90 days from date of first call.)

PriorityCare 800 or PremiumCare 800 Support
800-927-4014 (Charged on a per-incident or per-year basis.)

PriorityCare 900 Service
900-646-0004 (Charged on a per-minute or per-incident basis.)

Customer Service

For general questions about Symantec products, please call
800-441-7234 (U.S. and Canada) or 503-334-6054.

Symantec Corporation Headquarters
10201 Torre Avenue
Cupertino, California 95014
408-253-9600

Printed in the U.S.A. 436-0517-12 1/95 119493JR 4/95

Made from 10
Recycled Mater