Figure Out the Basics and Start Programming in No Time!

50 MILLION DUMMIES BOOKS IN PRINT

MAC® PROGRAMMING FOR DUMMIES®

3RD EDITION

Special Coverage of Movie Playing and Other Advanced Programming Topics!

A Reference for
he Rest of Us!®

y Dan Parks Sydow

BENJAMIN FRANKLIN AWARD Publishers Marketing Association

1997 READERS' CHOICE AWARD COMPUTER CURRENTS

The Fun and Easy Way™ to Create New Mac Programs

Your First Aid Kit® for Compiling Source Code with CodeWarrior Lite

Complexities of C — Explained in Plain English

Mac® Programming For Dummies®, 3rd Edition

Cheat Sheet

C Language Data Types

Use	Data Type	Example
Small whole number	`short`	`short days = 3;`
Big whole number	`long`	`long population = 5600200;`
Decimal numbers	`float`	`float seconds = 42.53;`
Event record	`EventRecord`	`EventRecord theEvent;`
Window	`WindowPtr`	`WindowPtr theWindow;`
Rectangle	`Rect`	`Rect smallSquare;`

C Language Math

In the following examples, assume that all variables are declared to be of type `short`.

Operation	Symbol	Example
Addition	+	`newAge = currentAge + 1;`
Subtraction	-	`loss = total - damaged;`
Multiplication	*	`days = weeks * 7;`
Division	/	`years = months / 12;`
Increment by 1	++	`loopCounter++;`

Important Toolbox Functions

In the following examples, assume that the example variables are declared as follows: `theEvent` is of type `EventRecord`, `theWindow` is of type `WindowPtr`, and `theRect` is of type `Rect`.

Task	Toolbox Function	Example
Get event information	`WaitNextEvent`	`WaitNextEvent(everyEvent, &theEvent, 0L, 0L);`
Open a window	`GetNewWindow`	`GetNewWindow(128, nil, (WindowPtr)-1L);`
Ready window for drawing	`SetPort`	`SetPort(theWindow);`
Location to start drawing	`MoveTo`	`MoveTo(20, 50);`
Draw a line of text	`DrawString`	`DrawString("\pThis is a test.");`
Draw a line	`Line`	`Line(100, 0);`
Set up a rectangle	`SetRect`	`SetRect(&theRect, 10, 10, 60, 80);`
Draw a rectangle	`FrameRect`	`FrameRect(&theRect);`
Beeping the speaker	`SysBeep`	`SysBeep(1);`

Mac® Programming For Dummies®, 3rd Edition

Cheat Sheet

C Language Basics

In the following examples, assume that the variable `count` is declared to be of type `short`.

Task	*Syntax*	*Example*
Comment	`/* */`	`/* this is a comment */`
Loop	`while`	`while ( count < 5 )`
		`{`
		`count++`
		`}`
Branch	`switch`	`switch ( count )`
		`{`
		`case 1:`
		`DrawString("\pYou have 1.");`
		`break;`
		`}`
Branch	`if`	`if ( count < 2 )`
		`{`
		`DrawString("\pLess than 2.");`
		`}`

Creating a New Program Using CodeWarrior

Task	*How to Carry Out*
Create a CodeWarrior project file	Choose New Project from File menu
Create a resource file	Choose New from ResEdit's File menu
Add resource file to project	Choose Add Files from Project menu
Remove resource placeholder	Click the placeholder file and then choose Remove Files from Project menu
Create new source code file	Choose New from File menu
Save source code file	Choose Save As from File menu
Add source code file to project	Choose Add Window from Project menu
Remove source code placeholder	Click the placeholder file and then choose Remove Files from Project menu
Write source code	That's up to you!
Run the code	Choose Run from the Project menu

Cheat Sheet $2.95 value. Item 0544-0.
For more information about IDG Books,
call 1-800-762-2974.

MAC® PROGRAMMING FOR DUMMIES®

3RD EDITION

MAC® PROGRAMMING FOR DUMMIES®

3RD EDITION

by Dan Parks Sydow

IDG
BOOKS
WORLDWIDE

IDG Books Worldwide, Inc.
An International Data Group Company

Foster City, CA ♦ Chicago, IL ♦ Indianapolis, IN ♦ New York, NY

Mac® Programming For Dummies®, 3rd Edition

Published by
IDG Books Worldwide, Inc.
An International Data Group Company
919 E. Hillsdale Blvd.
Suite 400
Foster City, CA 94404
`www.idgbooks.com` (IDG Books Worldwide Web site)
`www.dummies.com` (Dummies Press Web site)

Library of Congress Catalog Card No.: 99-61115

ISBN: 0-7645-0544-0

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

3O/RW/QT/ZZ/IN

Distributed in the United States by IDG Books Worldwide, Inc.

Distributed by CDG Books Canada Inc. for Canada; by Transworld Publishers Limited in the United Kingdom; by IDG Norge Books for Norway; by IDG Sweden Books for Sweden; by Woodslane Pty. Ltd. for Australia; by Woodslane (NZ) Ltd. for New Zealand; by TransQuest Publishers Pte Ltd. for Singapore, Malaysia, Thailand, Indonesia, and Hong Kong; by ICG Muse, Inc. for Japan; by Norma Comunicaciones S.A. for Colombia; by Intersoft for South Africa; by Le Monde en Tique for France; by International Thomson Publishing for Germany, Austria and Switzerland; by Distribuidora Cuspide for Argentina; by Livraria Cultura for Brazil; by Ediciones ZETA S.C.R. Ltda. for Peru; by WS Computer Publishing Corporation, Inc., for the Philippines; by Contemporanea de Ediciones for Venezuela; by Express Computer Distributors for the Caribbean and West Indies; by Micronesia Media Distributor, Inc. for Micronesia; by Grupo Editorial Norma S.A. for Guatemala; by Chips Computadoras S.A. de C.V. for Mexico; by Editorial Norma de Panama S.A. for Panama; by American Bookshops for Finland. Authorized Sales Agent: Anthony Rudkin Associates for the Middle East and North Africa.

For general information on IDG Books Worldwide's books in the U.S., please call our Consumer Customer Service department at 800-762-2974. For reseller information, including discounts and premium sales, please call our Reseller Customer Service department at 800-434-3422.

For information on where to purchase IDG Books Worldwide's books outside the U.S., please contact our International Sales department at 317-596-5530 or fax 317-596-5692.

For consumer information on foreign language translations, please contact our Customer Service department at 1-800-434-3422, fax 317-596-5692, or e-mail rights@idgbooks.com.

For information on licensing foreign or domestic rights, please phone +1-650-655-3109.

For sales inquiries and special prices for bulk quantities, please contact our Sales department at 650-655-3200 or write to the address above.

For information on using IDG Books Worldwide's books in the classroom or for ordering examination copies, please contact our Educational Sales department at 800-434-2086 or fax 317-596-5499.

For press review copies, author interviews, or other publicity information, please contact our Public Relations department at 650-655-3000 or fax 650-655-3299.

For authorization to photocopy items for corporate, personal, or educational use, please contact Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, or fax 978-750-4470.

<u>LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY</u>: THE PUBLISHER AND AUTHOR HAVE USED THEIR BEST EFFORTS IN PREPARING THIS BOOK. THE PUBLISHER AND AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK AND SPECIFICALLY DISCLAIM ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THERE ARE NO WARRANTIES WHICH EXTEND BEYOND THE DESCRIPTIONS CONTAINED IN THIS PARAGRAPH. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES OR WRITTEN SALES MATERIALS. THE ACCURACY AND COMPLETENESS OF THE INFORMATION PROVIDED HEREIN AND THE OPINIONS STATED HEREIN ARE NOT GUARANTEED OR WARRANTED TO PRODUCE ANY PARTICULAR RESULTS, AND THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY INDIVIDUAL. NEITHER THE PUBLISHER NOR AUTHOR SHALL BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES. FULFILLMENT OF EACH COUPON OFFER IS THE RESPONSIBILITY OF THE OFFEROR.

Trademarks: All brand names and product names used in this book are trade names, service marks, trademarks, or registered trademarks of their respective owners. IDG Books Worldwide is not associated with any product or vendor mentioned in this book.

IDG BOOKS WORLDWIDE is a registered trademark or trademark under exclusive license to IDG Books Worldwide, Inc. from International Data Group, Inc. in the United States and/or other countries.

About the Author

Dan Parks Sydow is a graduate of Milwaukee School of Engineering, with a degree in Computer Engineering. He has worked on software in several diverse areas, including the display of images of the heart for medical purposes. Because Dan can't stand the sight of blood — even electronic blood — he quit his nine-to-five job as a software engineer to become a freelance writer.

Dan got hooked on Macintosh programming a decade and a half ago when the Macintosh was first introduced. Since then, he has spared no effort in avoiding all other types of computers. He enjoys shedding light on topics that people have been led to believe were beyond their reach, and he has written over a dozen Macintosh programming books.

ABOUT IDG BOOKS WORLDWIDE

Welcome to the world of IDG Books Worldwide.

IDG Books Worldwide, Inc., is a subsidiary of International Data Group, the world's largest publisher of computer-related information and the leading global provider of information services on information technology. IDG was founded more than 30 years ago by Patrick J. McGovern and now employs more than 9,000 people worldwide. IDG publishes more than 290 computer publications in over 75 countries. More than 90 million people read one or more IDG publications each month.

Launched in 1990, IDG Books Worldwide is today the #1 publisher of best-selling computer books in the United States. We are proud to have received eight awards from the Computer Press Association in recognition of editorial excellence and three from Computer Currents' First Annual Readers' Choice Awards. Our best-selling *...For Dummies®* series has more than 50 million copies in print with translations in 31 languages. IDG Books Worldwide, through a joint venture with IDG's Hi-Tech Beijing, became the first U.S. publisher to publish a computer book in the People's Republic of China. In record time, IDG Books Worldwide has become the first choice for millions of readers around the world who want to learn how to better manage their businesses.

Our mission is simple: Every one of our books is designed to bring extra value and skill-building instructions to the reader. Our books are written by experts who understand and care about our readers. The knowledge base of our editorial staff comes from years of experience in publishing, education, and journalism — experience we use to produce books to carry us into the new millennium. In short, we care about books, so we attract the best people. We devote special attention to details such as audience, interior design, use of icons, and illustrations. And because we use an efficient process of authoring, editing, and desktop publishing our books electronically, we can spend more time ensuring superior content and less time on the technicalities of making books.

You can count on our commitment to deliver high-quality books at competitive prices on topics you want to read about. At IDG Books Worldwide, we continue in the IDG tradition of delivering quality for more than 30 years. You'll find no better book on a subject than one from IDG Books Worldwide.

IDG BOOKS WORLDWIDE

John Kilcullen
Chairman and CEO
IDG Books Worldwide, Inc.

Steven Berkowitz
President and Publisher
IDG Books Worldwide, Inc.

VIII WINNER
Eighth Annual Computer Press Awards 1992

IX WINNER
Ninth Annual Computer Press Awards 1993

X WINNER
Tenth Annual Computer Press Awards 1994

XI WINNER
Eleventh Annual Computer Press Awards 1995

IDG is the world's leading IT media, research and exposition company. Founded in 1964, IDG had 1997 revenues of $2.05 billion and has more than 9,000 employees worldwide. IDG offers the widest range of media options that reach IT buyers in 75 countries representing 95% of worldwide IT spending. IDG's diverse product and services portfolio spans six key areas including print publishing, online publishing, expositions and conferences, market research, education and training, and global marketing services. More than 90 million people read one or more of IDG's 290 magazines and newspapers, including IDG's leading global brands — Computerworld, PC World, Network World, Macworld and the Channel World family of publications. IDG Books Worldwide is one of the fastest-growing computer book publishers in the world, with more than 700 titles in 36 languages. The "...For Dummies®" series alone has more than 50 million copies in print. IDG offers online users the largest network of technology-specific Web sites around the world through IDG.net (http://www.idg.net), which comprises more than 225 targeted Web sites in 55 countries worldwide. International Data Corporation (IDC) is the world's largest provider of information technology data, analysis and consulting, with research centers in over 41 countries and more than 400 research analysts worldwide. IDG World Expo is a leading producer of more than 168 globally branded conferences and expositions in 35 countries including E3 (Electronic Entertainment Expo), Macworld Expo, ComNet, Windows World Expo, ICE (Internet Commerce Expo), Agenda, DEMO, and Spotlight. IDG's training subsidiary, ExecuTrain, is the world's largest computer training company, with more than 230 locations worldwide and 785 training courses. IDG Marketing Services helps industry-leading IT companies build international brand recognition by developing global integrated marketing programs via IDG's print, online and exposition products worldwide. Further information about the company can be found at www.idg.com.

1/24/99

Dedication

To my wife, Nadine.

Author's Acknowledgments

Many thanks to Carole McClendon and Steven Hayes for making this book happen; to Colleen Esterline for the suggestions, comments, and corrections that made my writing suitable to print; to Kelley Baker for her work on getting CodeWarrior Lite on the CD-ROM; and to Dennis Cohen for another helpful technical edit.

Publisher's Acknowledgments

We're proud of this book; please register your comments through our IDG Books Worldwide Online Registration Form located at `http://my2cents.dummies.com`.

Some of the people who helped bring this book to market include the following:

Acquisitions, Editorial, and Media Development

Project Editor: Colleen Esterline *(Previous Edition: Mary Goodwin)*

Acquisitions Editor: Steven H. Hayes

Technical Editor: Dennis Cohen

Media Development Editor: Marita Ellixson

Associate Permissions Editor: Carmen Krikorian

Media Development Coordinator: Megan Roney

Editorial Manager: Mary C. Corder

Media Development Manager: Heather Heath Dismore

Editorial Assistants: Paul Kuzmic, Beth Parlon, Alison Walthall

Production

Project Coordinators: Valery Bourke, Regina Snyder

Layout and Graphics: Linda M. Boyer, J. Tyler Connor, Angela F. Hunckler, Anna Rohrer, Brent Savage, Brian Torwelle

Proofreaders: Kelli Botta, Jennifer Mahern, Arielle Carole Mennelle, Nancy Price, Ethel M. Winslow, Janet M. Withers

Indexer: Ty Koontz

Special Help

David Mehring

General and Administrative

IDG Books Worldwide, Inc.: John Kilcullen, CEO; Steven Berkowitz, President and Publisher

IDG Books Technology Publishing: Brenda McLaughlin, Senior Vice President and Group Publisher

Dummies Technology Press and Dummies Editorial: Diane Graves Steele, Vice President and Associate Publisher; Mary Bednarek, Director of Acquisitions and Product Development; Kristin A. Cocks, Editorial Director

Dummies Trade Press: Kathleen A. Welton, Vice President and Publisher; Kevin Thornton, Acquisitions Manager

IDG Books Production for Dummies Press: Michael R. Britton, Vice President of Production and Creative Services; Cindy L. Phipps, Manager of Project Coordination, Production Proofreading, and Indexing; Kathie S. Schutte, Supervisor of Page Layout; Shelley Lea, Supervisor of Graphics and Design; Debbie J. Gates, Production Systems Specialist; Robert Springer, Supervisor of Proofreading; Debbie Stailey, Special Projects Coordinator; Tony Augsburger, Supervisor of Reprints and Bluelines

Dummies Packaging and Book Design: Patty Page, Manager, Promotions Marketing

♦

The publisher would like to give special thanks to Patrick J. McGovern, without whom this book would not have been possible.

♦

Contents at a Glance

Cartoons at a Glance

By Rich Tennant

page 59

The 5th Wave
By Rich Tennant
AT THE REAL PROGRAMMERS DATING BAR
WHOA! LOOK AT THE POCKET PROTECTORS ON THIS ONE!

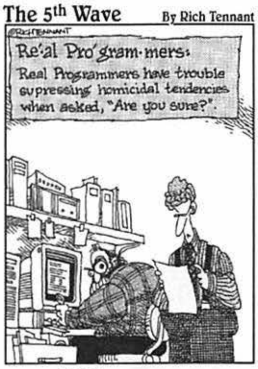
page 9

page 145

The 5th Wave
By Rich Tennant
ROCKY XI - ROCKY BALBOA BECOMES A Real Programmer
GIVE IT UP, ROCK!! THIS VIRUS IS TOO TOUGH TO CRACK! YOU BEEN WORKIN' AT IT SO LONG YOUR EYES HAVE SWOLLEN SHUT!
CUT ME, MICK. CUT ME.

page 203

page 335


The 5th Wave
By Rich Tennant
Re'al Pro'gram·mers:
Real Programmers have trouble supressing homicidal tendencies when asked, "Are you sure?".

page 103

The 5th Wave
By Rich Tennant
All through High School he wouldn't talk to anyone - hardly said a word. Now he's graduating from an Ivy League college with an advanced degree in communications.

page 315

Fax:* 978-546-7747 • *E-mail: the5wave@tiac.net

Table of Contents

Introduction

Macintosh computers are easy and fun to use. Even people who have never used a computer before find that within just a couple of hours they feel comfortable using a Mac to type a letter or draw a simple picture. But what about programming a Macintosh — is that also easy and fun? Not always. Don't worry though; programming a Mac doesn't have to be nearly as painful as you may have been led to believe.

You picked up this book for one of several reasons. Maybe you've programmed a computer, but not a Macintosh. Or you've never programmed, but are curious enough to try your hand at it. Or you just happen to be the type that enjoys the thought of pain. Now, now — I want to end that fallacy right here. Contrary to popular belief, no ordinance or law exists that states that the words *programming* and *fate-worse-than-death* must always appear together.

Why is programming thought of as aggravating, time-consuming, detail-oriented, and difficult? Because it can be — as it is practiced by professional software engineers. What about people who, instead of aspiring to become professional programmers, just happen to enjoy a good challenge? If you are one of those people, then I have good news. And this book is it.

Programming can be fun, interesting, and rewarding, and it doesn't have to be difficult. Just learning the very basics of Macintosh programming enables you to write a program that uses windows, menus, and graphics. You don't need a degree in computer engineering or a technical reference book the size of the New York City telephone directory to write a basic Mac program. Instead, you need a book that skips the technical details and theory. You need a book that presents you with just the essentials necessary to write a program that runs on a Macintosh. You need a book with a bright yellow cover and a title along the lines of *Mac Programming For Dummies,* 3rd Edition.

Why Program on the Mac?

Professional programmers get *paid* to write computer code. So why should you do it for *free*? Because you picked up this book, I have a feeling you may already know why you want to try your hand at programming the Mac. If it was simple curiosity that led you to this point, however, you may need a little convincing that you should want to take on this endeavor. The following list should contain at least one or two items that persuade you. You may want to learn to program the Mac so that you can:

- ✓ Take on — and conquer — a challenge.
- ✓ Understand what the heck your programmer and engineer friends and coworkers are always talking about.
- ✓ Gain an insight into how real programs, such as word processors or drawing programs, work.
- ✓ Write a simple game tailored specifically for your children.
- ✓ Take the first step to making a living as a computer programmer.
- ✓ Make a quick transition to Mac programming if you already know how to program other systems, such as Windows or UNIX.
- ✓ Learn about programming with graphics, menus, and windows if you already know how to write simple text-only programs.
- ✓ Add a new task under the Skills heading of your resume!

Who Are You?

If you have this book in your hand, you probably fit into one of the following three categories:

- ✓ You never programmed a computer before.
- ✓ You programmed a computer, but never a Macintosh.
- ✓ You tried programming a Mac, but were frustrated in your attempts.

If one of the above applies to you, this book is for you. If you have prior programming experience, that's great. But this book never assumes that you have programmed anything before. The explanations and programming examples in *Mac Programming For Dummies,* 3rd Edition, make no assumptions about your programming knowledge — everything starts at step one.

What You Need

Surprise! You need a Macintosh or Mac-compatible computer if you want to create a Macintosh program. Sure, you can just read this book and think you've learned how to program a Mac. But to really be sure that you know what you're doing, you have to try out the book's examples. What kind of Macintosh do you need? Just about any model will do. A Mac II, LC, Centris, Quadra, Performa, Power Macintosh — even the PowerBook — are all suitable. If you're one of the hundreds of thousands of people who bought an iMac, you contributed to Apple's 1998 comeback and 1999 success. You also bought a good machine for developing programs. In short, if you own a Mac, you're probably all ready to go. If you don't own a Mac, find someone who does and use it as often as you can. (I'm assuming, of course, that you know this person.)

Whichever model Macintosh you want to program on, it needs to be equipped with the following three things:

- ✓ 24MB or more of RAM memory
- ✓ System 7.5 or higher Mac operating system
- ✓ A CD-ROM drive

If you aren't sure how much memory or what operating system your Mac has, follow these steps to find out:

1. **Move the cursor so that it is over the desktop and click the mouse.**

   (I'm assuming you're at your Mac and it's turned on, of course.)

2. **Choose the first menu item under the Apple menu, which is the About This Computer item.**
3. **Look for the words Built-in Memory in the dialog box that opens.**

   If the value listed is at least 24MB, then you have the required amount of RAM.

4. **Look in the upper-right corner in the About This Computer dialog box.**

   If the number to the right of the Mac OS logo is 7.5 or higher (numbers such as 7.5, 7.6, 8.0, 8.1, and 8.5 are some of the numbers that qualify), then your system is just fine for programming.

What's on the CD

Along with a Macintosh, you need a program called a compiler. A *compiler* turns words that are readable by you, the person, into words readable by the Macintosh, the computer. Because you'll be anxious to try out the programming skills you're about to acquire, IDG Books Worldwide, Inc., and a software company named Metrowerks have seen to it that a compiler — the Metrowerks CodeWarrior Lite compiler — appears on the CD-ROM that accompanies this book. And to save you a little (and sometimes a lot of) typing, all of the programming examples in this book also appear in files on the CD-ROM.

About This Book

Many books about computers are reference books. Only when you need help with a problem do you refer to the book. As such, that type of book isn't meant to be read from cover to cover. If you're a flat-out beginner, this isn't one of those books.

You may have read computer books that teach you how to use an existing program on your computer. Books like that show you how to do certain things with a certain program. An example may be learning how to format text using WordPerfect. This isn't one of those books either.

Very good — now you know what kind of book this isn't. What then, is this book? It's a book that teaches you how to actually write a brand-new, never-before-seen-by-another-person Macintosh program. If you've never written a computer program, this may sound like an insurmountable task. If you have programmed, but not on a Macintosh, this may sound like an insurmountable task. It isn't.

Now, I know that on the bookstore shelf, just inches away from where this book was sitting, there were Macintosh programming books twice as thick as this one. Books with enough techno mumbo-jumbo and four-syllable words to make a rocket scientist scratch his head in wonderment. How can I make the claim that anyone can learn, with minimal effort, to write a Macintosh program just by reading this much-smaller-than-a-breadbox-sized book? It's possible because I:

- ✓ Spare you unnecessary technical details.
- ✓ Make no attempt to address advanced Mac programming techniques.
- ✓ Spare you unnecessary technical details.
- ✓ Make no attempt to cover every facet of Macintosh programming.
- ✓ Spare you unnecessary technical details.
- ✓ Avoid assumptions about what you already know about programming.

If you want to delve into the deepest, darkest inner workings of Mac programming, don't buy this book. If you want to learn fundamental programming concepts that allow you to create a Macintosh program that opens a window, draws and writes to that window, and displays menus, do buy this book.

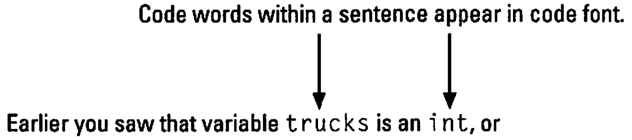
Because *Mac Programming For Dummies,* 3rd Edition, concerns itself with writing programs, not with using them, it's important that you know just how to use this book. Which leads to the very next topic. . . .

How to Use This Book

The fact that you bought, or are considering buying, a programming book with the word Dummies in the title tells me that you're probably new to programming (or at least new to programming the Macintosh). Learning to program is an incremental process. And you should learn each increment, or step, in a specific order. As your experience in writing Macintosh programs grows, you'll be able to take liberties and try out ideas of your own. At that time, you may find yourself using this book as a refresher or reference, skipping around between chapters and using the book's headings to locate just the information you need. But for now, I strongly recommend that you start, as they say, at the beginning. I won't give any exotic examples or pull any fast ones, but each programming example builds on material presented in previous chapters. So do read the book from cover to cover — front cover to back cover, that is!

It's Time to Establish Some Conventions

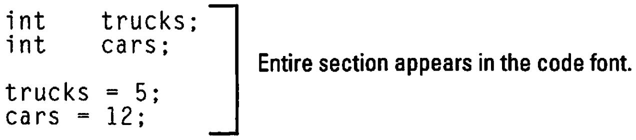
You'll find source code scattered throughout the chapters in this book. If you weren't familiar with the term before you read this introduction, you now know that *source code* refers to the code that programmers use to write computer programs. To help you quickly identify what is source code and what is regular text, all code that is mixed in with text on a page appears in a font that differs from that used for normal text:



Code words within a sentence appear in code font.

Earlier you saw that variable `trucks` is an `int`, or

In other places in this book, you see blocks of code — a section of source code that appears in its own paragraph, isolated from the rest of the text on a page. Those cases use the special code font as well:



```
int    trucks;
int    cars;

trucks = 5;
cars = 12;
```

Entire section appears in the code font.

How This Book Is Organized

This book contains seven major parts. Each part is composed of three or more chapters, which makes the material easy to digest. But please remember to chew 30 times before swallowing.

Just a bit earlier I recommended reading this book in order, without skipping material. Let me elaborate a little. I don't want to scare you into thinking you must read at a snail's pace, avoiding at all cost the skipping of even a single word. Before and after I list any source code — that stuff that lets the computer understand what you want it to do — I quickly review the concepts the source code covers. So if you do skip every once in a while, you won't be totally lost when you see new source code.

If you've programmed before, but not on a Macintosh, I hereby grant you permission to skim — and perhaps even skip — a chapter or two that pertain to the most basic of programming concepts. In particular, you may be familiar with much of the material in Chapters 12 through 15. For the rest of you, don't skip — you'll be quizzed at the end!

Part I: Introducing the Macintosh Basics

What makes the Macintosh different from other computers? What makes programming the Mac different from programming other computers? I give up, what? Just kidding. Part I shows you why you need a book devoted entirely to the Macintosh. You are also convinced that programming need not be such a scary endeavor — honest!

Part II: Resources: This Is Programming?

Text is boring. Windows, menus, icons, and pictures are exciting. The Macintosh is fun because programs that run on it contain some or all of these neat elements. They make Mac programs fun to look at and fun to use. All this talk about fun is fine, but what about adding all of these cool items to a program you're creating? Is that process fun, too? As a matter of fact, it is! And things called *resources* have a lot to do with making it fun. Resources help you create neat features, such as windows and menus, with — hold on to your hat — absolutely no programming on your part! In this part of the book, you learn exactly what resources are and how you can easily create them.

Part III: Using a Compiler

To write a program, you type in a series of commands — commonly referred to as *source code*. A computer can't, however, understand the words you type into it. The words have to be translated into numbers (computers love numbers). Turning words into numbers sounds like a tedious and ugly task. Fortunately, you'll never know for sure because you'll never have to do it. Instead, you just run a program that does all that work for you. The program that performs this translation is called a *compiler*. In Part III, I talk about what a compiler is and how to use it. In particular, I talk about the compiler that's included on this book's CD-ROM — the Metrowerks CodeWarrior Lite compiler.

Part IV: Learning the C Language

While it would make things easy if your Macintosh could understand English language statements such as, "Place a window on the screen and draw in it," things haven't quite progressed to that point — yet. Instead, you need to be a little more formal when you tell the Mac to do something. That's what a computer language is for. When you write a computer program, you write it in a computer language. There are several computer languages you could use. This book uses the C language in all its examples — it's the most popular programming language. Part IV describes computer languages in general and the C language in particular.

Part V: The Moment of Truth: Writing a Program!

With the preliminaries out of the way, it's time to create a Macintosh program — a real one. A program with a window and a menu. You provide a little flair to the program by adding animation to it. This section describes a very simple technique that allows you to bring your program to life. And isn't that what the Macintosh is all about?

Part VI: The Part of Tens

Ten things to do in order to make a program, and ten things you don't want to do. And ten indispensable functions, aids that make writing a program with menus and windows easy.

Part VII: Glossary and Appendixes

You won't remember everything that you read in this book. So I provide a few appendixes for those occasions when you get stuck: Appendix A is a reference for the C language; Appendix B is a reference for the Toolbox; Appendix C helps you if something goes wrong with your program; Appendix D is a glossary; Appendix E holds a few tips for iMac owners; and Appendix F lists what's on the CD-ROM included with this book.

Icons Used in This Book

TECHNICAL STUFF

Optional reading — while not technically intense, these explanations may not be suitable for small children!

WARNING!

Don't worry, you can't wreck your Mac. But you can wreck the work you're doing on it — I point out the areas where this could happen.

TIP

Programmers are crazy about optimizing their efforts. Just about *everything* has a shortcut. I note the more worthwhile ones.

MAC PSYCHOLOGY

Programming is shrouded in secrecy. Knowing a little Mac psychology goes a long way to understanding why some seemingly obscure terminology needn't be.

ON THE CD

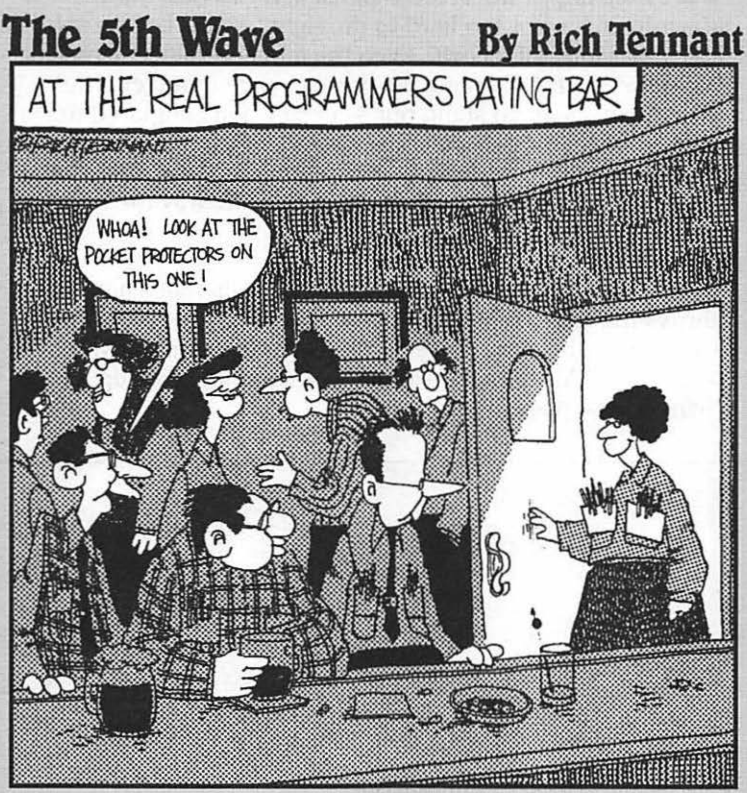
Points out places in the text where you need to use stuff on the CD-ROM that comes with this book.

What's Next?

Why, reading the book, of course. The introduction is over, and it's time to program. And — dare I say it — to actually have a little fun!

Part I

Introducing the Macintosh Basics


The 5th Wave
By Rich Tennant
AT THE REAL PROGRAMMERS DATING BAR
WHOA! LOOK AT THE POCKET PROTECTORS ON THIS ONE!

In this part . . .

Line up a few different types of computers in a row, and by looking at the screens of each, try to pick out the Macintosh. It's not too hard to do, right? That's because the Mac has a look all its own. Even the much-heralded arrival of Windows 95 and Windows 98 didn't change that fact. The Macintosh seems to stand out — even when compared to other systems that use menus, windows, and icons. Because the Mac is different from other computers, it probably won't come as much of a surprise to learn that the way the Mac is programmed is a little different, too.

Different is often equated with difficult. Well, that's not always the case. And programming the Mac is one example. To prove it, this part presents the code for an actual, honest-to-goodness Macintosh program — code that takes up less than a single page in this book!

Chapter 1

Windows, Menus, and a Mouse — That's the Mac

In This Chapter

- Introducing the graphical user interface
- Examining different parts of the interface
- Picking out the parts of the interface that you need for programming

During the telecast of the Super Bowl in January 1984, a commercial — one that would air only a single time — told of the emergence of a new computer. This computer, the commercial claimed, would change the way people thought of computing. The computer was, of course, the Macintosh. While it may not have revolutionized computing overnight, since the Mac was introduced, it has lived up to its promise of changing how people interact with computers.

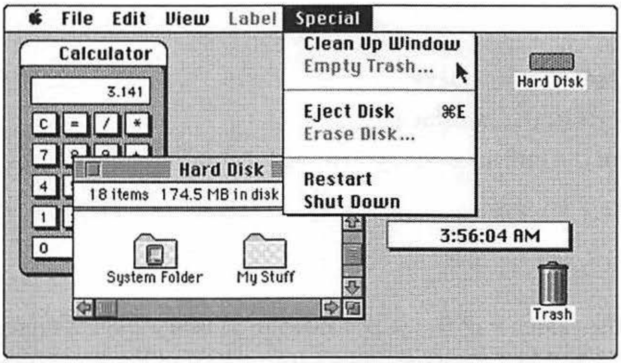
The Graphical User Interface

What was it about the Macintosh that, over a decade ago, set it apart from almost all other personal computers? It was the Mac *graphical user interface.* The graphical user interface, or GUI (pronounced *gooey*), is made up of icons, menus, and windows. (Graphical user interface is a mouthful, and GUI is just too darned ugly a word. So from here on, I usually just refer to the GUI as the *interface.*)

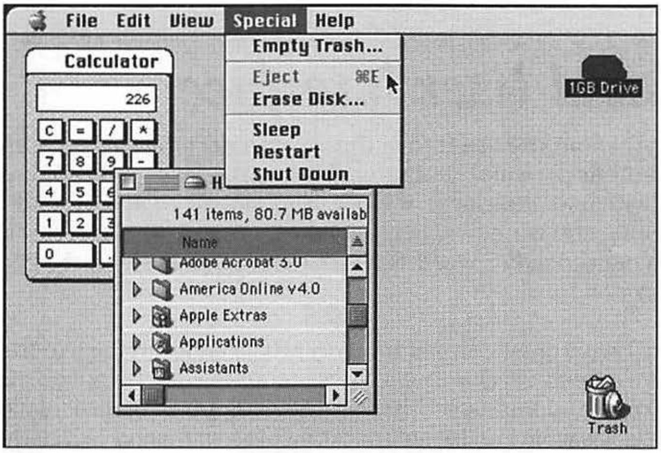
In the mid-1980s, most computers didn't have an interface like the Mac. Instead of a screen filled with graphics, computer users looked at a screen filled with text and numbers. Many of the computers that displayed this text-based interface were run by an operating system named *DOS* (which stands for *disk operating system*). People working on a DOS computer see a lot of words on their screens, not pictures. They tend to type keyboard commands to get things done. DOS users generally don't use a mouse. Some poor souls still use DOS (usually MS-DOS, a particular brand of DOS made by Microsoft to run on PCs). And while they have the somewhat graphical DOS shell as an optional add-on to DOS, these users are still behind the times.

Macintosh programmers don't care much for DOS programmers. It's nothing personal, though. It's just that Mac programmers feel they're riding the wave into the 21st century, while they see DOS programmers still floundering about in the Stone Age.

Here's what a typical DOS user sees on the screen:



Pretty exciting stuff, huh? Compare the DOS screen with a typical Macintosh screen with its windows, menus, and icons:



Looking at these two figures, it's quite apparent why many people prefer to do their work on a Macintosh computer. The screen of a Mac provides a much friendlier *environment*, or atmosphere, to work in.

By the way, DOS is pronounced *doss*, not *dooze* or *dose*. You may already know that, but I just wanted to be sure. That's to prevent you from being on the receiving end of a scornful look or comment from an established Mac or DOS programmer. That brings us to another bit of Mac psychology (really, computer programmer psychology in general). Some programmers like to make themselves look better by making others look worse. Jumping all over someone who incorrectly pronounces a computer-related term is a favorite tactic of this type of programmer. I help you weather this storm by providing the pronunciation of words whose pronunciation may not be intuitive.

The Macintosh computer isn't the only one with a GUI. A PC (for *personal computer*, and pronounced pee-cee) can run DOS, but it almost always comes equipped with a version of Microsoft Windows (such as Windows 95 or Windows 98). Most would agree that the Macintosh, however, is the standard by which other graphical user interfaces are judged. You can confirm this by listening to Windows 98 users say things like "With Windows 98, I can do that task just the same as on a Mac."

The Interface Parts

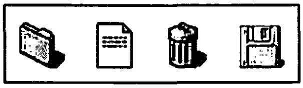
Users of Macintosh programs become familiar with the various parts of the interface, such as menus and icons, so that they can make better use of the computer. As a soon-to-be Macintosh programmer, you need to become familiar with these same interface components, but from the behind-the-scenes perspective of a programmer.

Working on your desk

The interface begins at the ever-present background screen. Apple calls this screen the *desktop*, which is that area that covers most of your Macintosh monitor. The desktop is often a light gray color, but it doesn't have to be. The desktop analogy is, of course, that you organize electronic files and folders on the desktop as you would real ones on a real desk. Just what a trash can is doing on *top* of the desk is anyone's guess. Oh well, no analogy is perfect. Still, *desktop* is a catchy name that's deeply embedded in the history of the Mac.

Looking at itty-bitty pictures

An *icon* is a picture. Not just any picture, though. It is a small graphic symbol that is a representation of something. If an icon is well designed, its representation is obvious. Icons are small, usually about a half-inch square or less. Here are a few of the icons that you find on the Macintosh desktop:



After you write your very own Macintosh program, it will have its very own icon. And where will this icon be? Somewhere on the desktop. Why *somewhere* on the desktop? Won't you know exactly where your new program icon will be? Yes, you will know where it is until you turn your program over to another user. Because like the icon of any program, the user will be free to drag your program's icon to any folder anywhere on the desktop of his or her computer.

Peeking through windows

Windows are an important part of any graphical user interface. Windows are so important that Microsoft has named its own GUI operating system after them. For users of PCs, Microsoft Windows has all but replaced DOS. That gives you a hint at how much people enjoy working with a graphical user interface.

Unlike a DOS program that writes text directly to the screen, a program designed for a computer with a GUI writes text to a window. The same applies to graphics. Because just about everything on a Mac is done in a window, the creation of a window should be one of the first programming tasks you tackle. Throughout the early chapters of this book, you see hints and references to creating a window, and in Chapter 17 you get the specifics.

Using the mouse

The mouse is used to make menu choices, move windows, and double-click on icons. In general, the mouse is the user's means of interacting with the computer. Or, if you want to go toe-to-toe with Mac programmers, you can say that the mouse is the standard *input device* for communicating with the computer.

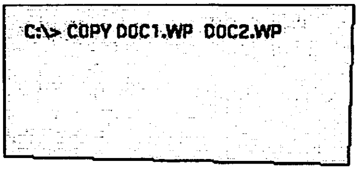
As a Macintosh programmer, you can write programs that discern exactly what the user is doing with the mouse. Well, not *exactly* what the user is doing with it. Your program can't know, for instance, if the user is holding the mouse by the cord and spinning it over his head. But your program can notice (and respond) if the user clicks on the mouse button while the cursor is over one of your program's menus. And that, of course, is a much more important thing to be able to recognize!

When certain events (such as a click of the mouse button) occur during the running of the program, the computer code for that program is able to recognize what happened. What's the tricky techno-terminology Mac programmers use for the occurrence of such an event? They call it, well, an *event*. There, that wasn't so technical after all! You can read all about events in Chapter 17.

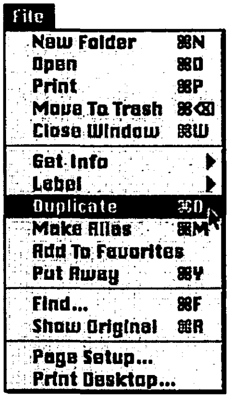
Ordering from the menu

Older computers performed an action in response to a command that the user typed in. On a Macintosh, you don't have to type orders. Instead, you can order your desired commands from a *menu* — sometimes referred to as a *pull-down menu*. To issue a command, you move, or drag, the mouse until the cursor on the screen is positioned over a menu. You then click the mouse and then make a selection from the list of commands that drops down. Because a menu does in fact drop down, rather than pull down, you may think that a better name would be *drop-down menu*. That may be true, but it's a little late in the game to make a change in the terminology Apple has selected.

Menus are one of the most powerful features of the Macintosh interface. Having all the available commands spelled out before you as menu *items* means that you don't have to memorize or look up the names of commands. And even if a command is easy to remember, most people find selecting a menu item quicker than typing in a command. As an example, consider the following figure which shows how to duplicate a file in DOS and on a Mac:

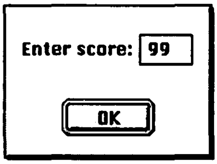


```
C:\> COPY DOC1.WP  DOC2.WP
```

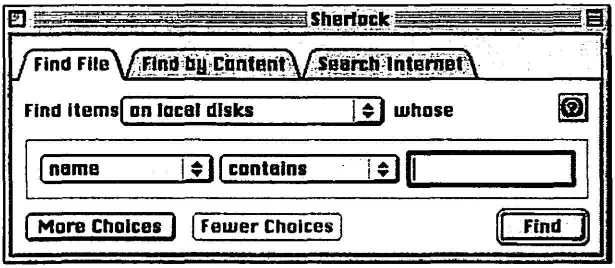


Speaking of dialogs

Windows generally *display* information. How does a program receive information from the user? Usually it's through a *dialog box*. What do dialog boxes look like? Here are a couple examples of dialog boxes:



Enter score: 99



Find items on local disks whose

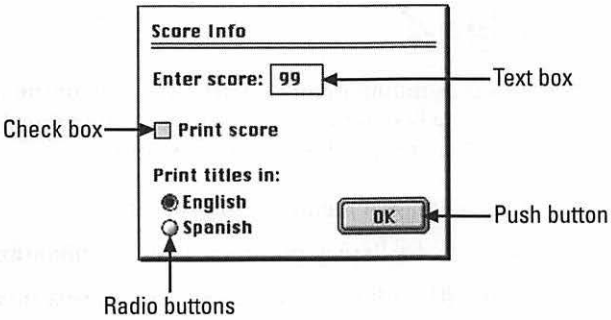
name contains

You can see that a dialog box may or may not look like a window. And it may or may not behave like one. How can you tell the difference between a window and a dialog box? If you are the programmer who designed the program, you already know. If you're just someone using the program, you may not be able to tell the difference. Here's a scientific, highly technical test you can apply to determine if the thing you're looking at on the screen is a window or a dialog box: If it has a bunch of stuff in it that you can click on, it's probably a dialog box.

Speaking of stuff in a dialog box, here's a quick look at most of the different types of items you may find in a dialog box:

- **Push buttons:** Clicking the mouse button while the cursor is positioned over a push button causes some action to take place immediately. The OK or Done button found in just about every dialog box are prime examples.
- **Radio buttons:** These allow you to select from a number of options. This type of button travels in packs — never alone. Only one member of the group is ever selected at any given time.
- **Check boxes:** Some options listed in a dialog box can be turned on or off. A check box lets you do that.
- **Text boxes:** If you ever typed a number or word into a rectangle, you worked with a text box. If a program needs information, whether it be your bowling score or your date of birth, this item allows you to type it in for the Mac to use. A text box is also called an *edit box* sometimes, just to confuse you.

Here's a figure that shows each of the dialog items I just mentioned:



Score Info

Enter score: 99

□ Print score

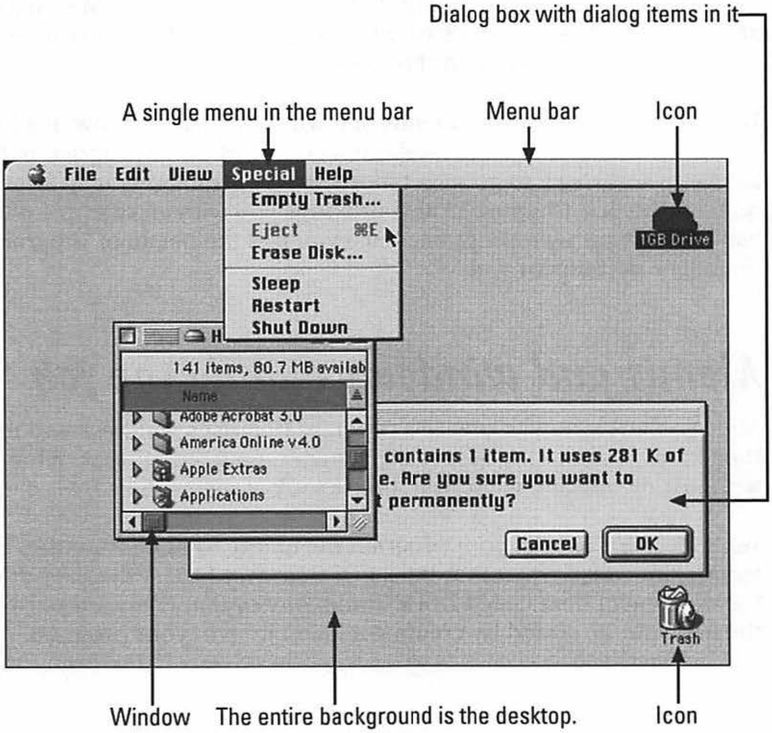
Print titles in:

● English

○ Spanish

Wrapping up the interface

That's just about it for the interface, except for one more figure. I created the following figure for two purposes. First, I want to summarize many of the parts of the Mac interface in one figure. Second, I want to test the capabilities of IDG Books' layout department; could they accurately reduce the size of what started out as a very large figure? Success!



141 items, 80.7 MB availab

contains 1 item. It uses 281 K of
e. Are you sure you want to
t permanently?

The Parts You Need

The list of things that a programmer can do with the parts of the interface is just about endless. I obviously can't produce this list in its entirety. But here are a few common tasks Mac programmers can perform:

- Build a functioning pull-down menu into a program.
- Add a movable window to display graphics, text, or animation.
- Create a dialog box with radio buttons, check boxes, text boxes, and push buttons.
- Add sound-playing capabilities to any program.
- Allow a program to open, save, and print files.
- Design a unique icon.

The list goes on and on. This wealth of programming options really gets a seasoned programmer excited. It also can scare the living daylights out of someone who hasn't programmed! Many programmers believe the more the merrier, but you should keep one very important point in mind: You don't *have* to include everything from the preceding list for a program to run on a Macintosh. In fact, when you first start programming the Mac, it's best to stick with the basics. That's why I concentrate on the first two items in the list, menus and windows, in this book.

While I do concentrate on menus and windows, I also throw in a little information — and a little code — about a couple of other program parts. Chapter 22 discusses where to go to get more Mac programming help after you conquer the basics. Chapter 22 also provides you with an example of what you can add to your own programs once you feel the beginner programmer label no longer pertains to you.

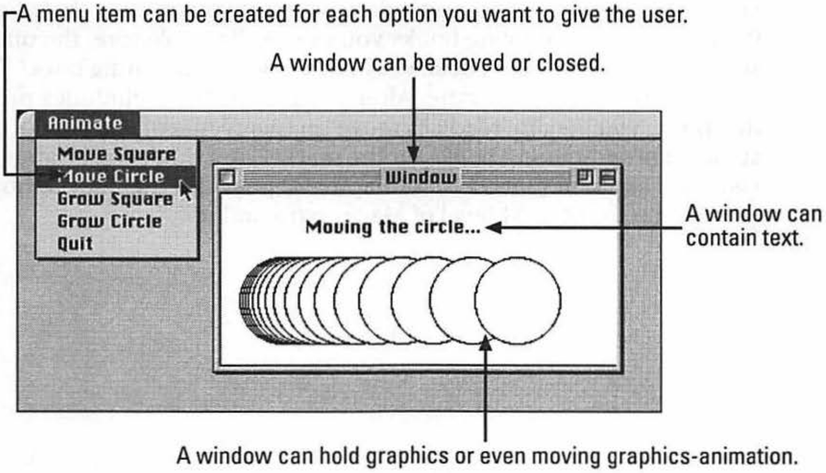
Menus and windows can do the job

After you know how to create a program that uses windows and menus, you then know many of the Mac basic programming techniques. What can you do with just menus and windows? Take a look at menus first.

Want to give a user of your program the ability to open a window? Add a menu item called Open that does just that. Want to let the user draw a circle? Create a menu item called Draw Circle. Any option you want to provide for the user can be added by creating a menu item in your program.

What about displaying text or graphics? A window can contain either words or drawings, or both. What if you want to create an animated effect, a moving picture? Because a window can hold graphics, it can also hold moving graphics.

The following figure shows you why a program with but a single menu and a single window is a true Macintosh program.



A menu item can be created for each option you want to give the user.

A window can be moved or closed.

A window can contain text.

A window can hold graphics or even moving graphics-animation.

So, you think you're getting shortchanged, huh?

If you can write a program with a menu and a window, you can write a program that does just about anything you want it to do. Menus and windows, windows and menus — that's it? That's it as far as what the user of one of your programs is concerned. You, the programmer, have to be wiser about far more than that. But don't worry; in covering these two parts of the interface, you pick up knowledge and experience in all of the following programming areas:

- **Menus:** How to make, display, and work with them.
- **Windows:** How to display, move, and draw in them.
- **Text:** How to write words, in different sizes and styles, to a window.
- **Graphics:** How to draw lines and shapes to create pictures.
- **Animation:** How to make graphics that appear to move.
- **Events:** How to see just what the user is doing.

That last point sounds particularly interesting — and possibly illegal! Macintosh programs, more than many other types of programs, are interactive. A user does something, and the program somehow knows what the reader is doing and responds. Macintosh users enjoy the resulting feeling of control. And Macintosh programmers enjoy creating programs that satisfy users.

After you read this book and follow its examples, you may find yourself hooked on Macintosh programming. What do you do after that? Remember those other programming books you saw in the bookstore, the ones on the shelf right by this one? Those very fat, wordy, intimidating ones? They won't look so intimidating anymore. After you master the techniques presented in this book, you may be ready to move on to any one of the host of intermediate-level programming books on the market. But don't worry about all that just now; and don't forget that Chapter 22 provides the information you need to move on to the next level of Mac programming.

Chapter 2

What Makes Macintosh Programming So Different?

In This Chapter

- Taking the Mac interface challenge
- Seeing how source code makes Mac programming different
- Picking out differences in the insides and exteriors of Mac programs
- Finding out that easier (DOS programs) isn't always better (Mac programs)
- Using windows and menus makes Mac programming more fun

The programming skills and techniques needed to write a Mac program are different from those necessary to write a program designed to run on other computers. In this chapter, I cover the basic differences between Mac programming and programming for other machines. This chapter may be especially useful if you programmed before, but never on a Mac. If you never programmed before, this chapter helps you get acquainted with some issues you face when you program the Mac.

The Interface — That's the Difference

Why are Mac programs easy to use? Because the programmer puts extra effort into the interface to make it that way. If you intend to construct a graphical user interface rather than a text-based interface (such as DOS), you, the programmer, are responsible for more things. For example, programmers who write programs for DOS computers write lines of text to the screen. They don't have to worry about how to display a window, or create a pull-down menu, or any of that other fun, typical Mac stuff.

Versions of Microsoft Windows, such as Windows 95 and Windows 98, rely on a graphical user interface. While there are many similarities between the Macintosh GUI and the Windows GUI, there are also a number of differences. Knowing about one graphical user interface doesn't guarantee that you'll automatically know all about a different GUI.

So it's the interface that's the key difference between using, and programming, a Macintosh and other computers. With that said, should I wrap up this chapter right here and now? Sorry, you don't get off the hook quite that easily. The rest of this chapter elaborates on the differences between Mac programming and non-Macintosh programming. If you have any non-Macintosh programming experience, this information should help you make the transition to programming the Mac. If you've never programmed at all, the following pages are still of great use — they're loaded with basic Mac programming concepts and terminology.

If you haven't done much programming, or any programming for that matter, you may actually have an advantage over programmers with years of experience on DOS computers. No, I'm not just saying that to make you feel better! It's true. Those programmers have to unlearn many of their old ways of doing things. You get to start with a fresh, clean slate. By the way, if you find a slate, let me know. I haven't seen one around for years!

Secret Agents Aren't the Only Ones Using Code!

Before delving into the differences between Macintosh programming and other types of programming, I have one more digression. Programming — any programming — relies on *source code*. That's the stuff that lets you tell a computer just what to do. Since you read quite a bit about source code in the rest of this book, let me say a bit about source code right here and now.

A programmer creates a program; a user simply uses that program. The programmer writes *source code* to create the program; the user doesn't know or doesn't give a hoot about source code. Before this book is over, you, the programmer, will be on intimate terms with this thing called source code.

Like a relationship with a person, your relationship with source code may be both very satisfying and very frustrating. Just when you think you have things all figured out, along comes a new twist or turn that throws you completely off course.

I know many of the pitfalls that most new Mac programmers encounter, so I can help you bypass them. I've programmed the Mac for a decade and a half; I can aid you in your relationship with source code — I'm an expert at relationships. Of course, my six ex-wives may not agree.

Learning the language

Computers, while exceedingly powerful, lack one important capability that people possess, which is the ability to interpret. For example, you and I know the difference between two uses of a word such as *lead.* If I were to say, "You can lead a horse to water" or "Lead is a heavy, soft, malleable metal," you could recognize these two very different uses of the word. From the context in which a word is used, people can interpret its meaning. That's a skill that a computer doesn't have.

How do you then get a computer, which has no interpretive power, to understand and do what you want it to do? By issuing commands to it. But not just any old commands. You can use only commands that are defined by a rigid set of rules. By using only these established commands, the computer doesn't have to interpret anything, which is the way the computer likes it. That's what a computer language is all about.

Like a spoken language, such as English or German, a computer language has a limited vocabulary. Fortunately, the number of words a computer language allows you to use is *very* limited. That means learning a computer language is *much* easier than learning a spoken language. Thank goodness! *Gott Sei Dank!*

Different languages

Just as there are different spoken languages, there are different computer languages. Wouldn't just one be enough? Again, like spoken languages, one would be enough if you could get everyone in the world to agree to use the same one! Over time, different people, different universities, and different companies have all created what they felt was the best computer language. And over time, as computers changed, computer languages have changed.

BASIC, Pascal, C, C++, and Java are the names of five common computer languages. In this book, I use the C language for all of the programming examples. What criteria did I use to make this choice? I studied, experimented, and worked with each. Then I accepted the $100 bribe that Dennis Ritchie, the creator of the C language, offered me. Seriously, I selected the C language because it is currently the language of choice of Macintosh programmers.

Computer programmers battle ceaselessly about which language is the best one to use. Like debates about politics or religion, no one ever wins one of these arguments. Should you be in the vicinity of one of these discussions, my best advice is to head for cover!

I devote the five chapters of Part IV to the C language, so please look there for more detailed information on the C language.

All programs were once source code

The rough draft of this book, whether I write it in English, German, or Sanskrit, is called a manuscript. The same concept applies to computer languages. Regardless of which language you use, you get the same result: *source code*. Whether you use C, C++, Pascal, or Java, the product of your work is a page, or perhaps tens or hundreds of pages, of commands (source code).

How does source code differ from the program itself? Source code is transformed into programs by something called a *compiler*. A compiler performs this amazing feat in just a couple of seconds. The CD-ROM that's bundled with this book includes a Lite version of the Metrowerks CodeWarrior *compiler*, a software program that turns source code into a Macintosh program. *Lite* means that some of the functionality of the full-featured version has been removed. Don't scowl as you read that — what did you expect for practically free! The Lite compiler lets you compile the C language examples from this book. The full-featured version, available from Metrowerks, lets you write source code in your choice of four computer languages: C, C++, Pascal, or Java. More on compilers in Part III.

If you've seen programs that run on a Macintosh and programs that run on other types of computers, you may have noticed that they don't resemble one another very closely. The *exteriors* of the two types of programs are very different. Does that mean that the source code from which the programs evolved also looks different? Clever you — indeed it does! This is, of course, another one of the major reasons that Mac programming is so different from other types of programming.

I won't show you any source code now. I do, however, want to explain how the different look of two programs means that different programming efforts were put into each. In fact, I go into that right now.

Programs Inside and Outside

Source code can be thought of as the basis for the interior of a program. A compiler turns source code into still a different type of code — object code. It's this object code that is actually the program itself. So code is the programmer's tool for planning out and implementing a program, and code actually makes up the final program. Code is something the user of a program doesn't see or work directly with. What the user does see — menus, windows, graphics — can be thought of as the exterior of a program.

Many people believe that a program that runs on a Macintosh computer is easier to use than a DOS program. (You probably think that, too, because you're reading this book.) A Mac program is easier to use because its exterior (what the user sees) contains useful features, such as windows and menus. But what about its *interior* — its code? Is the code for a Macintosh program easier to write than the code for a DOS program? The short answer is no; writing code for a Mac is harder than writing code for a DOS program. The long answer involves a story about cars. Sure, it's a bother not to settle for the quick answer, but I promise that this short story is helpful.

Imagine a car built in the 1970s. On a cold day, you start the car by pumping the gas pedal several times with your platform shoe, and perhaps then holding the pedal to the floor as you turn the ignition over, trying not to get the keys caught in your love beads. To stop the car on a wet or icy road, you pump the brake pedal.

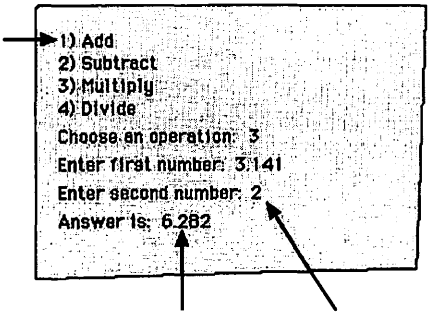
Now imagine yourself in a car of the 1990s (and a better haircut). With fuel injection, you simply turn the key and start the car, regardless of the temperature outside. With anti-lock brakes you simply press down on the brake pedal, regardless of road conditions. The car of the 1990s is easier to use and works better than the car of the 1970s. But to simplify the parts of the car that the driver uses, the parts on the inside became more complicated. If you peek under the hood of the 1990s car, you see much more machinery than is under the hood of the 1970s car. That's true with many things that are affected by technology. A smooth, sleek, easy-to-use exterior masks a complex, highly refined interior.

But wait! This isn't *Chilton's Auto Guide*. This is a book on programming the Mac. So how does all of the above pertain to programming? A Macintosh program is like the car of the 1990s, while its DOS counterpart is like the car of the 1970s. While the user of the Mac program finds it is easier to use and more intuitive than a DOS program, the programmer who writes the Macintosh program deals with much more complex code than the programmer of the DOS program.

What about that other GUI, Windows? Is the Mac programmer responsible for more, or different, things than a Windows programmer? More, no. Different, yes. Although Mac programs and Windows programs may look alike in some ways, the programmers responsible for writing the source code for each do things in a different way. If you know someone in the unenviable role of having to learn programming for Windows, you should tell them about two titles: *Windows 95 Programming For Dummies,* by Stephen Davis, and *Windows 98 Programming For Dummies,* by Stephen Davis and Richard Simon (both from IDG Books Worldwide, Inc.).

Easier Doesn't Mean Better

It's easier for a programmer to write a DOS program because the programmer is responsible for less. Just what is meant by responsible for? Read on to find out. As you read, refer to the following figure. It shows part of a screen displaying a DOS program that acts as a very simple calculator:



```
1) Add
2) Subtract
3) Multiply
4) Divide
Choose an operation: 3
Enter first number: 3.141
Enter second number: 2
Answer is: 6.282
```

Write text.

Write numbers.

Read numbers that the user enters.

The person who programmed this calculator was responsible for a number of things: writing text to the screen, writing the on-screen menu that lets a user select an arithmetic operation, writing code so that the program reads numbers typed in by the user, and writing code that performs a calculation and then displays a number to the screen.

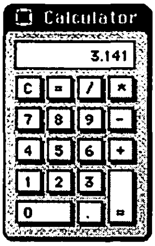
Data is a general computerese term for letters, words, or numbers. When a computer program displays words or numbers on the screen for the user to view, it is *writing* data to the screen. When a computer program receives words or numbers from the user, it is *reading* data. The most common means of entering data for the program to read is by typing on the keyboard. There is another means, but it's usually not available in DOS programs. Macintosh programs sometimes allow you to use the mouse to enter data.

The number of things that the DOS programmer is responsible for doesn't sound overwhelming, and it's not. Because a DOS program doesn't contain windows, icons, or menus, a DOS program is easier to write than a Macintosh program.

To someone who hasn't programmed before, or has programmed very little, easier surely sounds better than harder. But there is a price one pays to write a simple program — you end up with a simple program! A simple program, like the DOS calculator program, doesn't look very interesting and doesn't do a whole heck of a lot.

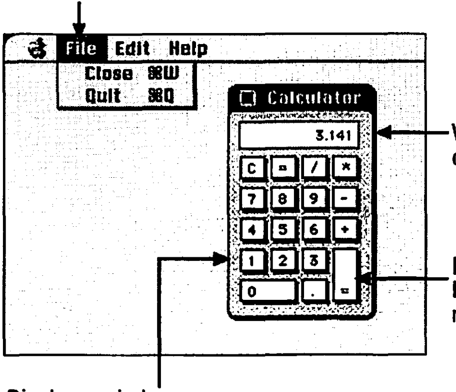
Mac Programs — Interesting, Fun, Exciting!

Why has the Macintosh become so popular over the last several years? You already know the answer: Mac programs are easy to use, fun to work with, and interesting to look at. Remember how the DOS calculator program looked? The figure that follows shows a Macintosh calculator program. It's a free program that Apple includes with all Macintosh computers.



In the previous section, you saw the job a DOS programmer has if he decides to write a calculator program. Now take a look at what a Macintosh programmer would be responsible for if she were to write a spiffy calculator program like the one just pictured:

Display a menu.
Determine when the user makes a menu selection, and which menu item was selected.



Write numbers in a specific area of a window.

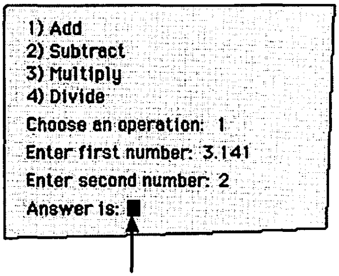
Display objects in a window.
Determine when the user clicks the mouse on an object.

Display a window.
Determine when the user moves it.

Now you've seen the kinds of jobs that DOS and Mac programmers face when they want to complete the same task, which is building a calculator program. Comparison is inevitable — plus I've got to tie this conversation to the chapter title at some point!

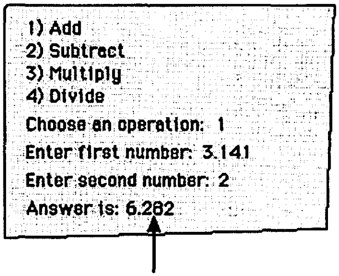
Giving information

The preceding example shows that the Mac programmer, like the DOS programmer, writes data and reads data. But the Mac programmer does both a little differently. Take a look at writing data first. Remember where the calculated result was written to in the DOS calculator? In a DOS program, data is written at the current location of the cursor:



```
1) Add
2) Subtract
3) Multiply
4) Divide
Choose an operation:  1
Enter first number: 3.141
Enter second number: 2
Answer is: █
```

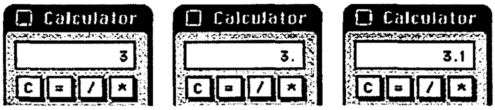
Cursor appears after the last text written to the screen.



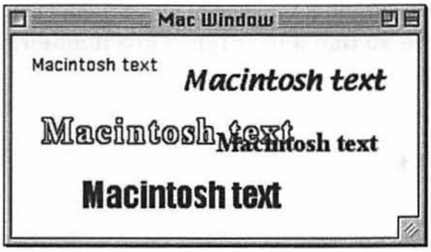
```
1) Add
2) Subtract
3) Multiply
4) Divide
Choose an operation:  1
Enter first number: 3.141
Enter second number: 2
Answer is: 6.282
```

The next data that is written to the screen appears at the location of the cursor.

In a Mac program, data can be written *anywhere* in a window. In the Macintosh calculator program, the user clicks the mouse on a number or symbol button, and the corresponding number or symbol is written in the white box at the top of the calculator. As a digit is entered, it always appears at the far right of the white box. Here the 3, ., and the 1 are entered one after another:



Not only can text and numbers be written anywhere in a Macintosh window, they can even be made to overlap other text. And the *style* — the appearance — of data can be altered:



Macintosh text

Macintosh text

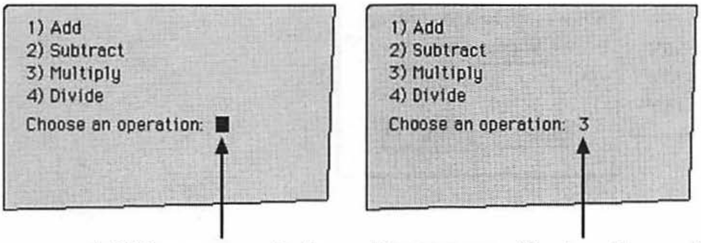
Macintosh text Macintosh text

Macintosh text

The ability to control the appearance and placement of data is an important feature that separates the Macintosh from many other computers. Another difference is how the user of a program enters data into the Mac.

Getting information

Programs written for DOS computers read data by pausing and waiting for the user to type in words or numbers. Pressing the Enter or Return key signals the program to read the typed value and then continue. Once again, the calculator example:



```
1) Add
2) Subtract
3) Multiply
4) Divide
Choose an operation: ■
```

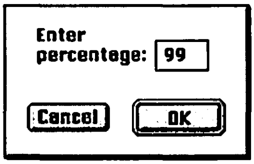
```
1) Add
2) Subtract
3) Multiply
4) Divide
Choose an operation: 3
```

A DOS program waits for the user to enter a number.

The program will not continue until the user types a number and presses the Enter key.

Ready for one of the most central concepts of Macintosh psychology? Ready or not, here it is: *The user is the boss.* People like using the Macintosh because they feel that they are in control. A good Macintosh program seldom freezes the screen, forcing the user to do something before continuing. Where have you seen this type of unfriendly forceful behavior? In the preceding example of the DOS calculator program that won't continue until the user enters a number.

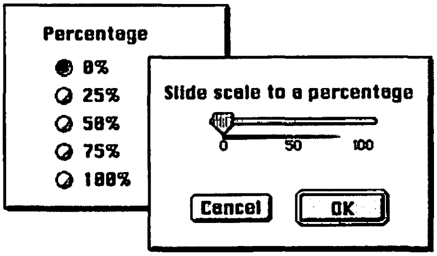
A Macintosh program can read data in a variety of ways. Like DOS programs, a Mac program can be designed so that a user types in a number:



Enter percentage: 99

Note in the preceding example the presence of both a Cancel button and an OK button in the dialog box. That gives the user the option of changing his or her mind, which is another excellent example of the Macintosh philosophy that the user is the boss.

If a Mac programmer wants the user to make use of the mouse rather than the keyboard, the programmer can use radio buttons or a scale with a slider to read in a value. Here are examples of each of these methods:



Percentage
● 0%
○ 25%
○ 50%
○ 75%
○ 100%

Slide scale to a percentage

Writing text to the screen and reading data from the user are the two primary responsibilities of a non-Mac programmer. You, the challenge-loving individual that you are, have additional duties. Adding a window to your program is one of them.

Working with windows

Programs written for a DOS computer simply display text and numbers on the screen. On a Mac, everything is displayed in a window. A program that uses a window makes you, the programmer, responsible for the following:

- ✓ Opening, or *displaying*, the window.
- ✓ Drawing or writing to the window.

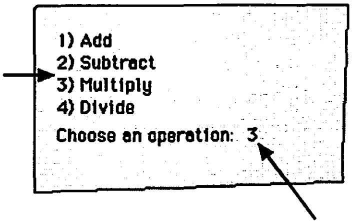
- Making it possible to move the window on the screen.
- Closing the window.

Macintosh provides the user with a myriad of options. Are you starting to get the impression that there's just too much for a Mac programmer to learn? Are you getting nervous about all of this talk about responsibility? You aren't? Good. Then skip the rest of this note. But for those of you considering giving up, I'll let the cat out of the bag and mention a topic that may provide a ray of hope — the *Toolbox*. Apple has written a ton of code for you already, code that simplifies such things as creating and moving windows and creating and displaying menus. Because these functions are used by programmers as tools to build Mac programs, Apple got cute and named the entire collection of them the Toolbox. You learn more about the Toolbox in Chapter 15.

Opening a window and writing text to it is a simple process. I know, I know — you've heard claims like this before. But in Chapter 4, I prove it. There you see the code for a Mac program that uses a window. And best of all, the code for the program fits on less than half a page! Better still, I've gone ahead and typed in all the code and put it on the CD-ROM that comes with this book. That way, as you follow along, you don't even have to type in any source code!

Menus mean choices

Another major difference between Macintosh programs and those written for older computers is the idea of pull-down menus. Mac users like to feel that they are in control of a program, rather than at the mercy of what a program allows or forces them to do. Macintosh menus enhance that feeling of control. A program that doesn't have pull-down menus may still offer some form of menu, but it's not the same. The menu choices are listed on the screen, and the user *must* choose one before the program continues, which isn't really much of a choice:



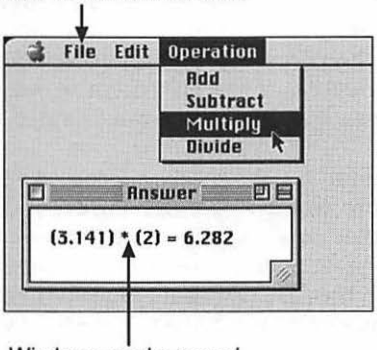
```
1) Add
2) Subtract
3) Multiply
4) Divide
Choose an operation: 3
```

A selection must be made from this window.

No other action can take place until a menu option is typed.

As you can see, this kind of menu is a stark contrast to the Macintosh way of doing things. With Macintosh menus, the user has a choice of making several, perhaps dozens, of choices. And if the user decides not to make a selection from one particular menu, she can still perform other actions. The screen doesn't freeze up and force the user to make a decision. A different menu can be selected, or a window can be moved:

Other menus can be used.



```
(3.141) * (2) = 6.282
```

Windows can be moved.

When compared to the DOS brand of menus, Macintosh pull-down menus offer a seemingly infinite variety of choices to the user. With this vast improvement, you may think that the menus represent a comparable increase in work for the Mac programmer. Think again! Macintosh menus are easy to implement and involve only a minor amount of extra work for the programmer.

Menus and windows are two of the most distinguishing features of a Macintosh program. You know they're cool because they were two of the first Macintosh features that Microsoft "borrowed" when creating Windows, its own graphical user interface. Because menus and windows are so important to Mac programs, they are also the two topics I spend the most time explaining in the remainder of this book. By the time you complete this book, you'll be able to include menus and windows in each and every Macintosh program you write. And you'll also be convinced that although Mac programming is very different (and sometimes harder) than any programming you may have done before, the resulting programs are well worth the effort.

Chapter 3

Using and Programming the iMac

In This Chapter

- Checking out what makes an iMac special
- Why the iMac's features appeal to programmers
- Learning about the processors, or CPUs, that inhabit Macintosh models
- Programming for the PowerPC processor

Chapter 2 discusses some of the key differences between Macs and PCs — but there are also differences between Macs and iMacs. That is, one model of Macintosh differs from another model. The explanation of how each Macintosh model differs from the others could generate enough material to fill an entire book — so why single out the iMac? Because the iMac represents not just a change for Apple, but also a trend. Apple broke all its previous sales records when it introduced the iMac in mid-1998, and now in 1999 there is no sign of a slow down. Not only will the iMac be enhanced and, perhaps, spawn new versions, but new Macintosh models seemingly unrelated to the iMac will share some of the iMac features. If you're lucky enough to own an iMac, this chapter will help you see how your computer fits into the scheme of programming the Mac. And if you don't own an iMac, this chapter won't be a waste — it provides a few pointers on how you should keep the iMac in mind as you write your Macintosh programs.

iMac Features

The translucent Bondi blue (that's a blue-green to those who don't think in the same cool terms as Apple) case that encloses the iMac is innovative and interesting. But there's more to the iMac's magic than just good looks. Here are a few of the features standard with each iMac:

- 233 MHz or 266 MHz PowerPC G3 processor
- 66 MHz system bus
- 512K of backside Level 2 cache

- 15-inch (13.8-inch viewable) high quality display
- Built-in stereo speakers with SRS sound

An impressive list — even (or perhaps, especially) if you don't know what half of the terms mean! Here's a quick look at what the preceding points refer to — and how they might affect someone who writes programs that will run on a Macintosh computer.

Processing power

A computer's *microprocessor* (or *central processing unit,* or *CPU,* or *processor*) is a microchip that essentially runs the computer. A program — any program — consists of code. Code is nothing more than a set of instructions that tells the computer what to do: start running, open a window, add some numbers, and so forth. These instructions are handled, or processed, by the computer's microprocessor.

TIP

In this book you're going to learn all about code. Fortunately, you'll be able to learn about it from a human's perspective — as opposed to a microprocessor's perspective. Don't worry — you won't have to know the details of how a microprocessor does its thing.

As you might imagine, the faster a microprocessor works, the faster things happen. That's why a geek spends so much time boasting about the speed of the processor in his computer. Processor speed is stated in *megahertz (MHz).* Computer manufacturers are in constant competition to up the speed of their computers, so if you follow the computer ads, you'll notice that this number is going up and up. A couple of years back a computer with a 100 MHz processor would have been considered pretty fast. Now, 100 MHz is considered kid's stuff. The original iMac computers shipped with a 233 MHz processor, and the second "wave" of iMac's came with an even faster 266 MHz processor. That's pretty fast for an inexpensive machine.

Hundreds of thousands (and, perhaps soon, millions) of iMac users have a fast computer. What does that mean to a computer programmer? It means a big market is out there that's guaranteed to be able to run programs that depend on speed. So, for instance, a person or company designing a complex, high-speed game that's to run on Macintosh computers knows for a fact that there's a big potential audience for their game. No matter how many big, galactic-crossing, giant space ships are to go careening across the screen at incredible speeds, the iMac will be able to keep up with the action.

More stuff that makes it fast

The 233 MHz or 266 MHz processor driving the iMac is the chief reason the iMac is a fast computer. But there are a couple of other reasons, too.

A computer's processor is the workhorse of the computer, but it doesn't work alone. One of the other components the processor deals with is memory. The time it takes to transfer information from the processor to memory and back again can slow down a computer. So computer manufacturers have come up with an extremely fast type of memory called *cache* (pronounced *cash*) memory. Another component that aids in speeding up the transfer of information is the speed at which the system bus operates. In short, the system bus is the pathway that information travels as it goes from the processor to memory and vice versa.

Okay, that all sounds like very heady stuff. You probably don't want any more details (which is good, because I don't *know* any more) — you probably would rather just know what it all boils down to. Here it is. The iMac has 512K (one half of one megabyte) of cache memory and a 66 MHz system bus. Boiled down further (is it getting hot in here, or is that just me?), that amount of cache memory is good and that system bus speed is good. Together with the fast processor that you just read about, the iMac is one wickedly fast machine. And again, that's something that programmers like to hear. They want to know that fast machines are on the market so that their programs will be running to their full potential.

Looks nice, sounds nice

Remember a few years back when the term *multimedia* was all the rage? The use of that word has faded a bit of late. Multimedia — the convergence of different types of media such as text, graphics, sound, and animation — itself is still big, though. So why isn't the word itself so big anymore? Probably because computers that are adept at working with different types of media aren't a rarity anymore. Anyone who buys a computer today fully expects that computer to display sharp images, play video clips, and blast sound out of built-in speakers. And someone who buys an iMac won't be let down.

The iMac's built-in 15-inch monitor is a high-quality display with minimal flicker and clear-focus. Its resolution — the number of pixels, or dots that cover the screen — can be set to as high as 1024 pixels horizontally and 768 pixels vertically. If my math is correct (you can double-check and then grade me by multiplying 1024 by 768), that means there are over three-quarters of a *million* little dots on the screen of one iMac. Having so many dots, or pixels, on a monitor means that the monitor can display a lot of detail. And that means text, pictures, animation, and video all look good. If you write a program that will run on an iMac, you know that the user will see just what you intend that person to see.

TIP

Chapter 16 discusses graphics and Mac programming. Chapter 21 shows you how to include a digitized photo in your own Mac program.

Just about every computer sold today comes with speakers. But that doesn't mean every computer sounds good. The iMac *does* sound good. The iMac has two speakers built into the front of the computer. As important as the quality of the speakers is the "behind-the-scenes" technology that allows a Mac program to get quality sound from the program to the speakers. On the iMac, that technology is *Sound Retrieval System (SRS)* stereo sound. SRS, which is licensed from SRS Labs, is a technology that manipulates sound to bring it into the three-dimensional zone that is akin to the sounds we hear when actually listening to "live" sounds. When you write a Mac program that includes sound, you know that iMac users will hear the sound just as you programmed it to play.

I'll let the cat out of the bag (or perhaps I should say, I'll let the cow out of the barn) here by mentioning that Chapter 21 shows you how to include a digitized sound in your own Mac program.

Programming the PowerPC

If your car's engine goes kaput, you could replace the entire engine. But not with just any motor. Automobile engines aren't interchangeable. The same is true of a computer's processor. Like a car's engine, a computer's processor is the "main" component of the machine. Several years back, all Macintosh computers had a processor made by Motorola. Different Motorola processor models were available, but their names all started with "68," as in the Motorola 68000, the Motorola 68030, and the Motorola 68040. Besides all starting with "68," each model number was five digits, so each was said to be in the 68-thousand family of processors. Because in the computer world, K is often used to denote thousand, the Macintosh was said to use the 68K family of processors. About five years ago, that changed.

In 1994, Apple released the first Macintosh computer *not* based on a 68K processor. Instead, this new Mac came equipped with a processor from a different family of Motorola processors — the PowerPC family. In the last five years Apple has been making a transition from the 68K family to the PowerPC family. Now, *all* new Macs are based on processors from the PowerPC family.

Fortunately for us Mac programmers, very little of our programming efforts are spent on trying to determine whether the Macs our programs will be running on sport a processor from the 68K family or the PowerPC family. In fact, for the relatively simple programs described in this book, you need to know *nothing* about the processors that run the Macs on which your program might run. Only after you move out of the realm of the easy stuff does this become a concern. So why mention it at all? Because the concern is sufficiently large enough that you should be forewarned. As you make the leap from beginner to intermediate programmer, this business of which type of processor a Mac has does become important. If you complete this book and are considering getting a more advanced Mac programming book, look for a PowerPC chapter or appendix in that book.

Chapter 4

Removing the Fear, Part I: Don't Let Mac Programmers Scare You!

In This Chapter

- How some programmers get their jollies from scaring newbies, and how you can get your knees to stop shaking
- How source code is used to create a program
- Why you need to know the rules of the C language
- How source code terminology isn't that scary
- How compiling turns source code into a program

Many Macintosh programmers, the majority, in fact, are helpful and considerate when they recognize a newcomer in their midst. Others, well . . . let's just say they aren't quite as friendly. You're the new kid on the block, and they're going to make sure you know it.

Some programmers enjoy the almost god-like status they hold over mere mortals, those lowly nonprogrammers. Why wouldn't they want to keep that power? Using an incredibly complex vocabulary is one way of locking the door that separates them from the nonprogrammers. In this chapter, I unlock some of those doors by defining some programming terms in words that you can relate to. Terms like *source code, function, compiler,* and a few others. Removing the pomp from these words should put you at ease and keep your knees from shaking every time you hear a Mac programmer talking around the water cooler at lunch time.

Demystifying Source Code

In Chapter 2, you were introduced to *code*. As you'll see a little later, there are different types of code. When code is mentioned, though, it's *source code* that is usually the type of code being referred to. Source code is a general term for one or more commands in a computer language — any language. Does *source code* still sound a little scary? Let me further demystify this treacherous-sounding term. The most intimidating thing about source code may be the last half of its name, the ominous-sounding *code*. The word generally implies secrets, symbols, top-level intelligence officials, charges of treason, and your subsequent execution. So why *wouldn't* the word intimidate you? To lessen your fear, try thinking about *code* more along the lines of the definition given in *Webster's New World Dictionary*. Webster says a code is a system of symbols in which letters, figures, and so on are given certain meanings. Now that's not so bad, is it?

I could talk about source code in general terms until I am blue in the face, but you need something concrete to grab on to. Something to do with Mac programming. Take a look at a piece of real-life Mac source code: the `int`. Recall from this book's introduction that source code, such as the word `int`, appears in its own typeface that makes it easy to differentiate from the rest of the words in a paragraph.

In the field of mathematics, the numbers used in counting are called whole numbers. These are numbers such as 0, 1, 2, 3, and so on. Whole numbers never have a decimal point. You may already be aware that whole numbers can also be called *integers*.

Many people, programmers in particular, prefer *integer* to the words *whole number*. Now, the word *integer* isn't a terribly long word, is it? So you may think that there is no need to abbreviate it. Wrong. While programmers eventually become pretty good typists, they prefer to spend their time doing other things (like thinking about what to type next). As a result, in the programming world, abbreviations are everywhere. It turns out that the word *integer* is shortened to *int*. So now when you see `int` in Mac source code, you'll know what it means and that it stands for something.

While the use of *int* for *integer* may seem like just an abbreviation to you, it can be thought of as something else. *int* stands for something — it symbolizes the word *integer*. Now recall Mr. Webster's words regarding the definition of the word *code*. A code is a system of symbols that have certain established meanings. So there you pretty much have it — source code is a set of abbreviations. There's nothing sneaky or devious about it.

Years ago, when programming languages were being developed, computer memory and computer disk space were both expensive. Back then, programmers were very concerned about these factors. Some computer languages initially allowed for a symbol to be very limited in size — that's why you find many C symbols that are a few characters in length, such as the symbol *int*.

Playing by the rules of the game

Only one of the following four sentences is grammatically correct. See if you can guess which one it is:

The man is tall. ☺

The man are tall. ☹

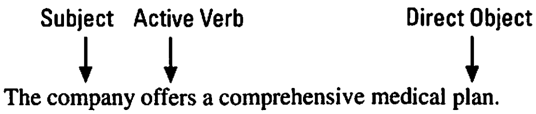
The man tall is. ☹

Tall the man is. ☹

The first one — very good! And without a hint, yet! Now, why aren't the other three sentences correct? Because the English language has a set of rules that guides you on how to create sentences, and the last three sentences violate one or more of these rules.

A computer language is similar to a spoken and written language such as English in that it too has rules. It has fixed rules that regulate how you can piece together the symbols that make up the language. Take the source code `int` that you learned about in the previous section. When writing source code, you can't just take `int` and haphazardly scatter it about. It is used only in certain well-defined instances. (I cover those instances in future chapters when I discuss the C language in detail. Chapters 13 and 14 in particular address the proper use of `int`.)

Now back to this rules business. The English language has plenty of rules. To get the following example, I first waited until my family was asleep. Then I peeked at my son's eighth-grade grammar book and found this:



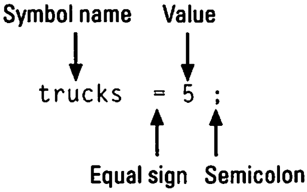
The company offers a comprehensive medical plan.

Why did I wait until everyone was asleep? So I'd be spared the embarrassment of getting grilled about why I even needed to look in an eighth-grader's book to find an example! That's right. I don't remember the rules of grammar! The book said that the above example has something to do with subject-verb agreement, in case you're wondering.

Source code, like the English language, has several rules. To make this perfectly clear, I want to move away from theory and on to a real example.

Say you're writing a program for a car dealership, and the dealer wants to keep track of the number of trucks he has on the lot. You can create a symbol that helps your program keep track of the number of trucks. What should you call the symbol? Perhaps simply `trucks`. Not incredibly clever, but it does the trick.

Now further suppose that the dealership has five trucks. You want to somehow relate the number 5 to the symbol `trucks`. You do so using the following rule:

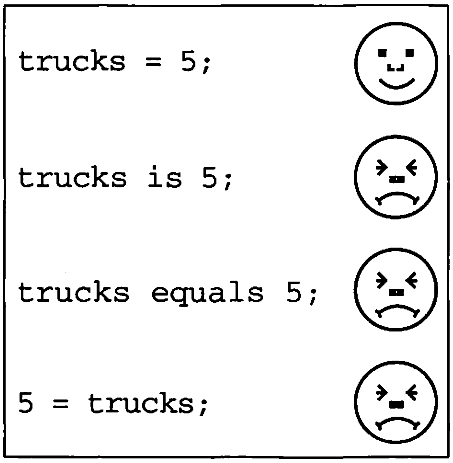


```
trucks = 5 ;
```

This figure tells you that to give a symbol a value, you first list the symbol's name, followed by the equal sign. After that, you list the value that is to be associated with the symbol. Finally, you end it all with a semicolon.

Up to now I've been using the word *symbol* to describe `trucks`. That's pretty descriptive, because the word `trucks` is being used to symbolize real trucks. In programmer parlance, though, such a symbol is called a *variable*. For now, it's enough to know that such a symbol is called a variable because its value can vary, or be changed. Chapter 13 gives you the low-down on variables.

To further compare the rules of the English language and the rules of programming source code, I offer another figure. Remember the figure with the tall man that started off this section? It's time to play again. Which of the four bits of source code associates the number five with the symbol `trucks`?



```
trucks = 5;

trucks is 5;

trucks equals 5;

5 = trucks;
```

Whether or not the preceding bit of source code makes perfect sense to you isn't the issue here. If you've ever looked at a computer programming book, with its pages and pages of source code, you've probably felt overwhelmed. Try as you might, you couldn't understand a bit of it. Stop and think for a moment. Were your expectations realistic? Just because many of the individual words in the source code looked English-like, did you really think you should be able to understand it? Especially when you didn't know any of the rules?

Once you know the rules of the game, the game becomes much easier to play. Do I feel strongly about that statement? You be the judge:

Once you know the rules of the game...
the game becomes much easier to play.

Right now, you don't know all, or perhaps any, of the rules. The chapters in Part IV lay out the rules for you by examining and explaining the C language. If you read and understand the words on this page, you are fully capable of understanding how to write source code. Stop doubting yourself! And stop letting those few nasty Mac programmers scare you about source code!

Decoding some source code terminology

While I'm on the subject of source code, I want to mention some of the many words used to describe it so that they too will seem a little less frightening.

The following little gem of source code is from the previous section:

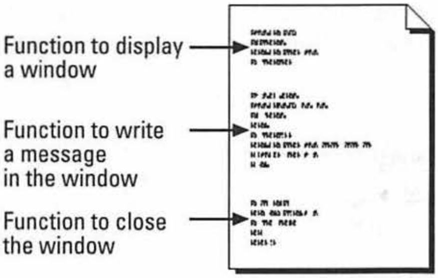
```
trucks = 5;
```

This line contains a single command. It tells the program to assign the number 5 to the symbol named `trucks`. In programming, a single command such as this is called an *instruction,* or *statement*. While statement is the preferred term, you can use the two words interchangeably.

Getting a grip on source code organization

Each single statement usually appears on a single line. But it doesn't have to. It's done this way for clarity. Placing two statements on two separate lines makes it clear that two separate commands are taking place. And it's another way of making source code a little less imposing.

Though source code may look about as organized as the contents of a bowl of alphabet soup, this isn't the case. Statements are written to carry out specific tasks. The instructions necessary to display a window on the screen provide a good example. Rather than scatter these instructions all about, the programmer places them together. When a number of instructions that perform a single task are grouped together, the result is called a *routine*. Because a routine has a single purpose, or function, it is sometimes called just that — a *function*. Routines, or functions, keep source code looking nice and tidy (to the eyes of the programmer, anyway!). The following figure, while sparing you the details of the code itself, gives you an idea of how source code can be divided into functions:


Function to display a window
Function to write a message in the window
Function to close the window

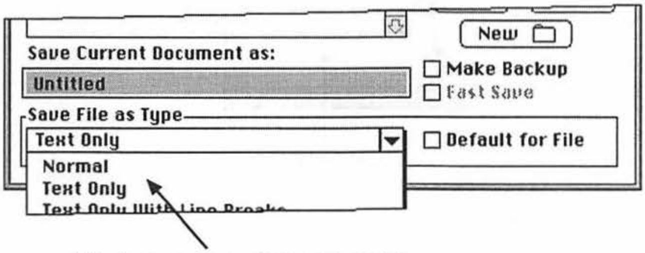
When several functions are grouped together, the result is a *program*. A program can consist of just a few functions, or hundreds. Don't worry. The programs in this book contain just a single function.

Eliminating Anxiety over Saving and Compiling Your Code

Besides source code, another subject that may send a shiver down your spine is *compiling*. As is often the case, the word itself is much scarier than actually doing it. Read on to learn why some elements of compiling source code may already be familiar to you.

Source code is nothing but text

In the previous section, you typed several commands (statements) to create source code. You save your source code work exactly as you save anything else you type — by saving it to a text file. If saving something to a text file sounds familiar, it should. This is the same type of file that all word processors can create. Word processors, like Microsoft Word, provide a menu of file formats. Part of that menu is shown here:


Save Current Document as: Untitled

Save File as Type: Text Only

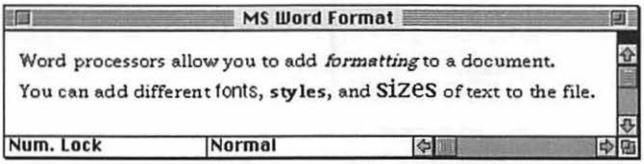
□ Make Backup

□ Fast Save

□ Default for File

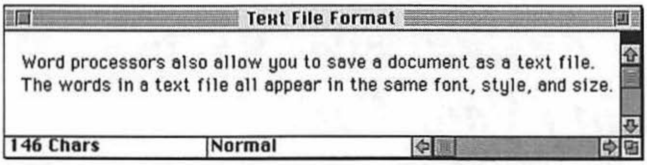
Word processors give you a menu of formats to which you can save a file.

Normally, a word processor saves a file in its own special format that saves all of the text along with all the formatting, such as italics, bold, or multiple fonts:



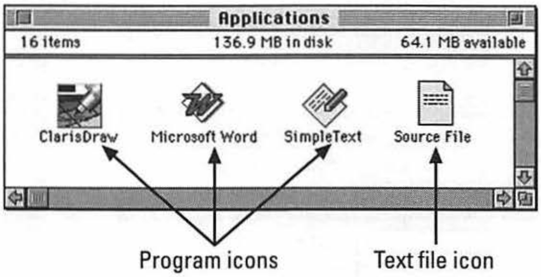
Word processors allow you to add *formatting* to a document.
You can add different fonts, **styles**, and SIZES of text to the file.

When you save a document as a text file, only the text itself is saved. If you underlined words, or made any other style changes, this information is lost.



Word processors also allow you to save a document as a text file.
The words in a text file all appear in the same font, style, and size.

You may ask yourself what ordinary text files have to do with programming your Mac. Patience, patience! You know exactly what a text file is, and that word processors can create them. You also know that source code is saved as a text file. Here's the payoff: This tells you that you can create a source code file using a word processor. Say you created a source code file using Apple's popular SimpleText program, which is a text editor that Apple distributes freely. When you quit SimpleText, you see the icon for SimpleText and any other programs you have visible on your desktop, plus a brand-new icon for the source code file. Your desktop may look a little like this:



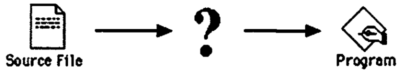
Program icons

Text file icon

Every program has an icon; it's what you double-click to run the program. Every file, such as a text file, has an icon, too. Double-clicking a program icon runs that program. Double-clicking a text file runs the word processor that created the text file. So what happens when you double-click the source code file? The word processor that created it runs, and a window containing the contents of the source code file opens. If you were hoping that the code you typed would run like a program runs, you must be disappointed. Don't get too downhearted, though. The solution is just around the corner.

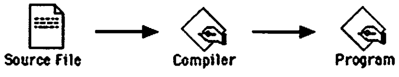
Completing the picture with compiling

Your source code file is nothing more than a text file with words in it. Even though it contains source code written in a programming language, the Macintosh views it as nothing more than a normal text file. You want your source code to become a program, and so something is obviously missing:



How do you get your typed-in source code saved in a text file to become an actual program? After creating the source code, you need to *compile* it. It's the compiler that turns the source code words into numbers. The compiler is to source code what the phone booth is to Superman — your text file may enter the compiler an ordinary text file, but it comes out a mighty, crime-fighting, superhero of a program.

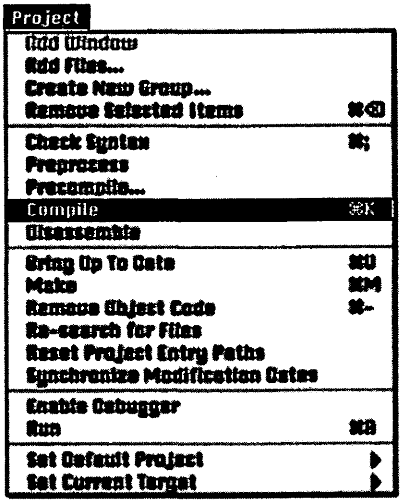
Words mean nothing to your Mac, but numbers it loves. The hundreds, thousands, or in some cases even millions of numbers that the compiler generates are what make up a program. Before going on, I want to complete the picture:



How on earth does the compiler know what numbers to create? Who cares! Whether it's using complex mathematical formulas or voodoo, it's obviously doing something very right. Why question it? What you do need to know is how to go about compiling your source code.

If you've programmed before, you may be familiar with *linking*. If you haven't programmed before, let me explain. On some computers, you have two steps involved in converting source code into a program. First you compile the source code, and then you link it. It's that way on a Macintosh, too. But Mac compilers combine these steps so the part about linking seems invisible to you.

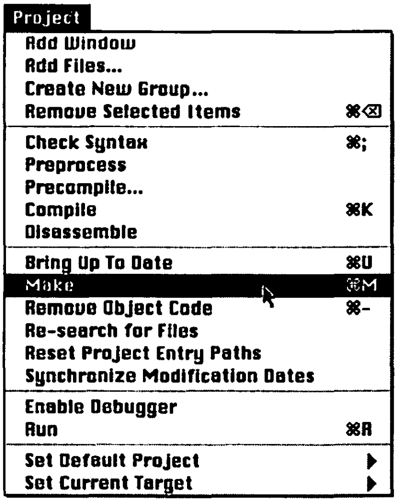
After source code is typed into a file, you compile the file to turn it into a program. I use the CodeWarrior compiler in this book, so to compile my file, I choose the Compile menu item from CodeWarrior's Project menu. Don't worry about not knowing what a Project menu is. The important thing is to know that compiling your source code is a one-step process. That's all there is to it. Here's a look at the CodeWarrior Project menu:



Sorry about blurring that figure, but I did it because I don't want you to get worried about the host of other items in the menu. It's just as well if you don't even know what they are; you won't need these other items while you're learning the basics of Mac programming. The Project menu contains several items, but Compile is the item you use the most. In fact, even when I cover the CodeWarrior compiler in depth in Part III, I only mention a few of the other items in this menu. That's how simple compiling really is.

The Compile menu item compiles a source code file, but even after compiling it, the source code file still isn't a true, ready-to-use program. So why do it? To make sure that your source code is all correct. You and I never make mistakes, of course, but some people do. When the compiler encounters a line of source code that it doesn't think is right, it lets you know. After you correct the offending bit of code, you compile the source code file again. After everything's fine with the source code, you can make a program out of it.

A software program is often called an *application*. If you're creating your own program, or application, you can say that you're *making* it. In fact, that's exactly what you say. If you look in the Project menu of CodeWarrior, you see that CodeWarrior provides a menu item called Make — the next figure shows that menu item. I won't repeat my cheap theatrical stunt of blurring the menu here. Instead, I trust that you won't become engrossed in, or intimidated by, the other menu items in the Project menu:



The Make item sneaks a little code of its own into your code. It does this so that when you quit the compiler, a brand-new icon representing your program appears on the desktop. If you *haven't* yet compiled your source code file when you choose the Make menu item, Make first completes that task. If you *have* compiled your source code, Make jumps right to the part about adding its own code.

Different compilers give you some options

Different software companies make different C language compilers. For instance, Apple makes one called MPW, which stands for *Macintosh Programmers Workshop*. MPW is a product designed specifically for professional Macintosh software developers. Another company, Metrowerks, makes a Mac compiler called Metrowerks CodeWarrior Professional, or simply CodeWarrior for short. CodeWarrior is used by many professional programmers — but it's also used by almost all people new to programming the Mac. That's because CodeWarrior has a very friendly interface — CodeWarrior makes it easy to type in your source code and compile that code into a program. Because of this, in the five years or so that CodeWarrior has been available, it has become far and away the leading compiler among Mac programmers. For these reasons, IDG Books and I decided upon CodeWarrior as the compiler to use for this book's examples. You'll find a limited version of the compiler on the CD-ROM that comes with this book. I cover CodeWarrior in detail in Part III. For now, I'll continue demystifying the process of compiling by giving you a very quick look at the CodeWarrior compiler in this section.

The Compile menu item compiles source code, but it doesn't turn the results into a program. The Make menu item both compiles source code and turns the compiled results into a program. So why would you ever want to use the Compile option when the Make option obviously does more? Despite the best planning, writing source code is a trial-and-error process. You'll write code, compile it, and then, if you've made any mistakes, you'll correct them and compile again. During this process, there's no need to turn the compiled code into a program. That process, which takes a little more time than simply compiling, can wait until you're sure that you have things right.

The menu I just showed has about a dozen items in it. I said that you only use a few of the items from this menu. That, of course, begs the obvious question: What are all those other menu items *doing* there? Remember, I didn't say *no one* would ever use these items. I just said that *you* don't need to use them. More experienced programmers take advantage of some or all of the features these items provide.

One last point about compilers, and this is a really neat point. The compiler features a built-in text editor. A text editor is like a word processor, except that it offers limited text formatting options. For instance, with a text editor you can't display a fancy ruler at the top of the window that allows you to change the indentation of different paragraphs. The primary purpose of a text editor is to — you guessed it — save text to a file. If you want to write a fancy report and include figures and headings and the like in it, then a text editor *isn't* for you. If you want to write source code, then a text editor *is* for you. A text editor is perfect for writing source code because text is all that a compiler understands — it doesn't give a hoot about the fancy formatting writers of other stuff include in their manuscripts, papers, and reports.

What's so neat about having a built-in text editor? It makes creating a program that much easier (and that much less frightening). With the built-in text editor, here are the steps you perform to create a Mac program:

1. **Run the compiler program (CodeWarrior in this case).**
2. **Type your source code using the built-in text editor.**
3. **Choose Run from the Project menu.**
4. **Quit the compiler program.**
5. **Brag to all your friends that you just created a Macintosh program.**

Well, maybe it won't go *quite* that smoothly, but that's a pretty darned good assessment. In any case, it's plain to see that creating a Macintosh program isn't nearly the nightmare-of-an-experience that some people may have led you to believe!

Chapter 5

Removing the Fear, Part II: The One-Minute Program

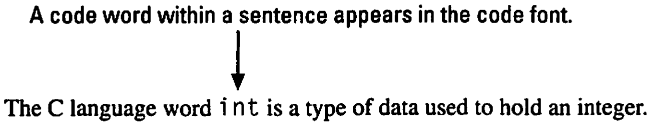
In This Chapter

- Looking at source code for an honest-to-goodness Mac program
- Discovering how a program opens a window
- Finding out how a program writes words to a window
- Using the Toolbox (free code from Apple)

In Chapter 4, you saw a little code (very little). In this chapter, you see just a little more — about a dozen lines. Source code is often intimidating because you usually see pages and pages of it. Would it remove some of the fear if I told you that the few lines of code in this chapter comprise an entire Macintosh program? It's true. As you can well imagine, Macintosh programs get much larger. But isn't it nice to know that the first one you'll be exposed to practically fits in the palm of your hand?

Remembering Those Conventions

Because you're about to look at some source code for a real-life Mac program, let me take just a moment to remind you about the conventions used in this book. Any source code that appears on the pages of this book looks different from the rest of the text. To help you quickly identify what is source code and what is regular text, all code that is mixed in with text appears in a special font:



The C language word `int` is a type of data used to hold an integer.

Other times you see a block of code. Some examples work best when you view several lines of code listed together:

```
int   trucks = 0;
int   i;

for (i=1; i<10; i++)
   trucks++;
```

Entire section appears in the code font.

A block of code is called a code *snippet.* That's because it isn't an entire program; it's just a part *snipped* from a complete program.

That's It? That's a Mac Program?

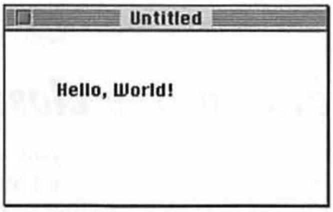
I can write a Mac program in 15 lines of code or less. Sounds kind of like *Name That Tune* doesn't it? It's true, though. I won't explain all of the code in detail; exactly how the code works isn't important here. What is important is that you realize these few unimposing lines of text are capable of opening a window and writing words into it. It runs on a Mac; it opens a window; it uses the window. Why, I believe that qualifies as a Macintosh program! Granted, it's a simple program. Still, this brief look at Mac source code should do much to eliminate any anxiety you may have about writing code.

Unveiling the program

Here, in its entirety, is the source code for the much-heralded example:

```
void  main( void )
{
   WindowPtr  theWindow;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   SetPort( theWindow );

   MoveTo( 30, 50 );
   DrawString( "\pHello, World!" );

   while ( !Button() )
       ;
}
```

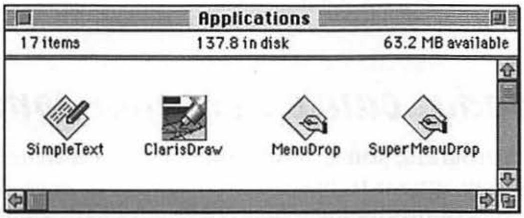
What incredible things does the program do? After you compile the code, it displays a window and writes the words *Hello, World!* in it. When you click the mouse, the window disappears and the program ends. Here's what you see when you run the program:



```
Hello, World!
```

Naming the program

Notice that it appears that the program has no name. That's because a Mac program is named *after* it has been compiled. In fact, you can even change the name of a program after it's all done and sitting on your desktop. Simply click the program's name and type in a new one. For example, I made a copy of a program that appears in Chapter 18, MenuDrop, and then renamed it SuperMenuDrop. The result is shown here:



Software companies don't think it's funny if users copy and rename programs. After all, the software companies spend a great deal of time and money to develop their programs. And companies would see even less humor in such antics if attempts were then made to distribute these newly named programs. If you tried this, not only would you have the dreaded Software Police after you, you'd also probably have federal authorities knocking on your door. With that said, why was it okay for me to rename the MenuDrop program to SuperMenuDrop a little earlier? Because I wrote that program, and I can give *my own* programs any name I want!

It's tedious and impersonal to keep calling my example program *my example program shown near the start of this chapter.* Even though it's just a simple Mac program, it still deserves a name of its own. Macintosh programmers take a lot of pride in their work and thus put a lot of time and effort into thinking of catchy, original names for their programs. With that in mind, I'll call my program ExampleOne.

Examining the code, but not too closely

If you're interested in ripping ExampleOne apart line by line, you have to wait until Chapter 17. In this chapter, I cover some of the statements in the ExampleOne program, but only at a superficial level, again just to get you acquainted with code.

Because you're probably not familiar with the C language, I won't mention too many specifics of the program in this chapter. But even without a knowledge of the particulars of C, you can still get a lot out of ExampleOne. Here's what you can learn from a brief look at a very short program:

- ✓ Source code really does have structure and organization.
- ✓ At first glance, source code appears to be quite cryptic, but it really isn't.
- ✓ Everyone is capable of writing a simple Mac program.

Getting it ready, cause here you come

When you write a Mac program, some of the commands you write are very short and simple. Yet they appear to make a lot of things happen. If you're like most people, the idea of producing a heap of results with a minimal amount of work sounds pretty good.

You get a lot of work out of these short commands because you aren't going at it alone. Before you start looking over your shoulder, let me explain. Apple has written a ton of source code for you already that's buried deep inside your computer. Some of the instructions you write activate this code and make it available for your program. I'll have more to say about how this process works throughout the book, including later in this chapter (see "That's It . . . But Don't Forget the Toolbox!").

One of the first things you do when you write a Mac program is let the Apple-written code that dwells in your Mac know that you are going to make use of it. Think of it as telling the Macintosh to get ready, 'cause here you come.

This process is called *initialization*. The ExampleOne program uses three separate statements to initialize the Mac:

```
InitGraf( &qd.thePort );
InitFonts();
InitWindows();
```

ExampleOne writes text to a window. On a Macintosh, the typeface in which a text is written is called a *font*. The `InitFonts` instruction warns the Mac that this program writes to a window, and thus uses a font. The Macintosh then does some internal hocus-pocus that gets itself all prepared to write text.

From the preceding paragraph, you can see that when a command starts with `Init`, the `Init` means initialize. The second half of the word tells what gets initialized. Pretty intuitive, isn't it? `InitFonts` initializes fonts. `InitWindows` initializes windows. `InitGraf` initializes . . . well, maybe that's not quite so intuitive. More on `InitGraf` later in Chapter 17.

Every Macintosh program has at least a few of these instructions that begin with `Init`, so you are doing well to get acquainted with them now.

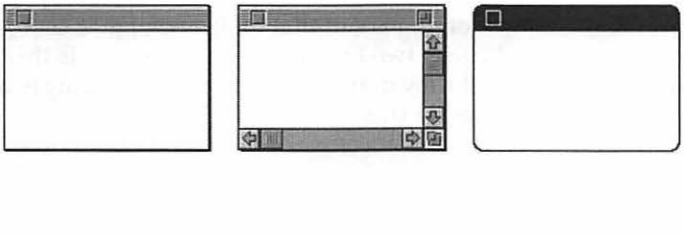
Opening a window

The ExampleOne program displays a window. This line of code prepares the Macintosh to do that:

```
WindowPtr  theWindow;
```

That's a little bit like the initialization I just talked about in the previous section, but not exactly. Similar, but different. Now isn't that helpful? Let me try again. This line of code lets the Mac know that a window is going to be created and displayed at some point in the program. Knowing that, the Mac makes sure there's a little space reserved somewhere in memory to hold information about this soon-to-arrive window.

What kind of information about a window does the Mac have to keep track of? For one, the type of the window that opens. Remember, windows don't all look the same:



Now that you've got the Mac all excited about the idea of a new window coming into existence, you certainly don't want to let it down. The following line of code actually creates a window:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

The GetNewWindow part of that last line sounds straightforward enough. But where is the program getting the new window? And what's all that business between the parentheses that follows GetNewWindow? Not so fast! Remember, this is an overview. I don't have to tell you everything right here and now!

Notice the return of theWindow. A moment ago, I said that theWindow would be used to hold information about a window. Take another look at this line of code:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

It serves two purposes. First, it displays a new window. Second, it gives theWindow the information it needs about the new window. Actually, this single line of code encompasses about four or five separate programming topics. I think I better stop right now before I get in too deep! Don't worry, you hear more about theWindow later in this book.

Writing to a window

After displaying a window on the screen, ExampleOne writes a few words in it. The line of code that writes these words is:

```
DrawString( "\pHello, World!" );
```

While you may not know what a string is, you certainly know what the word *draw* means. The Macintosh, being the very graphical kind of computer that it is, doesn't simply write text. No, the Mac considers writing a very dull sport. It much prefers to *draw* things. Not just lines and shapes, but even words.

By the way, a *string* is a group of characters — letters, digits, and so on. Thus the word *dog*, the sentence "Hey, you!" and the nonsense Ab123&%* are all strings. You get a more formal definition of strings in Part V.

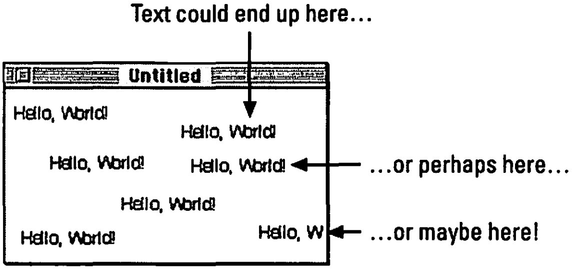
You're probably also wondering about that funky-looking \p that precedes the words *Hello, World!* Those two characters always precede the text, or string, in DrawString, but they don't show up when the string is written in the window. As usual, more on this in Chapter 17.

What about those two lines of code that precede the DrawString line? Here they are:

```
SetPort( theWindow );
MoveTo( 30, 50 );
```

Imagine that ExampleOne displayed three windows at the same time. What do you think would happen if you write — excuse me, *draw* — some words using DrawString? Which of the three windows would the text appear in? That's a real dilemma! It's a good thing that the command SetPort solves it. I won't tell you here how it solves it, but I will tell you that it involves your old friend, theWindow.

The idea of which window to draw text in brings up a second problem — where in the proper window should the text be drawn? If you tell the Mac to draw some words in a window, where do they end up?



The line that contains MoveTo handles where text should be drawn in a window:

```
MoveTo( 30, 50 );
```

The two numbers that follow MoveTo, 30 and 50, tell the Mac exactly where in the window to draw any text you want. Chapter 16 discusses exactly how those two numbers tell the Mac where to draw.

Ending the program

Button, button, who's got the button? The mouse has the button, and clicking it ends the ExampleOne program. Here are the two lines of code that address that task:

```
while ( !Button() )
    ;
```

If these two lines of code were translated into poetry, they would read as:

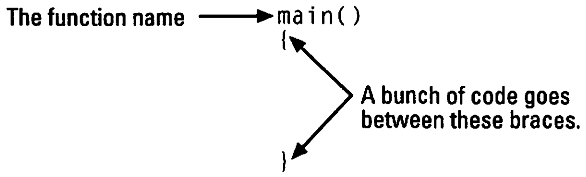
> While the mouse button is not clicked down, The program should stick around.

Shakespeare it's not, but you get the idea. When the mouse button is clicked, the program ends and the user finds himself or herself back at the desktop. A Mac program usually doesn't end when the user clicks the mouse. But I thought a simple ending would be very appropriate for this simple example.

Ending at the beginning and the end

I still haven't covered three lines of code, not to mention several blank lines. Take a look at the three lines with the two braces, { }, and `main()`.

Code can be grouped together in what are called *functions*. Most programs have several functions; ExampleOne has just a single function named *main*. How can you tell where a function begins and ends? By the braces — they mark the beginning and end of a function:



```
main()
{
}
```

How much code goes between the braces? What does the code here do? That all depends on what the function is supposed to do. In ExampleOne, the function initializes things, opens a window, and draws some text to the window. To accomplish that, I wrote ten lines of code between the braces (not counting the empty lines). Other functions do other things and could have more or fewer lines of code.

When it comes to Mac programming, blank lines don't count. In source code, you can stick a blank line anywhere and it doesn't upset things. Blank lines are also known as white space. Why insert a blank line if it doesn't do anything? To add clarity to your source code. Rather than have line after line of uninterrupted code, you can throw an empty line in to break things up. Notice that I used blank lines to group together related lines of code. An example is the three lines of code that handle initializations:

```
InitGraf( &qd.thePort );
InitFonts();
InitWindows();
```

If you look at the code for the complete program, you see that I have a blank line before and after these three lines.

That's It . . . But Don't Forget the Toolbox!

I sure hope that this rudimentary explanation for each line of code in the ExampleOne program has assuaged some of your programming fears. If not, don't worry — there's still time to get the hang of it. The chapter is complete, right? Come on, you know better. As long as I have the source code right in front of you, I may as well cover a couple of general topics that apply to it.

Source code and compiling (which I discuss briefly in Chapter 4) are two of the most scary topics any new programmer can encounter. Because Mac programs are so different from programs written for other types of computers, you may have guessed that there are some semi-frightening programming topics that pertain only to the Mac. How right you are! I devote the remainder of this chapter to one of the biggest and baddest of them: the Toolbox.

Imagining the glory of the Toolbox

Imagine this: A computer company hires top-notch, professional programmers to write thousands of small, efficient programs. Miniprograms, if you will. Each miniprogram can do something exciting, useful, or both. Like put a window on the screen. Or draw a circle in a window. Or move a window.

But wait, there's more. Then this company devises a way that you, a novice programmer, can access any one of these miniprograms just by typing a single line of source code. And then — and here's the clincher — the company gives all these miniprograms away, free! Okay, you can stop imagining now. This is one of those dreams that really does come true: Apple stuffs scores of these miniprograms into a file on each Mac's hard drive (the System file in the System Folder, if you must know) and into computer chips that are then soldered right inside every single Macintosh. Not only did they do all this; they also came up with a clever name for this collection of miniprograms — the Toolbox.

Calling the Toolbox

The miniprograms that make up the Toolbox are the tools you use to build your programs. To make use of a Toolbox miniprogram, you *call* it:

No, not like that! When I say you call a Toolbox miniprogram, I mean it a little more figuratively. Your source code calls on a miniprogram to perform a task. You probably weren't aware of it, but you've already come in contact with several Toolbox calls. The ExampleOne program uses eight of them! Look again at the ExampleOne source code. I place all the calls to Toolbox miniprograms in **boldface** type so that they are easier to spot:

```
void main( void )
{
   WindowPtr  theWindow;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   SetPort( theWindow );

   MoveTo( 30, 50 );
   DrawString( "\pHello, World!" );

   while ( !Button() )
      ;
}
```

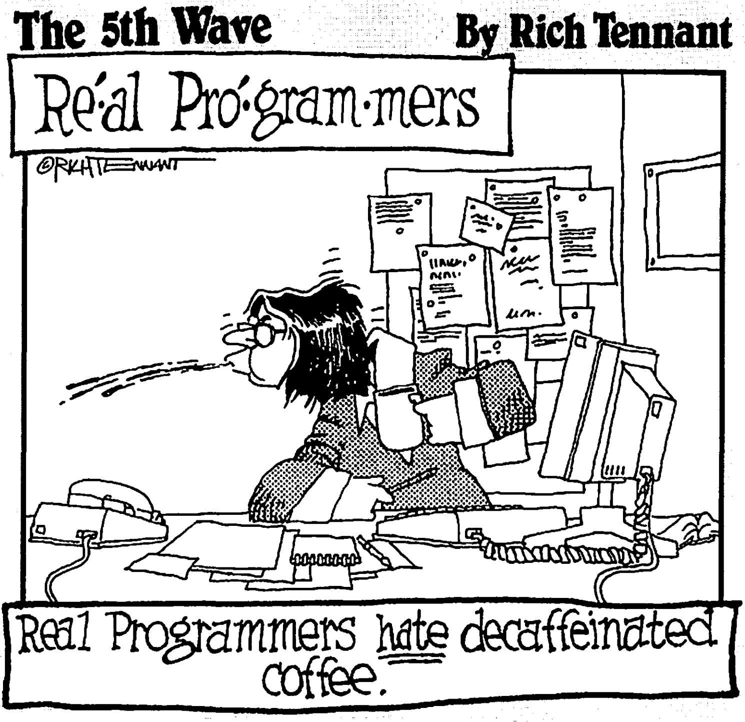
Feeling like you're not alone

You can surmise that the Toolbox is a very important part of Macintosh programming. In this section, I tell you just a little bit about what the Toolbox is. This small taste may have prompted even more questions. How can you tell what source code is a Toolbox call? How do you know when to use a Toolbox call? How do you know the names of all the Toolbox calls? Patience, patience my friend. I discuss the Toolbox in Chapter 15.

If you aren't expected to know everything about the Toolbox from this section, what *should* you know? That you are not alone in your programming endeavors. Apple support is right there beside you. In front of you, actually — right inside your Mac. The invisible code that Apple has tucked away in your Macintosh does all sorts of helpful and wonderful things for you. And, best of all, you never have to know how it works! You learn a lot more about the Toolbox in this book, but you never have to learn how each Toolbox miniprogram, or function, works. You only have to know that Toolbox functions do, in fact, work very well.

Part II

Resources: This Is Programming?



In this part . . .

The Macintosh is a fun computer to use. Now, if only it were a fun computer to program. But wait . . . it is! Though writing programs for many other kinds of computers consists only of typing in line after line of mind-numbing text and numbers, programming the Mac involves working with neat pictures that represent the windows and menus that are to appear in your program. Well, yes, it also includes a little bit of that mind-numbing business — but you won't see any of that in this part of the book. So don't worry about it just yet. Instead, enjoy the three chapters in this part. They deal with *resources* — the fun part of programming.

Chapter 6

What Are Resources?

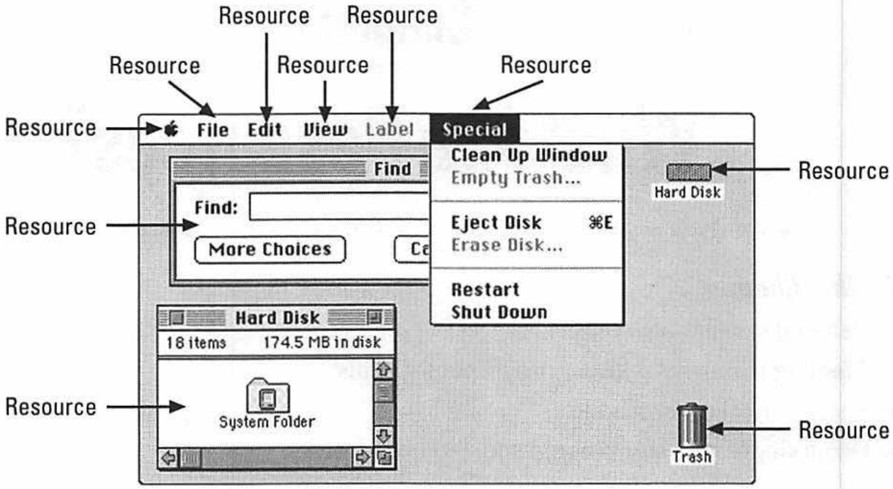
In This Chapter

- Defining the Mac's resources
- Creating resources without programming skills
- Keeping resources in a file
- Observing how resources and source code interact

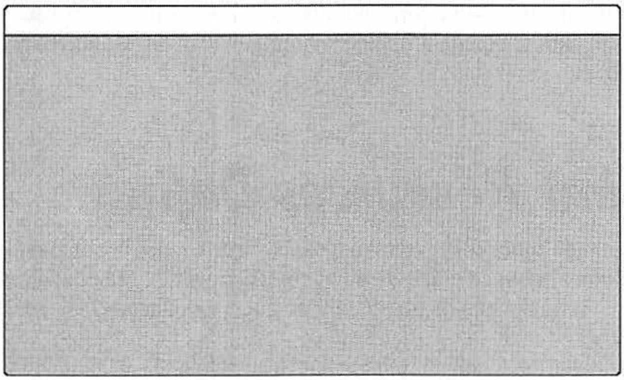
I may sound like a Boy or Girl Scout leader, but I've got to tell you that it's time to get resourceful. No, this isn't the chapter where you learn how to make a shelter out of fallen branches and stones. In this chapter, and the two following it, you learn about a topic very important to Macintosh programming — resources.

Defining What Resources Define

Different *resources* have different purposes, but for the most part resources are used to define what the different parts of the GUI (graphical user interface) look like. What parts of the Mac interface are defined by resources? Take a look:

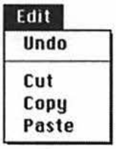


Okay then, what parts of the interface *aren't* defined by resources? Take another look:

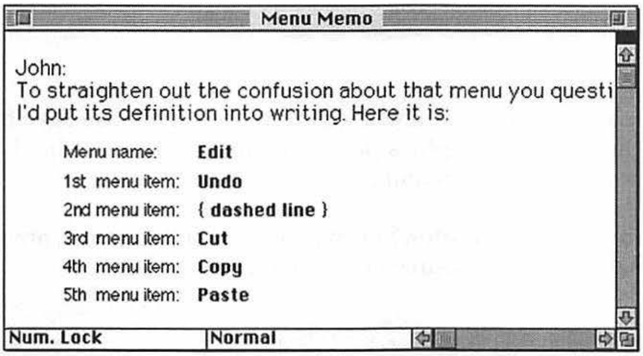


Wow! Resources are a big responsibility for the Mac programmer. Yes, resources are a key part of Mac programming. But here's some good news: Resources are also the fun part of programming. Hey, I heard that snicker! It's true though. Resources are fun because they're freebies — they involve absolutely *no* programming. You don't have to use source code or a compiler to create a resource.

Those last two figures should make it quite clear that resources are crucial to a Mac program. Without them, a Mac program would look much like a program written for another type of computer. It would consist of nothing more than words on the screen. But what do I mean when I say a resource *defines* a part of the interface? That depends on the part of the interface in question. If you were writing a Macintosh program with menus, you'd use resources to define each menu by specifying the menu's name and the names of the items that appear in the menu. Here's a typical Edit menu:



Using words, you'd define the preceding menu something like this:



John:
To straighten out the confusion about that menu you questi
I'd put its definition into writing. Here it is:

Menu name:	**Edit**
1st menu item:	**Undo**
2nd menu item:	**{ dashed line }**
3rd menu item:	**Cut**
4th menu item:	**Copy**
5th menu item:	**Paste**

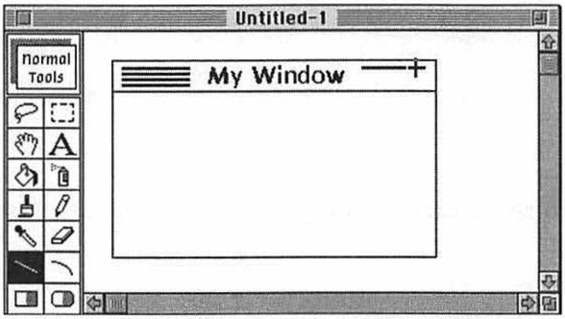
With resources, though, you don't have to type in quite as many words to define a menu. (Which is another good reason to like resources.) And when you use resources to define a menu, you get to see exactly what it looks like while you're creating the menu.

Look, Ma, No Programming!

Resources define what different parts of the interface look like. One resource may define what menu items are in a menu. A different resource may define what a certain window looks like. In any case, the act of creating the resource doesn't involve programming. How can that be, you ask? Take a gander at the following hypothetical example that shows how resources go about shaping the interface.

Hypothetically speaking about resources

Say I want to create a window. I run my graphics program and draw exactly what it should look like. Here I am in my paint program, in the middle of drawing a window:



When I have my window just how I want it to look, I choose Save from the File menu and then quit the graphics program. There — I just defined a window. And without a bit of programming.

What can I do with this window? I can make use of it anytime, anywhere, by calling the window in my source code. Kind of like this:

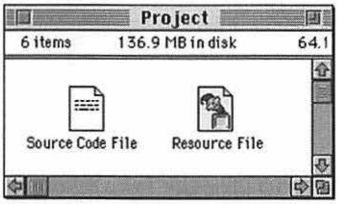
```
void  main( void )
{
   Display My Window
}
```

No, that isn't really how it's done in a real Mac program. Dirty trick? Maybe. But I did warn you beforehand that the example was going to be *hypothetical*. Anyway, what I'm trying to do is give you an idea of how something created without programming can be used in a program. While resources aren't created in a graphics program, they are created in a separate program that's as easy to use as a graphics program. And, like my teaser hypothetical example, you then make use of your window (or whatever else you've created) in your source code.

Realistically speaking about resources

Resources don't involve programming, but that's only partly true. Now wait a minute. Before you think that I lied about that no programming business, read on. Creating resources involves absolutely no programming, but getting your program to recognize and make use of resources does involve programming. So you can see that I wasn't entirely dishonest with you!

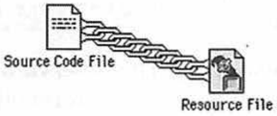
Resources are created and saved in their own special file — just like source code. Here's a folder that contains both a source code file and a resource file:



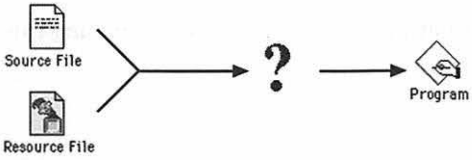
The source code file sports the familiar text file icon. What's that icon on the resource file? You have to read Chapter 7 to see why the resource file has such an interesting icon. Separating the resource file from the source code file allows you to create the resources independently from the source code. Just as I promised, creating resources does not depend on knowing or using any source code. Someone who has never programmed a line of code can create a file that holds resources.

The resource/source code connection

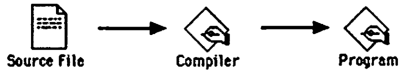
You need source code for a program to make use of resources. Because the source code and the resources exist in two separate files, something must bind them and let them work together:



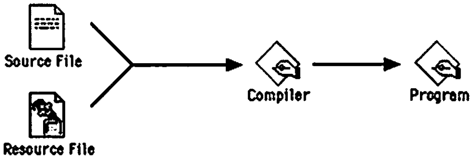
That's a little dramatic, but it does emphasize what must take place. The contents of the resource file and the contents of the source code file do, in fact, get bound together to form a program:



Yes, it's the mysterious question mark. You need something to turn source code into a program. Do you know what that something is? Here's the answer:



That's right — the compiler. It's now time to give the faithful compiler a big round of applause. Why? Because the compiler performs not one, but two really important Mac programming functions. Its first task is to turn your human-readable source code into the numbers that your computer can understand. Its second task is to merge this modified source code and the resources. The result? A Macintosh program — one with GUI components, such as menus and windows. With this new bit of knowledge, you should understand this update to the previous figure:



But How Do You Create a Resource?

In this chapter, I provide an overview of resources. This overview will be of help throughout this book, because you need to know what resources are in order to know when to use one to accomplish a particular programming task. In this chapter, I also manage to fill about a half dozen pages discussing resources. Yet I never exactly say how you create them. What can you conclude from this? That I get paid by the page, and I'm padding the book? A nice guess, but that's not the case. It's important to understand how resources fit into the grand scheme of a Mac program. That way, when you *do* see how to create a resource, you'll know why it's being made and how it's used.

So how *do* you go about creating a resource? Why, by reading the next chapter, of course.

Chapter 7

ResEdit, the Resource Editor

In This Chapter

- Creating a resource
- Naming a resource
- Identifying a resource
- Using ResEdit
- Making a resource file
- Creating and editing a menu resource

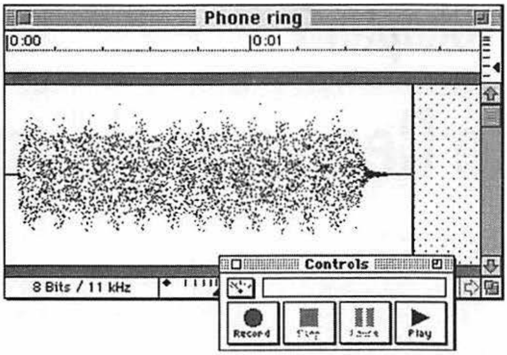
Enough chatting about resources — it's time for you to make one! This chapter shows you how to do that. You go on an in-depth tour of the process of creating a resource that can be used as a menu in any Mac program you make. And once you see how to make one resource, you'll have a pretty good idea of how to make others.

Editing — It's Not Just for Text Anymore

What does the word *edit* mean to you? According to *Webster's New World Dictionary*, to edit is to "revise and make ready a manuscript for publication." What does Webster know? Heck, he died over 150 years ago, back when a computer was a guy with a pencil and paper! In the modern Macintosh world, editing has absolutely nothing to do with manuscripts, or even words.

Forget that text!

When you see the word *editing* in a few of this chapter's headings, you may get excited. Finally, terms that sound familiar: edit, editing, editor. An editor is for editing text, right? A *text* editor is for editing text. There are other types of editors. For example, a sound editor shows a sound as a sound wave and lets you edit it, like what I'm doing right here:



What does editing sounds have to do with resources? Absolutely nothing. But it's such a cool topic, I just had to sneak it into the book somewhere. On the other hand, maybe showing a sound editor isn't so frivolous . . . it does demonstrate that text isn't the only thing you can edit.

ResEdit: One mighty resource editor

You may have already surmised that, like text and sounds, resources can be edited. And like text and sounds, you use a software program to edit resources. Apple makes a resource editor called ResEdit that does just that.

To avoid the ridicule of seasoned Mac programmers, be sure to pronounce ResEdit correctly. You say it *rez-ed-it.*

I use ResEdit for all the example programs in this book, but it isn't the only resource editor on the block. What criteria did I use to select it? Glad you asked:

- It's by far the most popular resource editor among Mac programmers.
- It's a straightforward, easy-to-use program.
- It's made by Apple, so it has to be pretty good!
- You already have it — there's a copy of ResEdit on this book's CD-ROM! To learn how to copy ResEdit from the CD-ROM to your hard drive, see Appendix E.

I thought you'd like that last point!

ResEdit isn't the only resource editor for the Mac. A company named Mathemaesthetics sells one named Resorcerer. But why pay for a resource editor when Apple gives theirs away free? As a beginning Mac programmer, there isn't any reason to. But as you get more serious about programming, it may be worth your while to invest in a resource editor that has extra features not found in ResEdit.

What's in a Name?

Resources come in different types. One type describes a menu. Another type describes a window. Each resource type has a name, of course. When you work with ResEdit, you get up close and personal with resource names, and so it's wise to spend a moment setting some ground rules for naming resources.

Don't quote me on this

Most programmers enclose a resource type in single quotation marks. That means that a window resource, a 'WIND' resource, is written as you see it here in this sentence. Another commonly used resource, the resource for a menu, is written as 'MENU'.

For any resource type, the quotations themselves are not part of the name. Why include them, then? Two reasons. First, it makes it easy to spot a resource type when it's mentioned in a body of text. Second, a resource type is *always* four characters. But in *some* resources, the fourth character is a blank space. So a sound resource, which is written as the letters *s, n, d,* followed by a space, is written as 'snd'. Using the single quotation mark reminds you that a space is included as part of the name.

A MENU is not a menu

If I write the word dog as DOG, you still recognize it as meaning a four-legged, barking pet. You may wonder *why* I capitalized all the letters in the word, but you still understand what it means. To you, dog and DOG mean the same thing. When naming resources, this same freedom to capitalize or not capitalize a word does *not* apply.

For example, note that each of the four characters in 'MENU' is an uppercase letter. That's important. When the Mac sees 'MENU', it knows you're talking about a menu resource. If it sees 'menu', it has no idea what you're talking about. If you ever see the menu resource written as 'menu', it's the typesetter's fault — not mine!

When the proper use of uppercase and lowercase in a word is important, that word is said to be *case-sensitive*. It turns out that *all* resource names are case-sensitive, not just the 'MENU' resource. Can you skip the talk about case-sensitive and just remember to always use uppercase when writing the name of a resource? I'd like to say it's as easy as that, but it's not. Some resource names *do* appear in lowercase characters, such as the 'snd' resource.

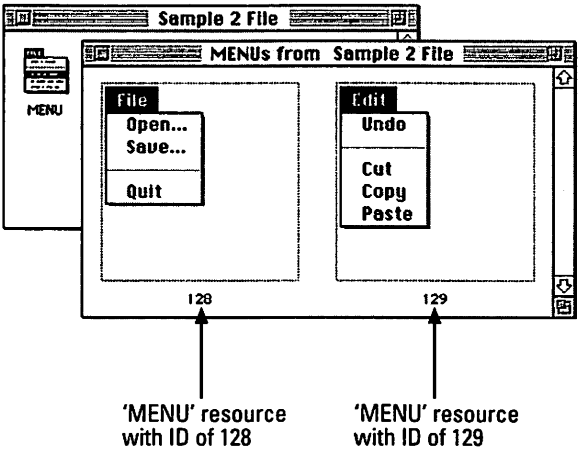
As an aside, the C language — which you'll be reading all about in Part IV — is also case-sensitive. In Chapter 5 you had a little taste of the C language. There you read about the `int`. If you tried to instead use `INT` in a C language source code file, you'd run into problems.

Resource IDs

Each and every resource has an identifying number — an ID. Why? Because resource editing with ResEdit is getting just too darned simple — it's time to toss around a few numbers just to remind you that you're programming a computer! But, of course, you know better than that. There is a very logical reason for giving each resource an identifying number.

Resources and source code work hand-in-hand to comprise a program. In your source code, you need to call and work with the resources that you want to include in the program; you call on individual resources such as a 'MENU'. When you do, you call the resource using its ID number. You can't just write source code that says, "Display the 'MENU' resource in the menu bar." There may be more than one 'MENU' resource in your resource file, and the program wouldn't know which one to use.

Here's how you can find out the ID of each 'MENU' resource in a resource file:



When a new resource is created, ResEdit assigns it an ID number. For most (but not all) resource types, ResEdit gives the first resource an ID of 128. After that, ResEdit starts numbering the resources of the same type consecutively, so the second resource of a type would be 129, the third would be 130, and so on. Don't bother wondering why — you never need to know. If you want further proof, just take a look at the preceding figure. The two 'MENU' resources are numbered 128 and 129. I rest my case.

Using the same number for two resource types can lead to real confusion for many new programmers. It's easy to see why many programmers mistakenly go to great lengths to ensure that there's no ID duplication in a resource file; they don't want the computer to use the wrong resource. Don't worry — it won't. That's because you never write source code that says something like *use resource 128.* Instead, your source code has a more explicit meaning more like *use window resource 128.* The Mac knows the difference between a 'MENU' resource with an ID of 128 and a 'WIND' resource with an ID of 128. So it's okay to have both in the same file.

What about accidentally giving two resources of the same type the same ID — such as creating two 'MENU' resources, both with an ID of 128? Again, there's no reason to worry. ResEdit won't ever assign the same ID to two resources of the same type.

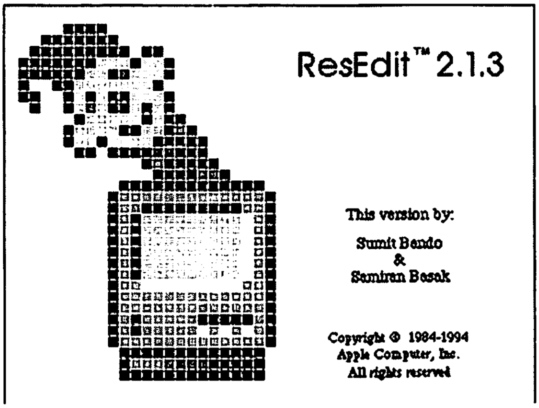
Using ResEdit

The more you program the Mac, the more you use ResEdit. Because resources are such an important part of creating a Macintosh program, using a resource editor is important, too. That makes the decision to devote an entire chapter to exploring ResEdit a very practical one.

Every Macintosh program has both a resource file and a source code file. Many programmers start a new program by creating the resource file, so that's where you should start, too. Hey, you're a Mac Programmer now — you'd better start acting like one!

Creating a resource file

To access the ResEdit program, double-click the ResEdit icon. You come face-to-face with the ResEdit Jack-in-the-Mac introductory screen:

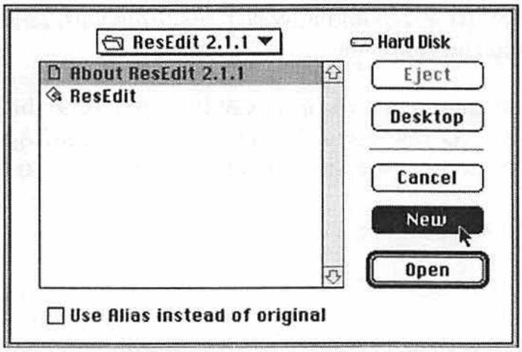


ResEdit™ 2.1.3

This version by:
Sumit Bando
&
Samiran Basak

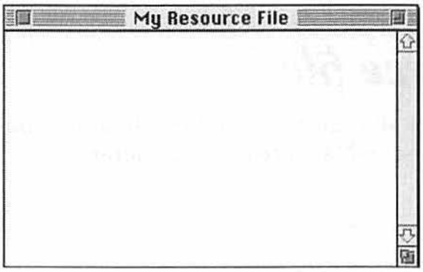
Copyright © 1984-1994
Apple Computer, Inc.
All rights reserved

After staring at the introductory dialog box for a while, click the mouse to continue. ResEdit automatically displays a dialog box that gives you the option of opening an existing resource file or creating a new one. Because I'm giving the feature-length tour here, I assume you don't have a resource file to open. You should therefore click the New button:



A second dialog box appears asking you to name the new file you are creating. All Mac files have names, and a resource file is no exception. Type in something clever like **My Resource File** and then click the New button.

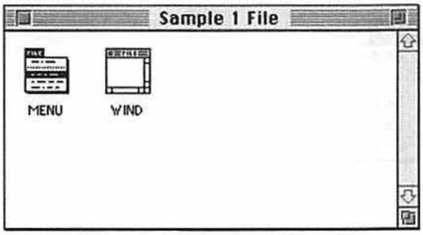
After clicking the New button, a brand new, empty window opens. Note that the window's title is the name I gave to the resource file:



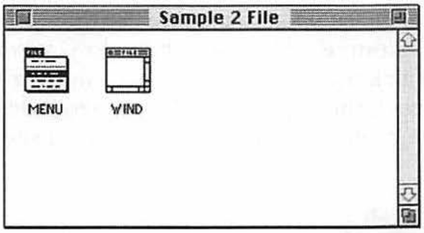
This window, which is called the *type picker,* eventually holds the names of the types of the different resources you create. Before proceeding, let me digress for a moment to explain the differences between the type picker window and the other two kinds of windows used by ResEdit. My digression allows me to stall on my explanation of how to actually add a resource to the file, thereby building the suspense to a nearly unbearable level.

Discerning the different ResEdit windows

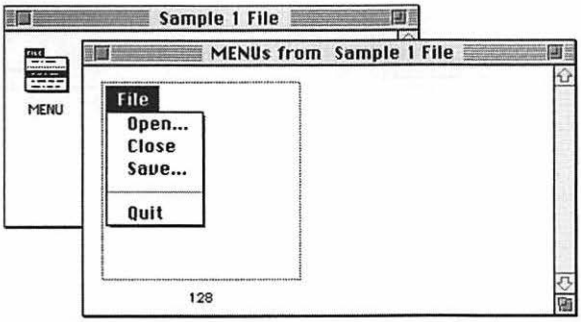
The type picker doesn't display each resource in a resource file. Instead, it displays each type of resource in the file. (You remember the different resource types such as 'WIND' and 'MENU'.) If a resource file has one 'WIND' resource and one 'MENU' resource, its type picker window looks like this:



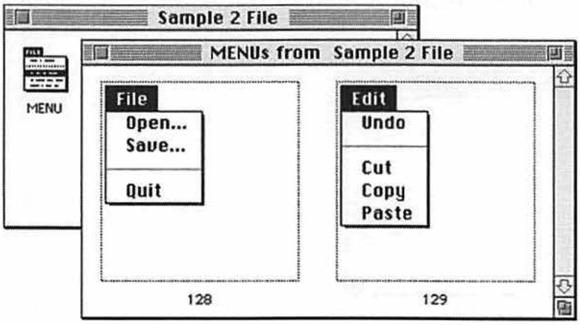
Now, look at the type picker for a resource file that has *two* 'MENU' resources and one 'WIND' resource:



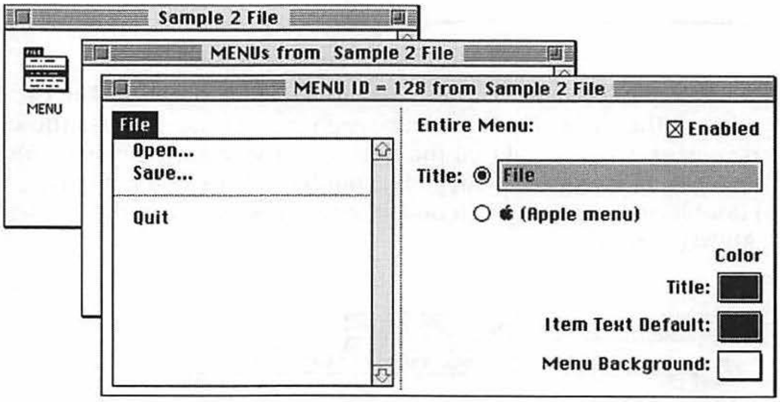
No, you didn't overlook anything, and it's not a misprint. The two type pickers look the same. That's because each of the examples has the same *types* of resources, which is all that the type picker shows. To list each *individual* resource of a single type in a file, double-click an icon in the type picker. Here I double-click the 'MENU' icon in the first resource file, the one with one 'MENU' resource:



When you double-click the 'MENU' icon, the window that opens is called a *resource picker*. The resource picker lists all of the resource files of a certain resource type. Here's the resource picker for the second resource file, the one with two 'MENU' resources:



Aha! Now you're getting somewhere. This resource file has two 'MENU' resources, and the 'MENU' resource picker shows two menus. You're not through yet, though. You know that the type picker displays the different resource types, and that the resource pickers display the different resources in each type. But ResEdit is a resource editor — so how do you *edit* one of these resources? Just double-click the name of a resource in the resource picker. For example, I double-click the dashed box that surrounds the File 'MENU' resource in the Sample 2 File. Here's the new window I see:



Entire Menu: ☒ Enabled

Title: ● File

○ (Apple menu)

Color

Title: ■

Item Text Default: ■

Menu Background: □

This window is an *editor,* which is the third and final window type ResEdit uses. An editor allows you to make changes to a single resource. There's a different editor for each type of resource. If you'd like to find out about using different editors for different resources, stay tuned — a little later in this chapter I cover the 'MENU' editor.

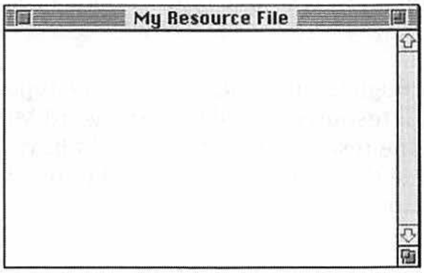
TIP

Beware of menu overload! You see *menu* written as 'MENU', MENU, and menu. How do they differ? A quick list may shed some light:

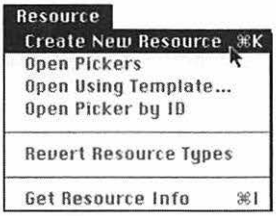
- **'MENU':** Refers to a menu resource. By convention, most Mac programmers and most Mac programming books put resource names between single quotes. That makes it obvious when a resource is being referred to.
- **MENU:** Also refers to a menu resource. This is the exact same resource as 'MENU'. ResEdit works with nothing *but* resources, so every four-character word you see in the resource editor refers to a resource. Because of this, Apple doesn't bother to surround each in quotes in its screen displays. That's simply the Apple and ResEdit way of doing things.
- **menu:** Refers to the actual menu in a program, not the resource.

Creating your very first resource

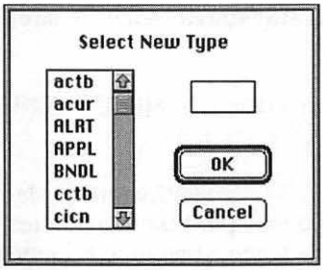
After creating a resource file and naming it something extraordinary, such as My Resource File, you have a window that looks like this:



This window is the type picker for the My Resource File resource file, but the type picker doesn't have any types in it. To add a resource, choose Create New Resource from the Resource menu:



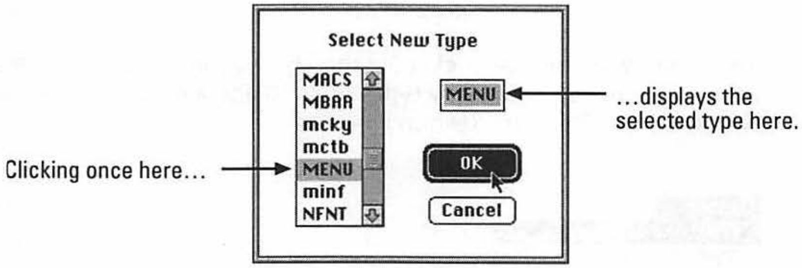
You see the Select New Type dialog box, shown here:



You use the scrollable list in the dialog box to choose which type of resource to create. Try scrolling through this list once. Pretty imposing, huh? This list has over 100 resource types. That's the bad news. Don't worry, though — the good news far outweighs the bad. A lot of these types create obscure resources that you never have to worry about. The 'itlk' type, for example, is defined by Apple as the "Remappings of certain key combinations before KeyTrans function is called for the corresponding 'KCHR' resource." Say again? I've programmed the Mac since it first came out over a decade ago, and I've yet to use that one. As a matter of fact, I've yet to hear of *anyone* using that one.

Some of the other resource types aren't as obscure, but I won't focus on them in this book. Even if I only cover a few resource types, I think I can manage to keep you well occupied.

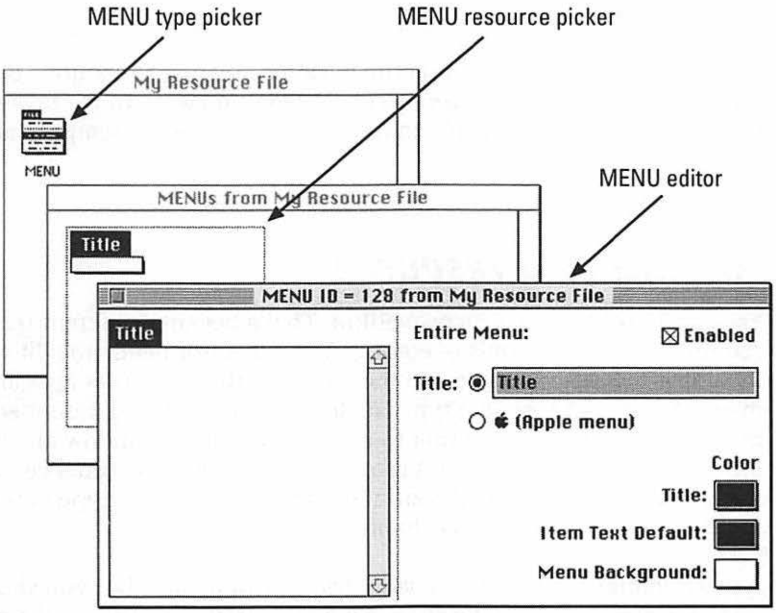
To create a resource, scroll through the list until you find the type you're interested in. To make a 'MENU' resource, scroll until the word MENU appears in the list. Next, click the resource type to highlight it. When you do, the resource type is displayed in the previously empty edit box. With the type selected, click the OK button:



Clicking once here...

...displays the selected type here.

After you click the OK button in the Select New Type dialog box, strange and wonderful things occur. A tiny picture of a menu appears in the type picker, a resource picker is created, and an editor opens. Here are the three windows that you see:



MENU ID = 128 from My Resource File

Title

Entire Menu: ☒ Enabled

Title: ● Title

○ (Apple menu)

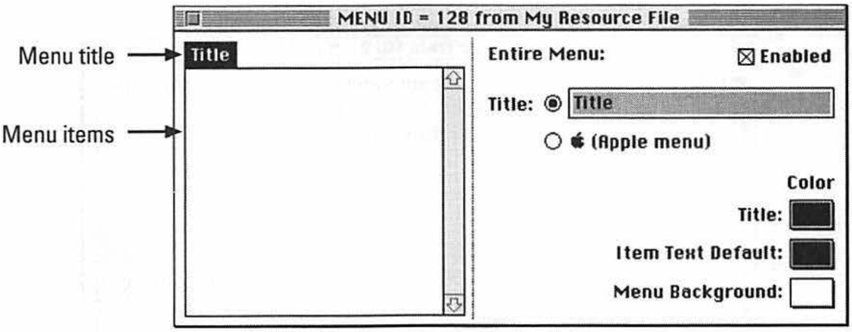
Color

Title: ■

Item Text Default: ■

Menu Background: □

Choosing Create New Resource from the Resource menu does just that — it creates a new resource. The 'MENU' resource you create this way could serve as a menu in a program's menu bar. The menu's title is, generically enough, Title. It has no items in it:



MENU ID = 128 from My Resource File

Title

Entire Menu: ☒ Enabled

Title: ● Title

○ (Apple menu)

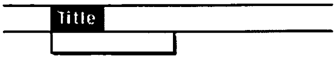
Color

Title: ■

Item Text Default: ■

Menu Background: □

Programs use resources. A program that makes use of my 'MENU' resource would look like this:



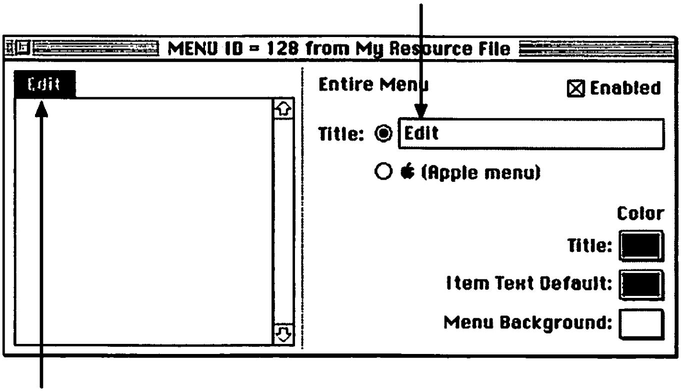
Stop and give yourself a pat on the back for creating a resource. But don't pat too long — you still have work to do. Besides, if I were to pat myself on the back too long, my wife would think I was choking and attempt to administer the Heimlich maneuver!

Adding to a resource

Each resource type has its own editor. That's because different resources require different methods of editing. A menu editor needs to edit words, including the menu's name and the names of the items that appear in the menu. A different resource type, such as a window, has different editing needs. For a window, you want to edit the size of the window and its general look, including whether or not it has a title bar along its top. I've discussed the 'MENU' resource, so I'll continue using that resource type in the remaining ResEdit examples in this chapter.

A programmer can perform several tricks with menus, but you should focus on the basics for now. A program's menu consists of a title — the name that appears in the menu bar that holds the menu — and a list of items, or commands, that appears when the menu is dropped down. To test ResEdit, you can create an Edit menu, a menu found in just about every Mac program. If you're following along, here's how to change the menu's title:

Typing a title here...



Entire Menu ☒ Enabled

Title: ● Edit

○ (Apple menu)

Color

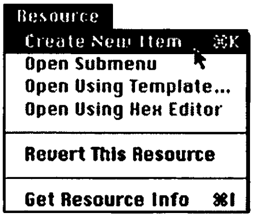
Title: ■

Item Text Default: ■

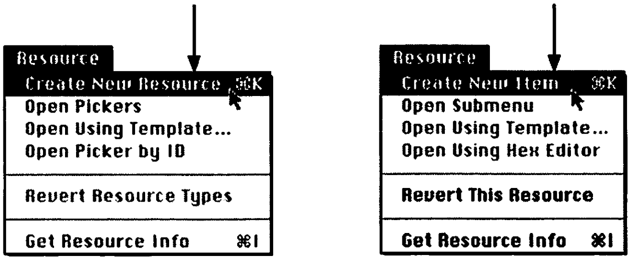
Menu Background: □

...causes it to also show up here.

Now add an item to the 'MENU' resource. To do that, choose Create New Item from the Resource menu:



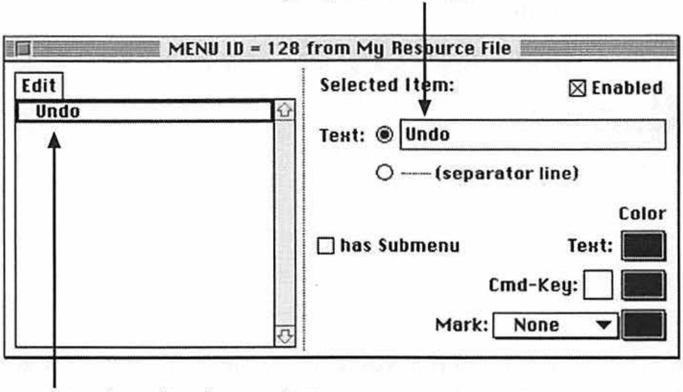
To create the 'MENU' resource, you choose the first item in the Resource menu, which is Create New Resource. To add an item to a 'MENU', you choose the first item in the Resource menu. This time, however, the item is Create New Item. ResEdit is no slouch of a program. It knows what you're doing, and it changes some menu commands to commands that are appropriate to the resource being edited. Here are the two faces of the Resource menu:


Resource menu before creating a resource

Resource menu after creating a resource

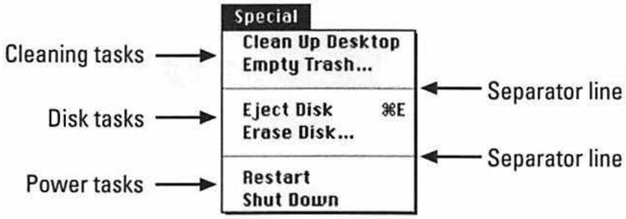
After choosing Create New Item, ResEdit inverts a section of the 'MENU' editor. Now type in the name of the first item you want to appear in the Edit menu. As you type, the characters appear in two places in the 'MENU' editor, the text box and under the menu's title:

Typing a menu item name here…



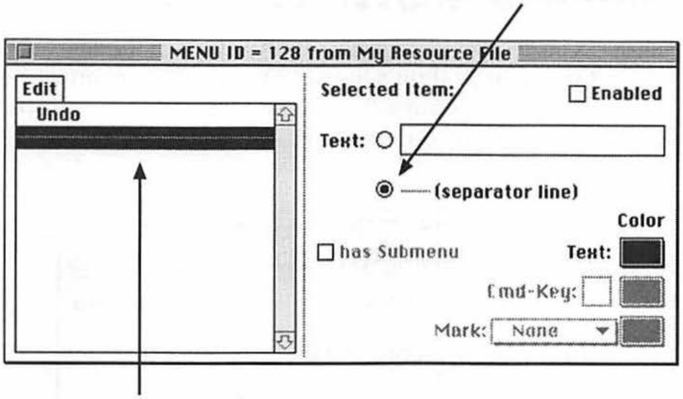
…causes it to also show up here.

That's it for adding a menu item. Well, that's *almost* it. That's it for adding a menu item that consists of text. There's also a menu item that isn't much of an item at all — the *separator line*. Separator lines allow you to group related menu items, like so:



To add a separator line to a 'MENU' resource, choose Create New Item from the Resource menu just as you did for the first menu item. Now, instead of typing in a name for the item, click the radio button labeled (separator line). The new menu item displays a dashed line:

Click this radio button…



MENU ID = 128 from My Resource File

Selected Item: □ Enabled

Text: ○

● —— (separator line)

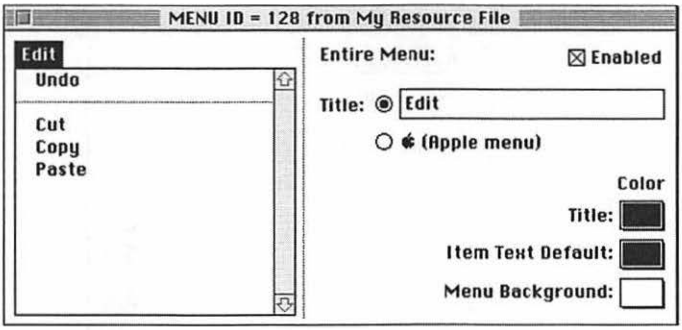
□ has Submenu

…to create a separator line.

To get a little practice, and to convince yourself that using ResEdit is a breeze, complete the Edit menu by adding three more items. Here's how to add Cut, Copy, and Paste commands to the menu:

1. **Choose Create New Item from the Resource menu.**
2. **Type** Cut.
3. **Choose Create New Item from the Resource menu.**
4. **Type** Copy.
5. **Choose Create New Item from the Resource menu.**
6. **Type** Paste.

Simple, isn't it? After those six steps the 'MENU' editor looks like this:



MENU ID = 128 from My Resource File

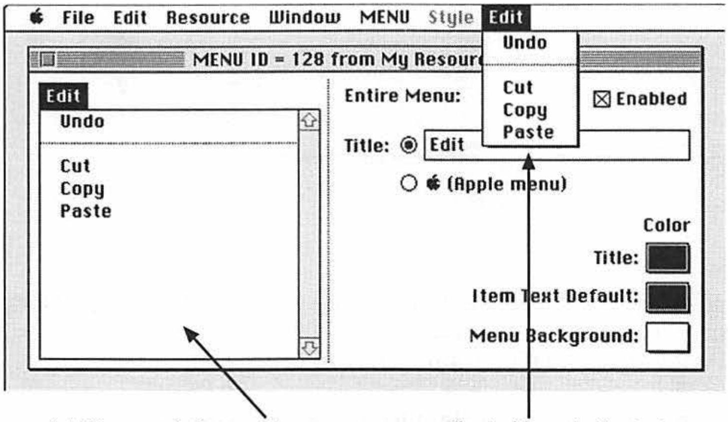
Entire Menu: ☒ Enabled

Title: ● Edit

○ (Apple menu)

Previewing a MENU resource

ResEdit allows you to preview a menu at any stage in its development. If you look at the last menu in ResEdit's menu bar, you see a menu with the same title as the one you're creating. Click the menu's title and it drops down to show the items you've added to your 'MENU' resource:



MENU ID = 128 from My Resour

Entire Menu: ☒ Enabled

Title: ● Edit

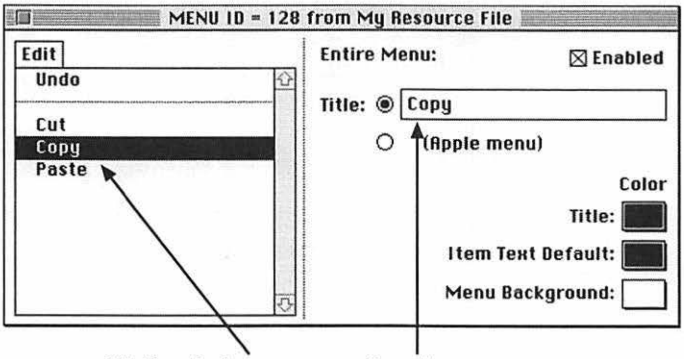
○ (Apple menu)

Additions and changes here... ...are reflected here in the test menu.

Editing an existing resource

When you edit an existing resource, you ResEdit work in one of the ResEdit editors. Since you're familiar with the 'MENU' editor, modifying an existing 'MENU' resource will be a task that's right up your alley.

To make a change to one of the items in a 'MENU' resource, click the item once to highlight it. Then type in the new item name:



MENU ID = 128 from My Resource File

Entire Menu: ☒ Enabled

Title: ● Copy

○ (Apple menu)

Clicking the item to edit highlights that item.

Type the new item name here.

The same procedure works for changing a menu's title. Click the title, such as Edit in the menu example, and then type in a new name.

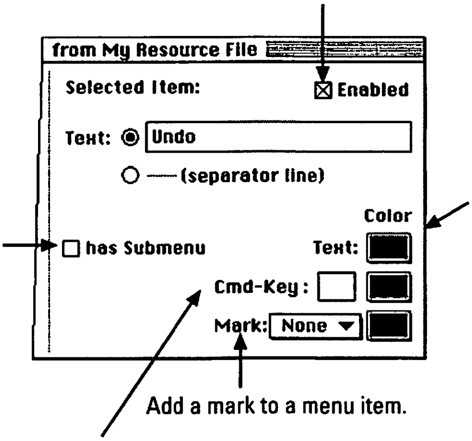
ResEdit also allows you to use its own Edit menu to make changes to a resource. To use the Edit menu, click once on the resource item, then make a selection, such as Cut Item or Copy Item, from ResEdit's Edit menu.

When you've completed your additions and changes to a resource file, save it. That's as easy as choosing Save from the File menu. The program doesn't ask you to name the file because you did that when you first started the ResEdit program.

Sorry, not now

I'm sure you noticed a few things in the 'MENU' editor window that I neglected to cover. The number of topics in computer programming is almost endless, and so I have to draw the line somewhere. Please forgive me if I stick to only the most basic concepts; otherwise I may never get to tell you all the really important things you need to know to get a simple Mac program up and running. But because you did inquire, though, I have to at least tell you what those other parts of the 'MENU' editor are for. As is my custom, I make use of a figure. Take a look at a portion of the 'MENU' editor window:

Enable or disable a menu.



from My Resource File

Selected Item: ☒ Enabled

Text: ● Undo

○ —— (separator line)

Color

□ has Submenu Text: ■

Cmd-Key: □ ■

Mark: None ▾ ■

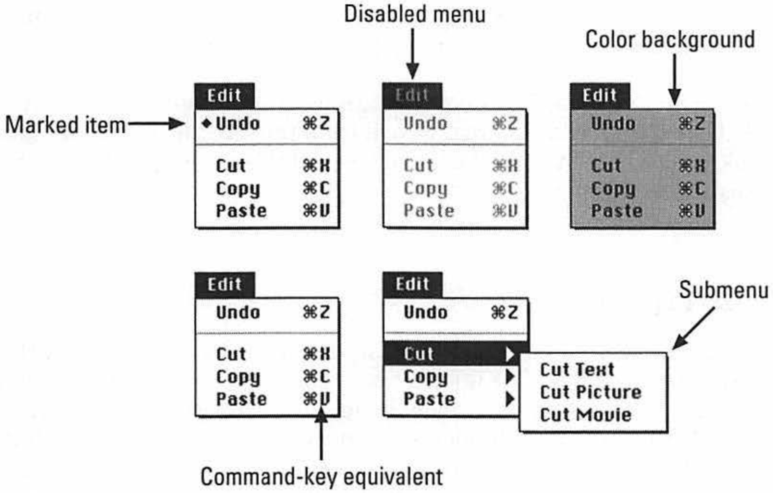
Add color to parts of a menu.

Add a submenu to a menu.

Add a mark to a menu item.

Give a menu item a Command-key equivalent.

To clarify this figure, I present yet another figure. Here's an example of each of those 'MENU' editor features:



Sure, you could accuse me of being a tease. I point out all the wonderful things that the 'MENU' editor can do, and then I tell you that I only cover a couple of them. Remember though, creating the resource is only half the battle. To implement a menu in a program, you need to write source code. And the more options you tack onto a menu, the more code you need to write.

So why did I bother to show you all of these neat things ResEdit can do? To show you all the neat things ResEdit can do! This book may be just the start of your programming endeavors. Should you survive its hundreds of pages (and I'm betting that you will), you may just want to go on to bigger and better things. Now you know that ResEdit is the primary tool for doing just that.

Chapter 8

Two Types of Resources: 'MBAR' and 'WIND'

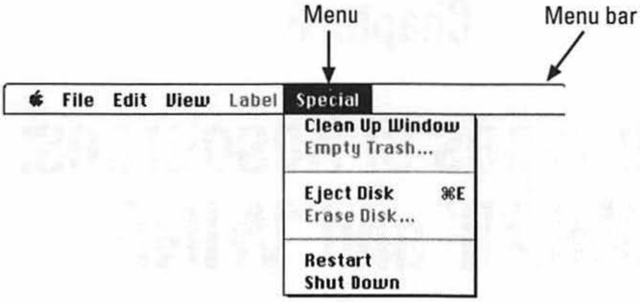
In This Chapter

- Assigning every resource an ID
- Spotting menus in their natural habitat, the menu bar
- Opening an existing resource file
- Working with the menu bar resource
- Creating and editing a window resource
- Previewing how source code uses a resource

Resources are a good news/bad news kind of thing. The bad news is that there are over 100 different types of resources. The good news is, who cares? You only need a few types to get a Mac program up and running! I discussed the 'MENU' type in Chapter 7 as an example of creating a resource. A 'MENU' resource represents a single menu. Now I address the other two types of resources used in this book — 'MBAR' and 'WIND'. These types of resources allow you to add a menu bar ('MBAR') and a window ('WIND') to your Mac programs.

Discovering the 'MBAR' Resource

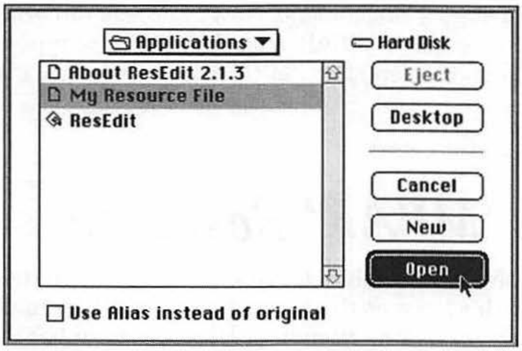
You sometimes hear people casually refer to that white horizontal bar along the top of the Macintosh screen as the menu. In fact, this is actually the *menu bar.* The menu bar holds one or more menus. Here's a menu bar with six menus in it (the counts as a menu):



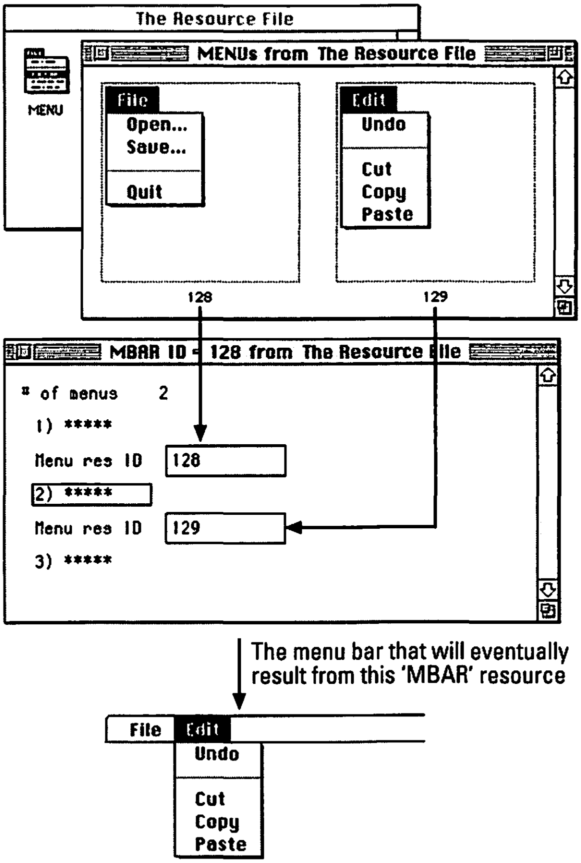
I don't mean to nitpick, but the distinction between the menu bar and the menus is important, and you're about to find out why. In Chapter 7, you see how to create and edit 'MENU' resources. To give your Mac program a complete menu bar, one other menu-related resource type needs to be created — the 'MBAR' resource. 'MENU' resources are individual entities that are not bound to one another in any way. Not, that is, until you specify which 'MENU's are to be a part of your menu bar. The 'MBAR' resource takes care of that.

Creating an 'MBAR' resource

When you open ResEdit, the first dialog box you see lets you create a new resource file or open an existing one. As in Chapter 7, you click New to create a file. I name mine My Resource File. After you create a resource file, you don't need to make a new one. To open the resource file, click once on its name and then click the Open button:



After opening an existing file, the type picker for the file is displayed. If I open My Resource File, its type picker looks like this:



# of menus 2
1) *****
Menu res ID 128
2) *****
Menu res ID 129
3) *****

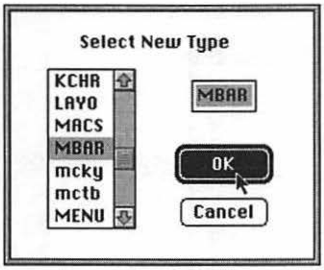
The menu bar that will eventually result from this 'MBAR' resource

Opening an existing resource file is a common practice. It's rare for a programmer to create a brand-new resource file, create resources in it, and then never edit that same file. As they work on a program, programmers usually think of new features to add, such as a new menu option for one of the menus.

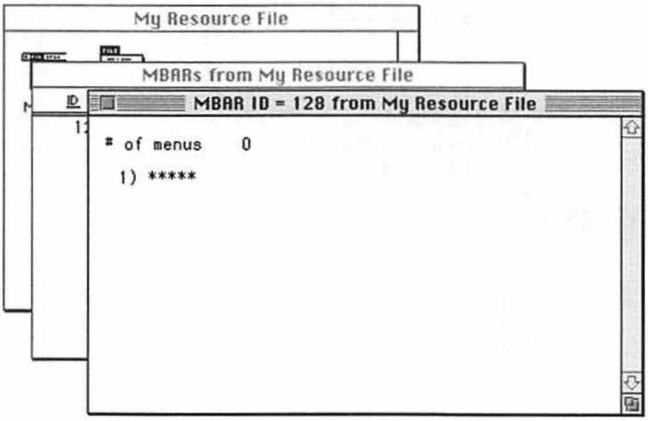
Adding an 'MBAR' to a resource file

Adding a new resource to a resource file involves the following steps (if the steps seem familiar, it's because they are the same steps you use to create a 'MENU' or any other resource, for that matter):

1. **With ResEdit running, choose Create New Resource from the Resource menu.**
2. **Select the resource type, which is 'MBAR', from the Select New Type dialog box that appears and then click the OK button.**



If you chose MENU from the Select New Type dialog box, a 'MENU' resource picker and a 'MENU' editor open. If you select MBAR, you again see a resource picker and an editor. This time an 'MBAR' resource picker and an 'MBAR' editor open:

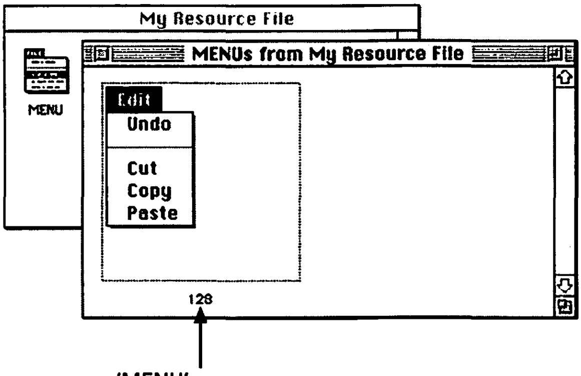

My Resource File
MBARs from My Resource File
ID
MBAR ID = 128 from My Resource File

```
# of menus    0
 1) *****
```

If you've read even just one book on computers, you've read a warning about saving your work — every computer book includes at least one. I'm not one to be left out of things, and so here's mine. *Save your work, and save it often.* In ResEdit, like most programs, to save your work, you choose Save from the File menu. Or just press the ⌘ key and the S key as a shortcut. Then if your Mac freezes up or the power cord gets pulled from the wall (accidentally, or intentionally by an irate spouse), you won't lose your work. It will still be there once you restart your computer.

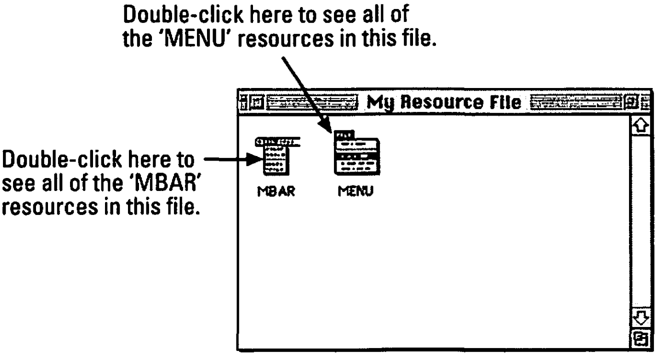
Adding a 'MENU' to an 'MBAR'

The purpose of the 'MBAR' resource is to list the individual 'MENU' resources that should be in a menu bar. If you want to include a particular 'MENU' in the list, what do you use to refer to it? Use its resource ID — that's a topic I covered in Chapter 7 in "Resource IDs". Before you start wildly paging back through the book, just look at the figure below. It shows the 'MENU' resource picker and the one 'MENU' it holds for My Resource File.

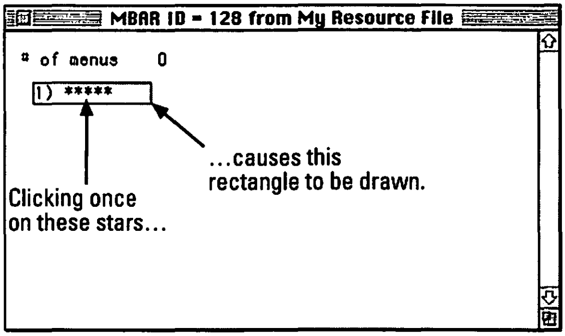


'MENU' resource
with ID of 128

At any time, you can view the resource picker for any resource type in a file. Simply double-click its icon in the file's type picker:


Double-click here to see all of the 'MENU' resources in this file.
My Resource File
Double-click here to see all of the 'MBAR' resources in this file.
MBAR
MENU

To add the one 'MENU' resource in My Resource File to this 'MBAR' resource, click once on the row of five stars in the 'MBAR' editor. After you click on the stars, you see a rectangle around the stars, like this:



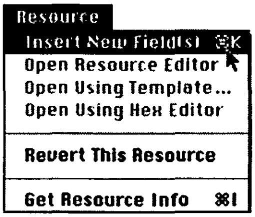
# of menus 0

1) *****

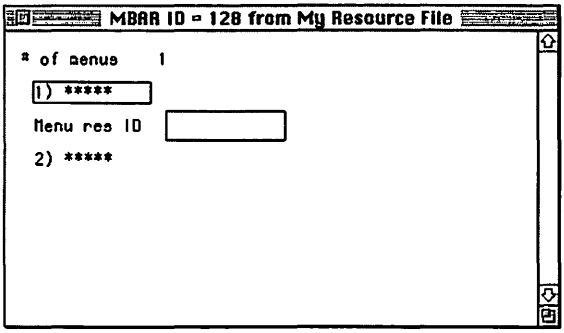
...causes this rectangle to be drawn.

Clicking once on these stars...

Next, choose Insert New Field(s) from the Resource menu. Here's that menu, with Insert New Field(s) selected:



Once again the Resource menu shows its chameleon-like nature by changing the name of the first item in it. For an 'MBAR', you're adding a new field — a new placeholder for the entry of a 'MENU' resource number. After selecting Insert New Field(s), here's what you see:



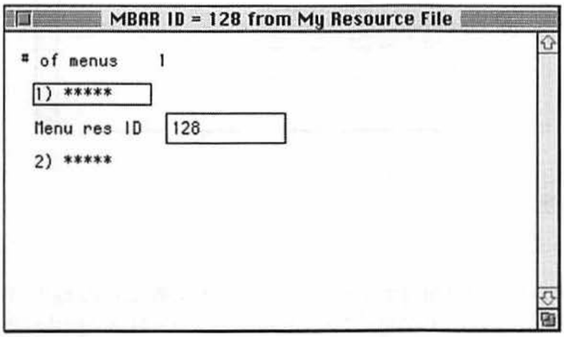
# of menus 1

1) *****

Menu res ID

2) *****

The new edit box is where you type in the resource ID number of a 'MENU' resource that you want in the 'MBAR' list. Click the mouse in the edit box and then type in the number 128. This resource file has just one 'MENU' resource in it, and that resource has an ID of 128. The 'MBAR' editor now looks like the one shown here:



MBAR ID = 128 from My Resource File

# of menus 1

1) *****

Menu res ID 128

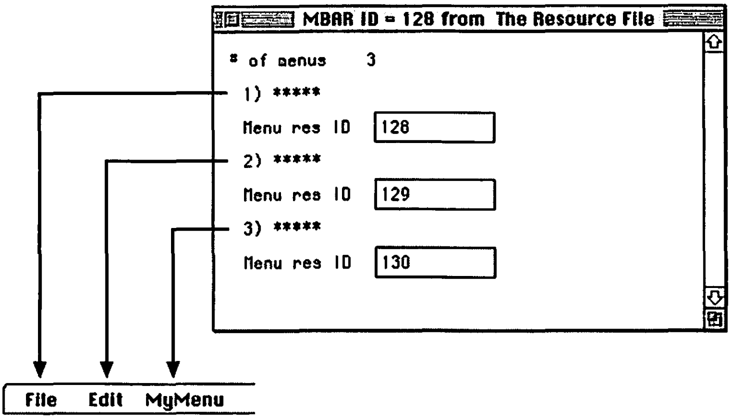
2) *****

If you have more 'MENU' resources in your resource file, you add each of them in the same manner by following these steps:

1. **Click on the row of five stars in the 'MBAR' editor.**
2. **Choose Insert New Field(s) from the Resource menu.**
3. **Click the mouse in the new edit box.**
4. **Type in the resource ID number of the 'MENU' resource you want to add.**

Menus come to order!

What determines the order in which each menu appears within a menu bar? In just about every Mac program, the File menu comes before the Edit menu (assuming, of course, you read from left to right). How does the programmer guarantee that the menus appear in this order? The 'MBAR' resource takes care of this awesome task. The first 'MENU' listed in the 'MBAR' is the first menu on the left in the program's menu bar. The second 'MENU' appears next, and so on. Here's a figure that shows the relationship between an 'MBAR' with three 'MENU' resources in it and the menu bar that results from it:

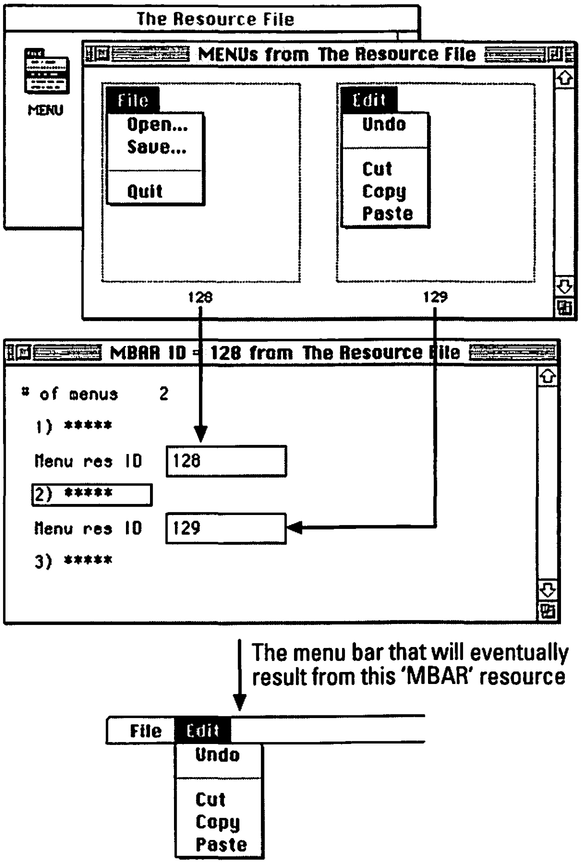


```
MBAR ID = 128 from The Resource File

# of menus    3
 1) *****
 Menu res ID   128
 2) *****
 Menu res ID   129
 3) *****
 Menu res ID   130
```

Hey, where's the Apple? In the preceding figure, no appears in the menu bar. Even though just about every Mac program uses the Apple menu, I don't show you how to include it in this chapter because implementing the Apple menu in a program requires a few techniques that are a little complicated. Don't worry, though. If you want this menu to appear in your programs, you'll find out how to add it in Chapter 21.

Summing up the 'MENU' and 'MBAR' connection

You may be happy to see that I end this long talk of menus with a killer of a figure that sums up the whole business of combining several 'MENU' resources into an 'MBAR'. Here's the interrelationship between a resource file with two 'MENU' resources listed in one 'MBAR' and the menu bar that would result:



```
# of menus     2
 1) *****
Menu res ID   128
2) *****
Menu res ID   129
3) *****
```

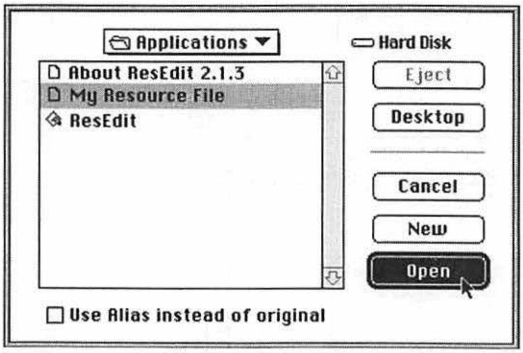
The menu bar that will eventually result from this 'MBAR' resource

Knowing that 'WIND' Is for Window

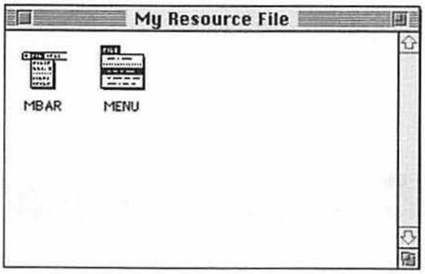
I played a really cheap trick back in Chapter 6 when I showed a window being drawn in a graphics program. I now want to take this opportunity to formally apologize for that. I'm sure you'll forgive me if you understand that I had to think of some way to illustrate creating a window without mentioning ResEdit (because you didn't know anything about ResEdit at that point). Now it's time to see how a window resource is *really* created the ResEdit way.

Opening a resource file . . . again

Begin by opening your resource file. Run ResEdit and then click once on the name of the resource file you want to open. Then click the Open button:



What do you see before you except the resource file's type picker. Mine now has two resource types listed — one for the 'MENU' resource created in Chapter 7 and one for the 'MBAR' resource that I just added a few pages back. If you're following along at home, your resource picker looks like this:



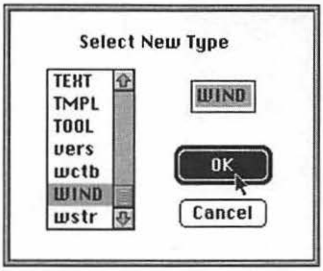
Breezing through a 'WIND' resource

You need to add a 'WIND' resource to My Resource File. If you've already read the material in Chapter 7 about creating a 'MENU' and the section in this chapter about creating an 'MBAR', then this process may sound familiar. If you've been skipping around, this is all new. Don't worry. Here's what to do:

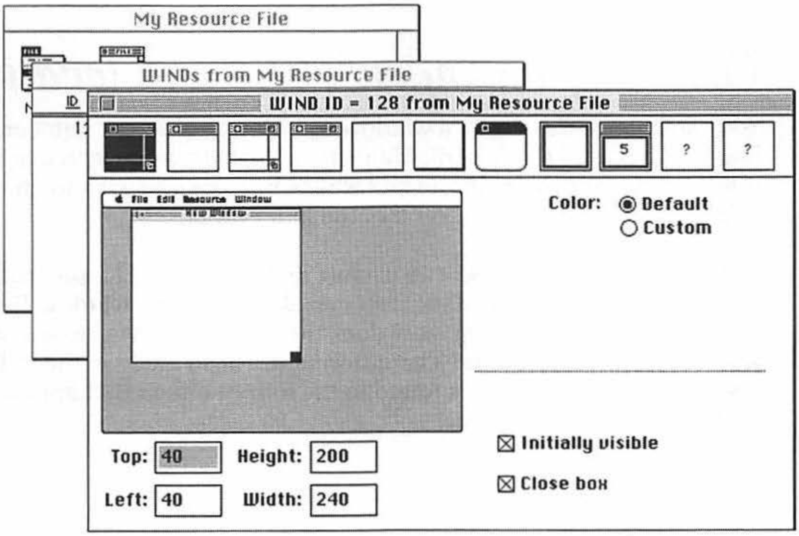
1. **With ResEdit open, choose Create New Resource from the Resource menu.**

   The Select New Type dialog box opens.

2. **Scroll down to WIND and then click it once. Click the OK button:**



A resource picker and a resource editor open. Here's a peek at the 'WIND' editor:


My Resource File
WINDs from My Resource File
ID
WIND ID = 128 from My Resource File
5
?
?
File Edit Resource Window
New Window

Color: ◉ Default ○ Custom

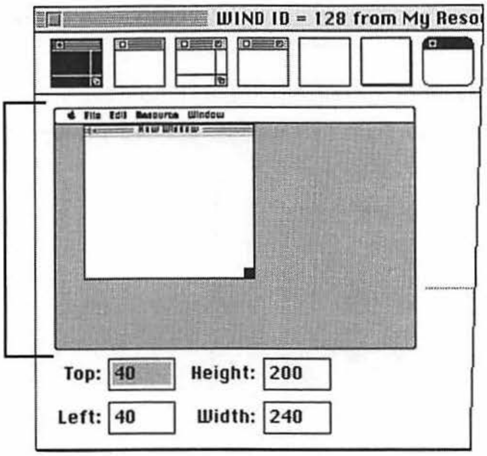
☒ Initially visible

☒ Close box

Top: 40 Height: 200

Left: 40 Width: 240

The 'WIND' editor provides you with an approximation of what your 'WIND' looks like as a window in a program. It shows a reduced view of a window on a Mac screen. Apple calls this feature of ResEdit the *MiniScreen:*



Top: 40 Height: 200

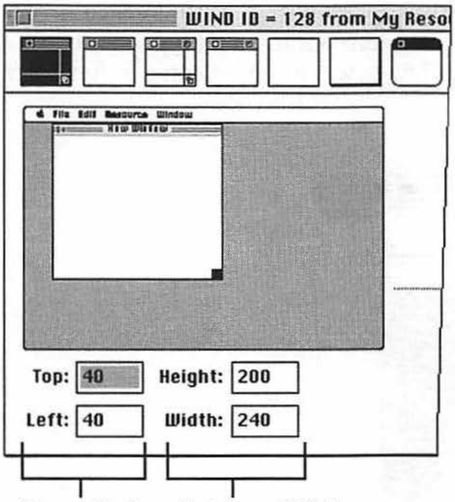
Left: 40 Width: 240

At this point, you could save your file and quit ResEdit — you've created a 'WIND' resource. But, hey, that would be too easy. You've come this far, so why not explore just a little bit? Besides, the longer you play around in ResEdit, the longer you can put off learning how to write source code!

Changing a window's size and location

You can change the size of a window by entering different numbers in the four edit boxes of the ResEdit MiniScreen. Click the mouse in a box, or press the Tab key until you're in the box whose number you wish to change. The Tab key highlights, in turn, each of the four edit boxes.

In a Mac program, the user can usually move a window to any location on the screen. In many programs, the user can also resize the window. But when it first appears on the screen, how does the window know where it should be and what size it should be? That information is included in the 'WIND' resource of the window. It's found in the four numbers that appear along the bottom of the 'WIND' editor:


WIND ID = 128 from My Reso

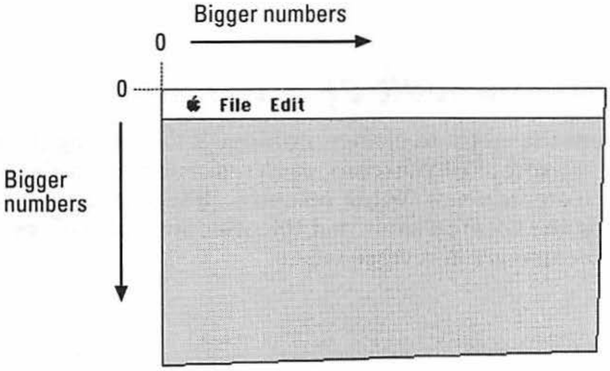
Top: 40 Height: 200

Left: 40 Width: 240

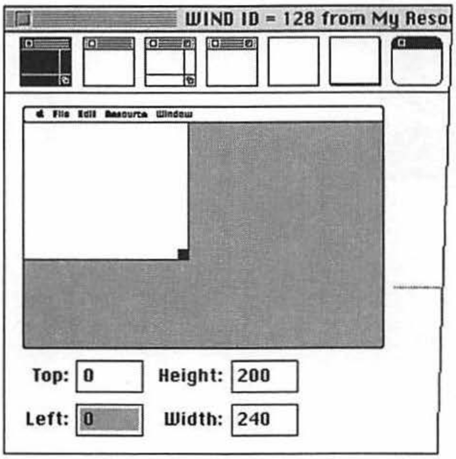
Top and Left control a window's placement.

Height and Width control a window's size.

It's simple enough that by setting the location of the window's top and left sides, and then adjusting the window's height and width, you can place a window anywhere on the screen. What may not be intuitive is just what the numbers refer to. They're part of the Mac's *coordinate system.* Every point on the screen is numbered so that the Macintosh can keep track of where things are located on its screen. Computers are very ordered creatures, so of course there is a strict protocol for numbering points on the screen. You get the details about this numbering system in Chapter 16. For now, you just need to know this much:



This figure shows you that the top left corner of the screen is considered the screen's vertical and horizontal zero point. So if I typed in 0 for the Top and 0 for the Left, here's how the change would be reflected in ResEdit's MiniScreen:



Top: 0 Height: 200

Left: 0 Width: 240

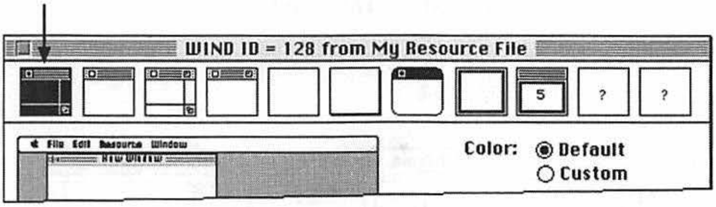
Try experimenting with the window's size and placement on the MiniScreen by typing different numbers in all four edit boxes. In Chapter 16, I describe all the particulars about these points that make up a Mac screen.

For you anxious types, I'll clue you in right here. Each point on the monitor is called a *pixel.* No, don't worry about getting glitter dust thrown in your eyes — these are *pixels,* not *pixies!* But pixels do have one thing in common with pixies — they're small. Very small. A line just one inch long is made up of over 70 pixels. But that's all I'm going to tell you. Now be patient until Chapter 16.

Changing the look of a window

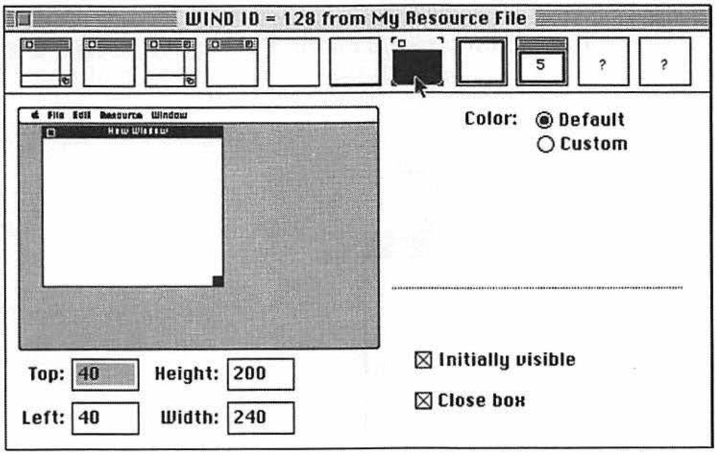
You can also use the editor to change a window's look. Along the top of the 'WIND' editor is a row of eleven icons, each representing a different window type. When you create a new 'WIND' resource, ResEdit assigns it the look of the first icon on the left. You know that this window type has been assigned because its icon appears in reverse video:

Highlighted icon when editor of a new 'WIND' opens


WIND ID = 128 from My Resource File
5
?
?
File Edit Resource Window
New Window

Color: ● Default ○ Custom

Try clicking a different icon in the row. In the figure below, I click the fifth icon from the right. Notice that the small window in the MiniScreen has changed its look to reflect the look of the selected window type.


WIND ID = 128 from My Resource File
5
?
?
File Edit Resource Window
New Window

Color: ● Default ○ Custom

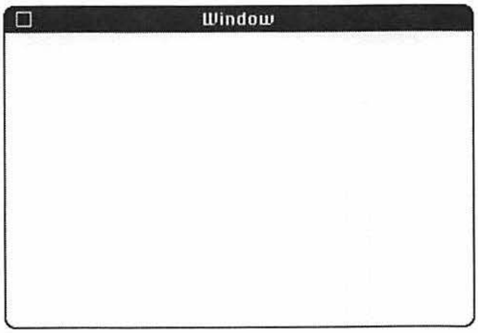
Top: 40 Height: 200

Left: 40 Width: 240

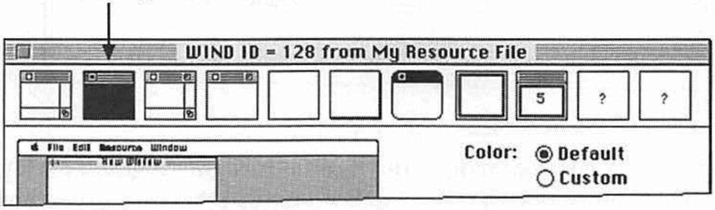
☒ Initially visible

☒ Close box

Here's how this window type would look once brought to the screen via source code:



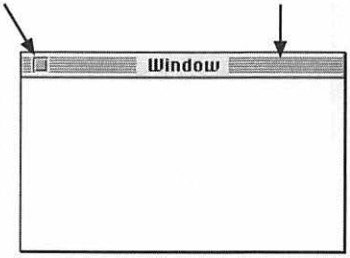
Now click the second icon from the left. That's the icon I use for all the 'WIND' resources that I create in this book.


The window type I use in this book

WIND ID = 128 from My Resource File
5
?
?
File Edit Resource Window
New Window

Color: ◉ Default ○ Custom

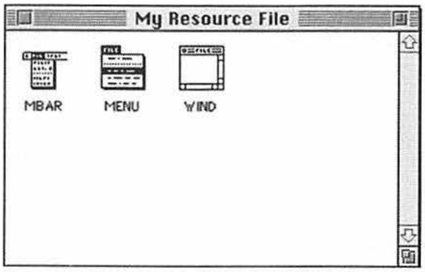
Why do I like this window type so much? Because it features a close box that allows the user to close the window. The window also has a title bar that lets the user move the window around on the screen. Here's what this window type looks like once brought to the screen with the help of some source code:


Close box

Title bar

Moving on

I've covered the major topics concerning the 'WIND' resource, so it's time to save my file and quit ResEdit. Before I do, take a look at the type picker for My Resource File:



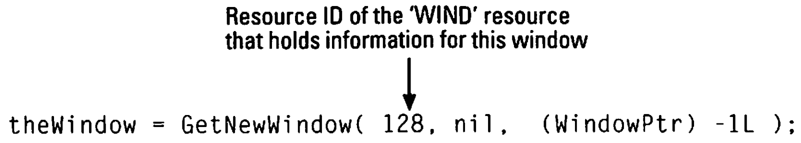
Of the 100-plus resource types available to you, these are the only three you need to get a real live Macintosh program up and running.

Proving that Resources Are Valuable to Source Code

Every resource has an ID number that helps to identify it. Identify it to what, though? In some cases, the ID helps one resource identify other resources, like when I listed 'MENU' resource ID numbers in an 'MBAR' resource. That let the menu bar 'MBAR' resource know which 'MENU' resources would be in its list of menus. In other instances, a resource ID helps source code find the resources it wants. Here's a line of source code from Chapter 5's ExampleOne program to show you what I mean:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

This line of code displays a window on the screen. But there are almost a dozen different types of windows, and a window can take on just about any size or screen location. How does GetNewWindow obtain this kind of detailed information about a window? Yes, yes, I hear you. From a 'WIND' resource, of course. What if the resource file contains more than one 'WIND' resource — which one does GetNewWindow use? GetNewWindow uses the resource whose number appears in the first parameter (parameters give information to a function) between the parentheses:



Resource ID of the 'WIND' resource that holds information for this window

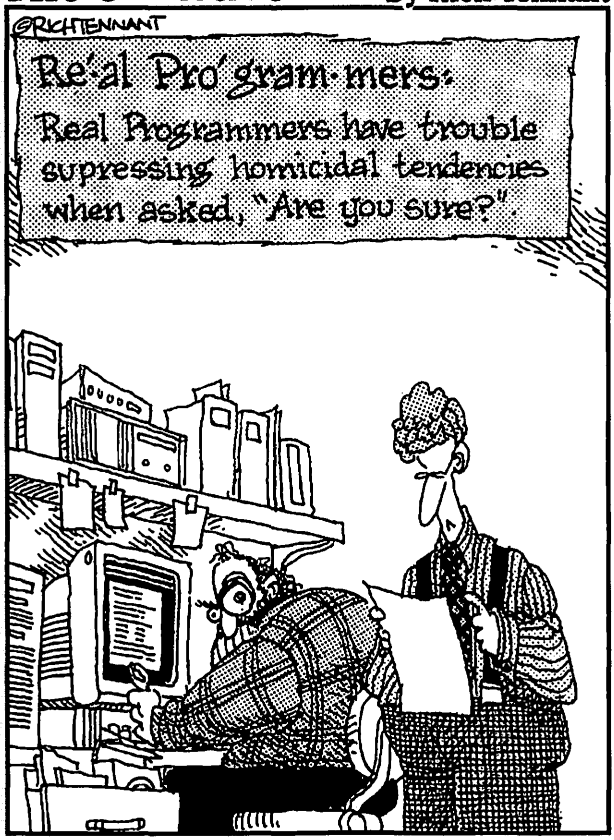
```
theWindow = GetNewWindow( 128, nil,  (WindowPtr) -1L );
```

Hopefully this figure proves that all your efforts creating resources with ResEdit are not without purpose. Resources really are important, and they are really useful when you write your own source code.

Part III

Using a Compiler

The 5th Wave By Rich Tennant


©RICHTENNANT
Re'al Pro'gram·mers:
Real Programmers have trouble supressing homicidal tendencies when asked, "Are you sure?".

In this part . . .

Your Mac isn't nearly as smart as it appears to be. You see, it doesn't understand a single word you type. When you create a source code file by typing in C language commands, the Mac needs the help of a software program to translate your source code into numbers the computer can understand. That very helpful program is called a *compiler.* This book uses the CodeWarrior compiler by Metrowerks for its examples. As you've probably already noticed, this book's CD-ROM even supplies you with a trimmed-down version of this same compiler.

To save you the trouble of wading through the hundreds and hundreds of pages that make up the CodeWarrior user's guide, I condense it all down to just a few chapters. Now, the fact that I ignored most of what's in my CodeWarrior manual may seem like a slap in the face to the technical writers over at Metrowerks. Hold on a minute, though. I'm not saying that their work is in vain. The entire Metrowerks CodeWarrior user's guide is very important to more advanced Mac programmers. For the rest of us, the very basics will do just fine. You'll find the basics of compiling with the CodeWarrior compiler in this part.

Chapter 9

Getting to Know You: The CodeWarrior Compiler

In This Chapter

- Understanding the differences between CodeWarrior Professional and CodeWarrior Lite
- Choosing CodeWarrior over other compilers
- Organizing files in a CodeWarrior project
- Creating and naming a project
- Adding libraries to a project
- Adding a file to and removing a file from a project

The topic of compilers often intimidates new programmers. But compilers really aren't such frightening beasts. A compiler does nothing more than turn your human-readable source code into a computer-readable program. In this chapter, and in the remainder of Part III, you discover the specifics of using the CodeWarrior compiler.

Comparing CodeWarrior Professional and CodeWarrior Lite

CodeWarrior Professional is the Macintosh compiler developed by a company named Metrowerks. CodeWarrior Lite is that same compiler, but with a few of its features disabled. You obtain CodeWarrior Professional by purchasing it from Metrowerks; you obtain CodeWarrior Lite by purchasing this book.

CodeWarrior Professional exists to provide the Macintosh programming community with a very powerful, easy-to-use means of developing Macintosh applications. CodeWarrior Lite exists to give a person like yourself the opportunity to see if programming is really your cup of tea before spending your hard-earned money on the full-featured version of CodeWarrior.

I said CodeWarrior Lite has some features disabled. So, is this version of the compiler of any use to you? Look at the following list of things you can do with the CodeWarrior Lite program that comes with this book, and then you can be the judge. CodeWarrior Lite allows you to:

- Open all of the example files on the CD-ROM that comes with this book.
- Examine all of the source code in all of the files you open.
- Compile all of the source code in all of the files you open.
- Create a standalone, double-clickable Mac application for each example.
- Modify the source code in existing examples.
- Compile and test the changes you've made to any of the examples.

Not bad. What can't you do with CodeWarrior Lite? You can't start from scratch and create a new project and build your own complete Macintosh application. To do that, you need to do two things:

1. **Buy the full-featured version of CodeWarrior.**
2. **Read this book!**

A few of the tasks discussed in this book apply only to readers who own CodeWarrior Professional, or Discover Programming (the Metrowerks version of CodeWarrior that provides programmers with an intermediate step between CodeWarrior Lite and CodeWarrior Professional). Those discussions are clearly marked as such, so don't worry about wasting your time if you don't own CodeWarrior Professional. Even if you're working with the Lite version of CodeWarrior, though, you may still want to read along to see how things are done in the real world of Mac programming. If you ever do decide to move from CodeWarrior Lite to CodeWarrior Professional, you may appreciate having the information under your belt.

Choosing CodeWarrior

Different software companies make compilers — so why did I select the CodeWarrior compiler by Metrowerks as the one to write about? For several reasons — as you're about to see.

CodeWarrior has everything you need

CodeWarrior is more than just a compiler; it's an *integrated development environment,* or IDE. That fancy phrase simply means that CodeWarrior includes several programming tools all rolled into what appears to be a single application. That means CodeWarrior gives you easy access to a text editor that creates source code, a compiler that compiles the code, and a debugger that hunts down mistakes in the code.

So you could say that CodeWarrior is loaded with features that make Mac programmers drool. I say, let them drool. As someone just starting out, you're looking for simplicity, and CodeWarrior offers that, too. While CodeWarrior is actually much more than just a compiler, it's still easy to use. It has a *clean interface,* as they say. As a beginner, you may only be interested in a handful of the many CodeWarrior menu options. And being the user friendly IDE that it is, CodeWarrior allows you to develop your own programs using only this handful of menu items. While the name CodeWarrior can refer to the program's entire IDE, the word is often used more loosely. Most programmers are just referring to the program's compiler when they talk about *CodeWarrior,* and I do the same.

Apple isn't the best (for once)

Apple makes a compiler, too. And while Apple products are quality products, their Macintosh Programming Workshop (MPW) isn't your best choice as far as compilers are concerned. That's because MPW was created for seasoned programming professionals. So while I normally wholeheartedly recommend Apple products, I refrain from that practice in this one case.

You can join the CodeWarrior support club

Sometimes it's good to be different — sometimes it's not. I didn't choose CodeWarrior just to part of the *in* crowd. I chose it because of support. If you ever have a question about the compiler, you can turn to just about any Macintosh programmer for help. Chances are just about any programmer that you talk to uses, or is familiar with, CodeWarrior.

Yes, some evil programmers love to see newcomers sweat. But the majority of Mac enthusiasts enjoy extending a hand to someone struggling with a programming problem — they've been down that road, too. As a matter of fact, if you have Internet access (either from an online service such as America Online or CompuServe, or through an Internet service provider), you can post your Mac programming questions to more experienced programmers. You usually get a response that very same day. If you can access Internet newsgroups, add `comp.sys.mac.programmer.codewarrior` and `comp.sys.mac.help` to the list of newsgroups you subscribe to.

Source code is held in a text file, and all versions of CodeWarrior have a built-in text editor that allows you to write source code and save it to a text file. The advantage of this approach is that you don't have to run both a text editing program (like SimpleText) *and* a compiler. That means you only need one application running rather than two. And, because the text editor is an integral part of CodeWarrior, the compiler can work with the editor. An example of the two working together is when you make an error in typing your source code. When that happens, the compiler automatically marks the error in the text editor. So that you know a text file was created by CodeWarrior, the compiler gives the file its own special icon:

Now turn your attention to the program icon that replaced the generic compiler icon. It looks like this:

The CodeWarrior program serves as your command center for writing and compiling source code, and for making and testing Mac programs. In short, you'll be using this one program to manage the files that will be turned into a program.

Creating a Project

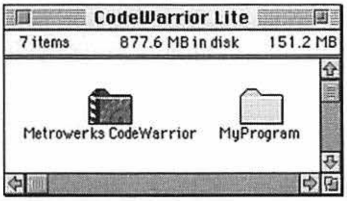
Whether you want to write a program that simply opens a window and writes *Hello, World!* or one that does inventory and accounting for a multimillion-dollar business, you start each program in exactly the same way: by creating a CodeWarrior project. What's a project? A very important part of the CodeWarrior way of doing things.

The source code of a program is held in a text file. Sometimes, when there's a lot of source code, a programmer divides it up into several text files. When it comes time to compile the source code, CodeWarrior is smart enough to know how to combine the source code from different files into one program. It does need a little help, though. The compiler needs to be told *which* files to use. A *project* groups multiple files together so that CodeWarrior knows which ones to use when it combines files into a program.

Every Macintosh program you create using CodeWarrior starts as a project. You create a new project, give it a name, and then add files to it to build and shape your program. Most of the files — the libraries — already exist. (Several libraries are included on the CD-ROM that holds CodeWarrior.) The other files — the text file holding your source code and the resource file holding your resources — have to be created by you.

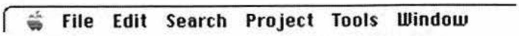
Creating a project folder

Of course, before you can explore creating a project, you have to install CodeWarrior Lite on your hard drive. If you haven't done so already, read Appendix F to see how that task is accomplished. The CodeWarrior environment — that is, the folders and files that make up the CodeWarrior package — reside in a single main folder. If you installed CodeWarrior Lite from the CD-ROM that comes with this book, that folder is named CodeWarrior Lite. If you purchased CodeWarrior Professional, this folder has a different name, most likely something like CodeWarrior Pro 4, or something similar. Regardless of the name, it was the installation process that created this folder and its contents for you. If you're a neat, tidy person, you want to create a new folder to hold your new project. If you're not neat and tidy, create one anyway! Double-click this main CodeWarrior folder to open it and then choose New Folder from the File menu. Give the folder an appropriate name, such as MyProgram, as I do here:

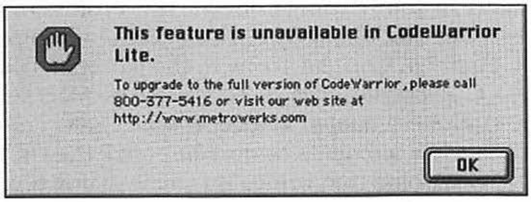


Creating a new project

In the main CodeWarrior folder, alongside your newly created MyProgram folder, is a folder titled Metrowerks CodeWarrior. This folder holds the CodeWarrior application. Double-click the Metrowerks CodeWarrior folder to open it, and then double-click the CodeWarrior icon to start up the program. Your Mac's desktop looks the same, but its menu bar has changed. You see a menu bar that looks like this:



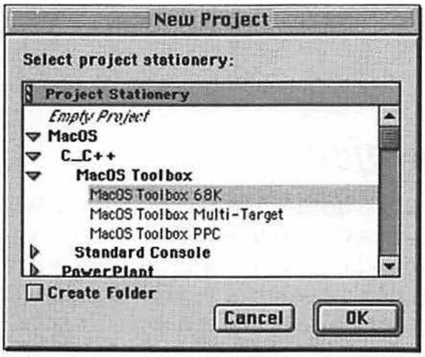
To create a new project, choose New Project from the File menu. If you're using the version of CodeWarrior that comes from the CD-ROM included with this book, you see a dialog box like this one:


This feature is unavailable in CodeWarrior Lite.

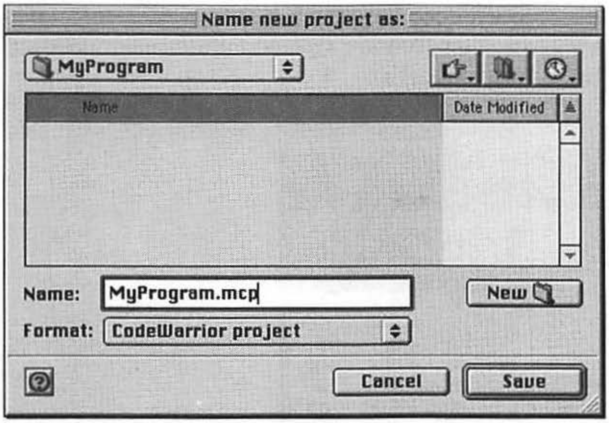
To upgrade to the full version of CodeWarrior, please call 800-377-5416 or visit our web site at http://www.metrowerks.com

Dirty trick? Not at all. The limited version of CodeWarrior that is included with this book allows you to open all of the example projects that I created for this book. It *doesn't* allow you to create new projects. If it did, you'd have little reason to buy the full-featured version of CodeWarrior. Since CodeWarrior is about the only thing that Metrowerks sells, they wouldn't stay in business too long if they gave their product away for free, right?

If you happen to already own CodeWarrior Professional, choosing New Project from the File menu results in the display of the dialog box shown below. If you're using this book's Lite version of CodeWarrior, just follow along with me so that you know what to expect should you decide to upgrade to the real thing:



Before clicking the OK button in this dialog box, do two things. First, click on the Create Folder check box to uncheck it. You already created a folder to hold what will be the new project. Now, choose a *project stationery* from the dialog box list. CodeWarrior allows you to create different types of projects. You can create a project that is used to hold files that are written in C language source code, or instead create a project that will hold files written in a different language, such as Java. There are other types of projects too. In this book I'll always use the same type of project stationery — it's the one highlighted in the above figure. If you're using CodeWarrior Professional and are at this dialog box, select this stationery by clicking on the appropriate little arrow icons in the list. First, click on the arrow to the left of MacOS. Then click on the arrow by C_C++. Finally, click on the arrow by MacOS Toolbox. Now click once on the stationery named MacOS Toolbox 68K. That's a lot of clicking, but it only takes a few moments to do. *Now,* click the OK button to finally select the stationery. Doing that brings up a new dialog box:

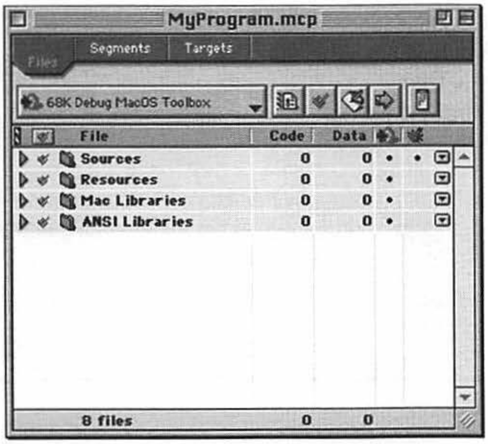


Name: MyProgram.mcp

Format: CodeWarrior project

In this dialog box, navigate your way into the project folder you created — the one I named MyProgram. Use the pop-up menu located at the top of the new project dialog box to backtrack out of the Metrowerks CodeWarrior folder and into the main CodeWarrior folder. Then double-click the MyProgram folder in the scrolling list beneath the pop-up menu. You should be in the MyProgram folder, as shown above.

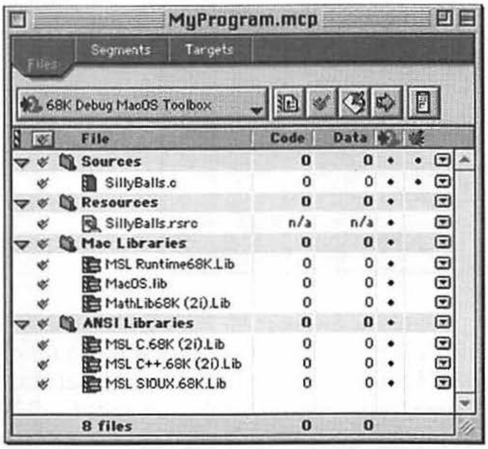
Note that the MyProgram folder is empty, and it should be. There won't be anything in it until I click the Save button to create a new project. First, though, I type in a name for the project. Of course, I remember to use the proper project-naming convention (see the "Naming a project" sidebar in this chapter). Assuming my program's name is MyProgram, I name the project MyProgram.mcp. Again, refer to the previous figure. After clicking the Save button, the dialog box disappears and a CodeWarrior project window opens:



68K Debug MacOS Toolbox

File	Code	Data
Sources	0	0
Resources	0	0
Mac Libraries	0	0
ANSI Libraries	0	0
8 files	0	0

Click on the little arrow icons along the left side of the project window to reveal the names of the files that are grouped under each heading (such as Sources, Resources, and so forth):



File	Code	Data			
Sources	0	0	•	•	
SillyBalls.c	0	0	•	•	
Resources	0	0	•		
SillyBalls.rsrc	n/a	n/a	•		
Mac Libraries	0	0	•		
MSL Runtime68K.Lib	0	0	•		
MacOS.lib	0	0	•		
MathLib68K (2i).Lib	0	0	•		
ANSI Libraries	0	0	•		
MSL C.68K (2i).Lib	0	0	•		
MSL C++.68K (2i).Lib	0	0	•		
MSL SIOUX.68K.Lib	0	0	•		
8 files	0	0			

Naming a project

When you start a new project, CodeWarrior asks you to give it a name. Just about any name should work. CodeWarrior would recognize all of the following examples: My Program, MyProgram, SuperGame, Dan's C Program.

A project holds the files used to create a single program. The project usually has the same name as the program will eventually have. Thus, if you're developing a game that will be named SuperGame, you should name the project SuperGame. That's not a steadfast rule, but it is the way things are typically done. A project is a Macintosh file, so its name can consist of up to 32 characters — just like any other Mac file.

While all of the preceding names would work, it's important that you add a little something to the end of your project's name — no matter what the name. Pick a name and then type a period followed by the letters *mcp*. These letters stand for Metrowerks Codewarrior Professional, by the way. If I added this ending to each of the above examples, the project names would now look like this: My Program.mcp, MyProgram.mcp, SuperGame .mcp, Dan's C Program.mcp

Using the .mcp ending in a project name has become a CodeWarrior *convention*. That is to say, anyone who uses CodeWarrior immediately recognizes a file name ending in this way as a CodeWarrior project.

Libraries: The hidden additive in every project

All of the examples included in this book are small enough to fit easily in one text file. But all projects contain more than this one file. One of the project files is the source code text file that you create. Another file that appears in each project is a resource file. The other files found in each project are called *libraries.* A library contains source code that is already compiled and is ready to be combined with your own source code. Who wrote the code in the library? In this case, the folks at Metrowerks and Apple are responsible.

Exactly who wrote the code and how they turned it into a library aren't important details. What is important is that the code works in conjunction with your own code — with almost no effort on your part.

ON THE CD

If you're using CodeWarrior Lite, you can't create the new project pictured above. But if you'd like to see such a file, look inside the *...For Dummies* Examples folder that you copied from this book's CD-ROM to your hard drive. In that folder you find a folder titled C09 MyProgram. Look in there for the MyProgram.mcp project. To open this project, just double-click its icon (see Chapter 10 for more information on opening projects). While you can't use CodeWarrior Lite to create this project from scratch, you can view the one I created with my full-featured version of CodeWarrior.

In the previous figure, you can see that CodeWarrior is kind enough to add several files to a new project. Most of these files are libraries that hold code that gets used in conjunction with the source code you write yourself. Fortunately, you never have to worry about the code that's in these files — so don't worry about them (and don't worry if your project has library files with names that differ from the ones shown in this book's figures). If you just want to have something to worry about, though, see the sidebar "Libraries: The hidden additive in every project" in this chapter.

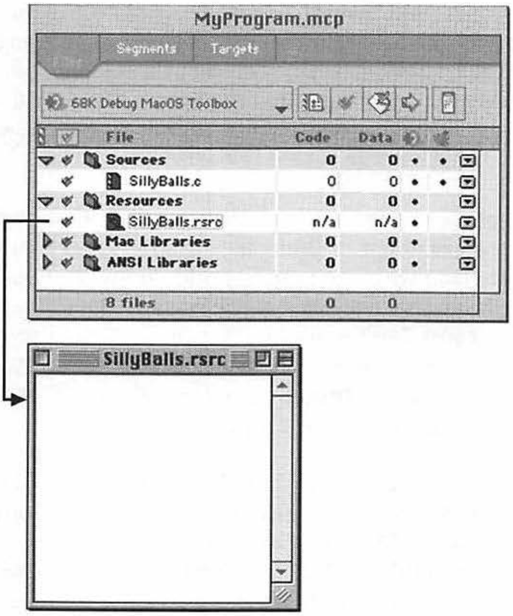
What about those files that have the words *SillyBalls* in their names? It turns out that these files aren't of much interest to you. So why did CodeWarrior place them in the new project? Mostly to serve as placeholders of a sort. They're in the project to remind you that you need to add your own source code file (in place of the SillyBalls.c file) and resource file (in place of the SillyBalls.rsrc file). You'll see how that's done a little later in this chapter. But first, it's time for a ResEdit flashback.

Working Together

One of the two files you need to add to any new project is the resource file. Of course, you have to create such a file before you can add it to a project. (Part II of this book provides information on creating a resource file.) In the next few pages, I show you how CodeWarrior works with ResEdit to create a resource file. I also recap how to create a resource file by providing the steps you'd take to create a resource file that can be used with the MyProgram.mcp project.

Launching ResEdit

In Macintosh terminology, *launching an application* means double-clicking the program's icon to start up the program. Because you edit resources just about any time you create a new project, CodeWarrior provides a neat way of launching ResEdit to make it easier to do. Rather than have CodeWarrior search through folders for the ResEdit program, you can simply double-click the file named SillyBalls.rsrc in the project window. When you do that, ResEdit starts right up. If you already have ResEdit running at the time you double-click the SillyBalls.rsrc file, that's okay. In either case, the ResEdit program opens and an empty resource file named SillyBalls.rsrc appears. Kind of like this:



File	Code	Data		
Sources	0	0	•	•
SillyBalls.c	0	0	•	•
Resources	0	0	•	
SillyBalls.rsrc	n/a	n/a	•	
Mac Libraries	0	0	•	
ANSI Libraries	0	0	•	
8 files	0	0		

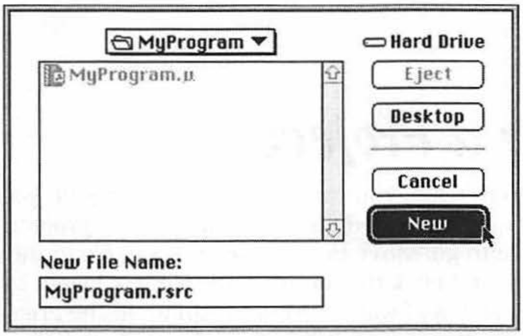
TIP

This method of firing up ResEdit works in both the Lite version and the full-featured version of CodeWarrior. If you haven't already done so, open the MyProgram.mcp project found in the C09 MyProgram folder and give it a try!

Creating the new resource file

The name SillyBalls.rsrc isn't the ideal name for a resource file because it doesn't provide any hint as to what the file is to be used for. ResEdit doesn't allow you to rename the SillyBalls.rsrc file, but it does allow you to easily create a new file — just choose New from the File menu.

You want to keep the new resource file right alongside the project that will use it. To move to the folder that holds your new project, use the pop-up menu at the top of the dialog box that appears after choosing the New menu item. Now type in a name for the new resource file. You want your project's resource file to have a name similar to the name of the program it will become a part of (resources and code are merged to form a program). The simple thing to do is give the resource file the name of the program, with a period and the letters *rsrc* following. For the program that is to be named MyProgram, a resource file named MyProgram.rsrc is quite appropriate. After entering the name, click the New button:



MyProgram.rsrc

Now you have a second empty resource file on your screen. This new one is named MyProgram.rsrc:

Adding a 'WIND' resource

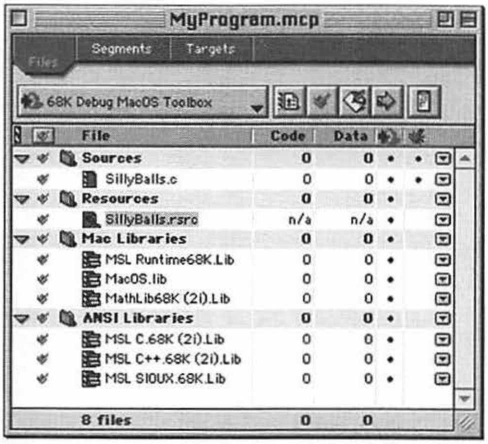
The program that you create using the MyProgram.mcp project is very simple; the only resource this program needs is a single 'WIND' resource. To add a resource to the open MyProgram.rsrc file, follow these steps:

1. **Choose Create New Resource from the Resource menu.**
2. **Use the scroll bar in the dialog box that appears to scroll down to the WIND item and click it.**
3. **Click the OK button.**

That's all you need to do to create the MyProgram.rsrc resource file. Choose Save from the File menu to save your work. You can quit ResEdit if you want, or, if you think you may be tinkering around with resources in the near future, you can just close the MyProgram.rsrc file and leave ResEdit running. Return to CodeWarrior Professional or CodeWarrior Lite by either clicking the MyProgram.mcp project window that should be visible on your desktop, or by selecting the compiler's name from the menu that appears at the far right of the menu bar.

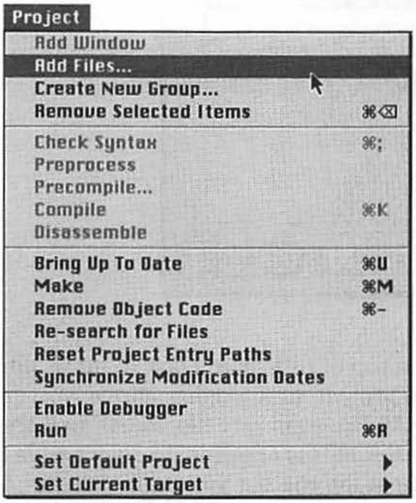
Adding a File to a Project

After you create a resource file, you can add it to your project. You can direct CodeWarrior to place a newly added file in any area of the project window where you want the file to go. Move the cursor over any file in the area you want the file to end up, and click the mouse button once. I want to add a resource file to the project, so I want the file to end up in the area that CodeWarrior has labeled Resources. In this next figure, I click the SillyBalls.rsrc file name in the project window.

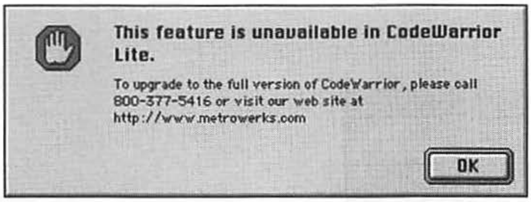


File	Code	Data		
Sources	0	0	•	•
SillyBalls.c	0	0	•	•
Resources	0	0	•	
SillyBalls.rsrc	n/a	n/a	•	
Mac Libraries	0	0	•	
MSL Runtime68K.Lib	0	0	•	
MacOS.lib	0	0	•	
MathLib68K (2i).Lib	0	0	•	
ANSI Libraries	0	0	•	
MSL C.68K (2i).Lib	0	0	•	
MSL C++.68K (2i).Lib	0	0	•	
MSL SIOUX.68K.Lib	0	0	•	
8 files	0	0		

You only need to use some of the dozens of menu items available in CodeWarrior. Here's the first menu item you use:



The Add Files menu item in the Project menu allows you to — ready for this? — add files to your project. Here I add the MyProgram.rsrc resource file to the MyProgram.mcp project. After you select Add Files, you see this dialog box if you are using CodeWarrior Lite:

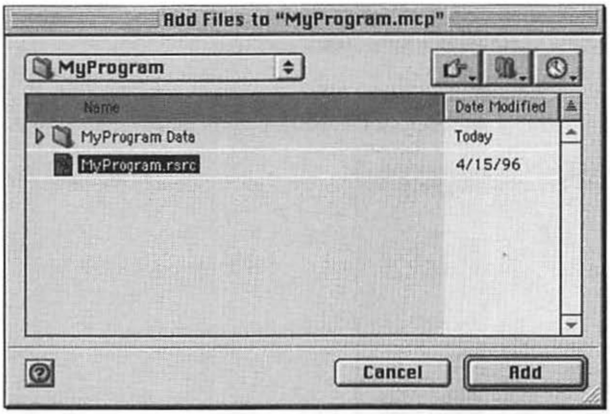


This feature is unavailable in CodeWarrior Lite.

To upgrade to the full version of CodeWarrior, please call 800-377-5416 or visit our web site at http://www.metrowerks.com

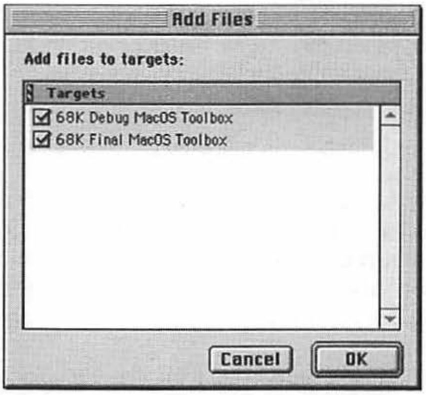
Darn! Foiled again by the limited version of CodeWarrior. Like creating a new project, adding files to a project is a no-no in the Lite version of CodeWarrior. Being able to add your own resource and source code files to an existing project would allow you to develop any program you want, and it would be a sneaky way of avoiding buying a real, completely functional compiler.

Here's what you see when you choose Add Files from the Project menu in the full-featured CodeWarrior:



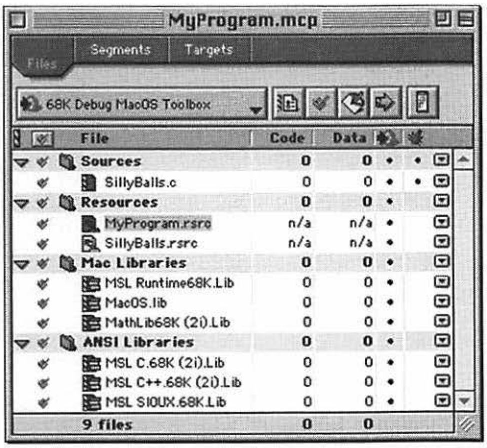
You use the pop-up menu at the top of this dialog box to move into the folder that holds the file to add to the project. In the figure above, you can see that I'm in the folder that holds the MyProgram.rsrc file. Recall that this is the MyProgram folder I created earlier in this chapter. To select a file to add to the project, click the file's name in the top list and then click the Add button.

You may notice that the MyProgram.mcp project file isn't named in the top list in the above figure even though the pop-up menu shows that I'm in the MyProgram folder. Don't worry — that project file is still in the MyProgram folder. CodeWarrior uses a *filter* to allow only certain types of files to be displayed in this dialog box. Only files such as libraries, resource files, and source code files (files that you'd want to add to a CodeWarrior project) are displayed. When you click the Add button, you'll most likely see a dialog box like the one shown here:



CodeWarrior Professional allows you to create different programs from the same project. Often a programmer will create a test version of a program first (known as a debug version — a version that is helpful in getting all the bugs, or errors, out of the code) and then later creates a final version of the program. You and I won't need to worry about any of this version business in this book. If you ever see this dialog box, go ahead and click the OK button to let CodeWarrior add your selected file to the project as it sees fit.

After clicking the OK button, the resource file is added to the project. Here's how the window of the MyProgram.mcp project looks after I add the MyProgram.rsrc file to it:



File	Code	Data		
Sources	0	0	•	•
SillyBalls.c	0	0	•	•
Resources	0	0	•	
MyProgram.rsrc	n/a	n/a	•	
SillyBalls.rsrc	n/a	n/a	•	
Mac Libraries	0	0	•	
MSL Runtime68K.Lib	0	0	•	
MacOS.lib	0	0	•	
MathLib68K (2i).Lib	0	0	•	
ANSI Libraries	0	0	•	
MSL C.68K (2i).Lib	0	0	•	
MSL C++.68K (2i).Lib	0	0	•	
MSL SIOUX.68K.Lib	0	0	•	
9 files	0	0		

Removing a File from a Project

After adding MyProgram.rsrc, the newly added resource file joins the SillyBalls.rsrc file in the Resources section of the project window. Remember, the file ended up in this particular spot because I specified the location by clicking the SillyBalls.rsrc file name before choosing Add Files from the Project menu. Now that the MyProgram.rsrc file is added to the project, I can remove the SillyBalls.rsrc placeholder file. This file serves as nothing more than a reminder to add your own resource file — it doesn't hold anything of use to your project.

I begin by first clicking once on the SillyBalls.rsrc name to highlight it, and then I choose Remove Selected Items from the Project menu. I say *I'll* do this because *you* won't be able to do this if you are using the CodeWarrior Lite version of the software. Removing files from a project, even placeholder files, is something that CodeWarrior Lite can't do.

WARNING!

Clicking a file name in the project window and then choosing Remove Selected Items from the Project menu is a technique that can be used to remove any file from a project. But you don't want to just go about haphazardly removing files, now do you? In particular, you want to make sure all of the library files that CodeWarrior placed in the project remain in the project.

That takes care of adding and removing a file to a CodeWarrior project. In Chapter 10, I show you how to create a source code file and add it to the project. Don't worry if you don't know anything about the C language. You can still create a source code file and add it to a project without any knowledge of C.

Chapter 10

Creating Source Code Isn't Hard, Honest!

In This Chapter

- Opening an existing CodeWarrior project
- Creating a source code file (finally!)
- Adding a source code file to a CodeWarrior project
- Typing in the source code

Writing source code involves two steps. First, you have to be familiar with the basics: How to create a source code text file, how to make that file part of your project, and how to correctly type in the code. The second step is to know *what* to type in. In this chapter, I *show* you how to deal with the basics, and I *tell* you what to type in. At the end of the chapter, you should be comfortable with working with source code, and you'll have a source code file that you can find out how to compile in Chapter 11.

Opening an Existing Project

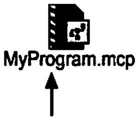
Unless you're a programming wizard, it's pretty unlikely that you'll ever create a new project, write all the source code for it, and turn it into a program in one sitting. So opening projects is something you want to get used to doing.

If you are running the CodeWarrior program, exit it by choosing Quit from the File menu.

In Chapter 9, you start CodeWarrior by double-clicking its icon. Now I'm going to show you a second way to run CodeWarrior. You can launch, or start, CodeWarrior directly from the icon of any project created with CodeWarrior. Double-clicking the icon of a project launches CodeWarrior *and* opens that project.

To *launch* a program means to run it. Then why not just say *run* the program? For the same reason the Macintosh interface consists of icons and windows, not just text — Mac users like a little fun and novelty in their computers. And the terminology surrounding the Macintosh sometimes reflects this mood.

If you're using CodeWarrior Professional, you know where to find the MyProgram.mcp project you created in Chapter 9; it should be in the MyProgram folder you created along with the project. If you're using the Lite version of CodeWarrior that comes with this book, you can find a version of the MyProgram.mcp project in the C10 MyProgram folder in the ...*For Dummies* Examples folder that you copied from the CD-ROM to your hard drive. In either case, the MyProgram.mcp project icon looks like this:



You can launch CodeWarrior and open the MyProgram.mcp project in one shot by double-clicking the icon of this project. Go ahead and do that now.

If CodeWarrior is already up and running, you can also open an existing project by using the Open menu item found in the File menu. If a project window is open, first click its close box to close it. (There, now you just found out how to close an open project, too!) CodeWarrior Lite also allows you to open a project this way, so feel free to experiment with this menu item.

Now that you know the shortcut for opening a project, it's time to move on to something more productive, such as creating and adding a source code file to a CodeWarrior project.

Working with a Source Code File

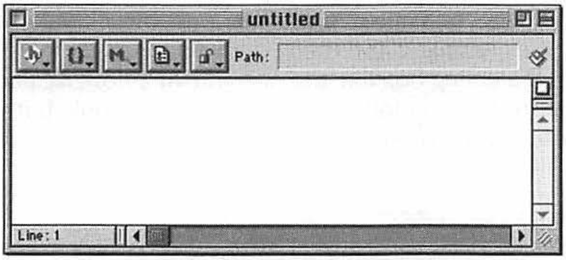
Are you ready to get a feel for working with source code files? In Part IV, I fill you in on the details of discovering what source code to type in. In this section, I show you how to create and work with a source code file.

Creating a source code file

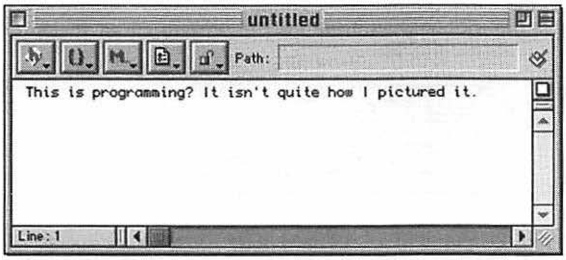
A source code file is a text file. It shouldn't be surprising that the process of creating a source code file is not unlike that of creating a text file. Just as you'd create a new text file in a text editor (such as SimpleText), in CodeWarrior, you choose New from the File menu to create a new source code file.

If you attempt to open a new file in the Lite version of CodeWarrior, you see a nasty dialog box that tells you CodeWarrior Lite won't allow that action to take place. I've got a way around that (sort of) that I tell you about just ahead.

If you choose New from the File menu of the full-featured version of CodeWarrior, an empty window like this one opens:



Looks just like a window in a Macintosh text editor or word processor, doesn't it? And if you start typing, it behaves like one, too:



```
This is programming? It isn't quite how I pictured it.
```

Just as I promised you several times, a source code file is nothing more than a text file.

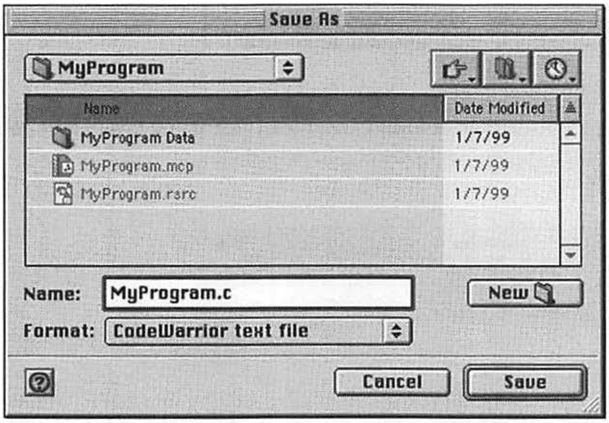
Saving the source code file

You probably already guessed that the words I typed in the window in the preceding figure don't qualify as source code. I can remedy that. But first I want to save the file and then add the file to the project — creating a new text file with the New command does not automatically place the file in your project.

TIP

You should add the new file to your project right away so that you don't forget to do it. You don't have to type in all your source code before saving a file or adding it to your project. After the file is added to the project, you can edit it at any time.

You're familiar with the Save and Save As commands in the File menu — they work identically to those found in any Mac text editor or word processor. The Save command saves a file, and you should use it periodically to save changes and additions to your source code. The Save As command saves a file, but first it allows you to name it. CodeWarrior Professional users create a new file with the New command and then choose Save As to name the file. Doing that brings up a dialog box like the one shown below. Again, you readers using CodeWarrior Lite can follow along here in the book, but you won't be able to do this with your version of the program:



Name	Date Modified
MyProgram Data	1/7/99
MyProgram.mcp	1/7/99
MyProgram.rsrc	1/7/99

Name: MyProgram.c

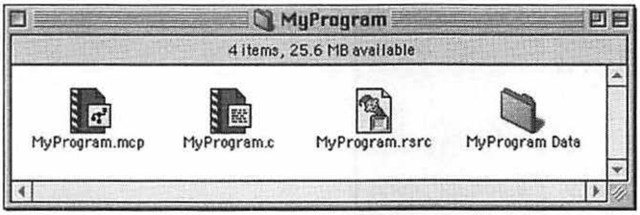
Format: CodeWarrior text file

To keep things organized, save the source code file to the same folder where you keep your project and resource file. Use the menu at the top of the Save As dialog box to move to the folder containing the project. When the folder name is shown at the top of the dialog box (as it is in the preceding figure), you can type in a file name.

Give your source code file any name you want, but make sure it ends with a period followed by the letter *c*. That ending to the file name is important. When you add a file to a project, CodeWarrior only allows you to add certain types of files. That makes sense, because you can't expect CodeWarrior to know what to do with, say, a letter to your Aunt Millie, right? One of the types of files CodeWarrior is happy to work with is, of course, a file that holds C language source code. The period and the letter *c* at the end of a file's name tells CodeWarrior that the file contains C language code. After typing in the name, click the Save button.

This point is worth repeating. A source code file is a text file. But CodeWarrior doesn't want to work with just any old text file — it wants source code. Without the *.c* at the end of the file name, you won't be able to add the file to the project. In fact, when it comes time to add the file to the project, a C language source code file with no *.c* at the end of its name won't even appear in the scrollable list of files to add. It will have seemingly disappeared! What happens then? You panic. You think your hours of typing once-in-a-lifetime ideas are wasted, gone forever. You end it all by jumping out of your window. I can't have that on my conscience — that's why I've devoted all this space to naming files.

The following figure shows how the MyProgram folder looks with a project file, a resource file, and a source code file all saved in it. If you're following along with the Chapter 10 folder that came on this book's CD-ROM, then the folder is named C10 MyProgram.



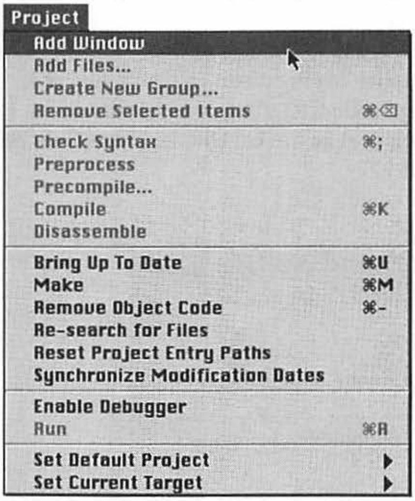
Take a close look at the figure above. Every Mac program you create has this configuration — a folder that holds a project data folder, a project file, a source code file, and a resource file.

All Macintosh programs have names, and all yours will, too. While it makes sense to give your program the same name as the source code file that holds the code used to create it, its name can be different. But don't spend a whole lot of time thinking of a clever, catchy name right now — you've got work to do!

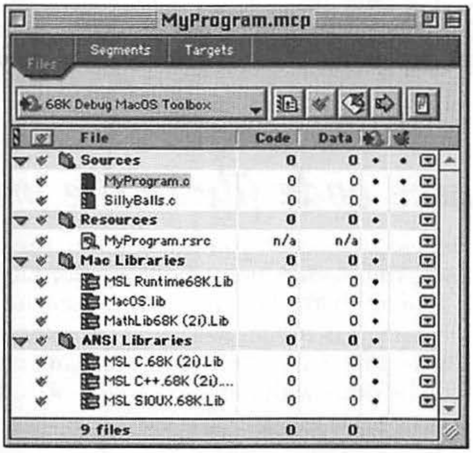
Adding the source code file to the project

If your source code file is named and saved, you can add it to the project. Here's some really good news. You readers using CodeWarrior Lite should be pleased to hear that this is absolutely the last task in this book for which the full-featured version of CodeWarrior is required. *Everything* else that I do in this book, such as compiling, testing, saving, running code, and making a standalone Mac program, can be done with the Lite version of CodeWarrior.

Using CodeWarrior Professional to add a source code file to a project is a simple task. First, click once on the SillyBalls.c name in the project window. This name serves as a placeholder that shows you a good spot in the project window to add your own source code file. Next, click once anywhere on the window that holds the source code. (That's to make sure that it is the *active*, or frontmost, window.) Then choose the first menu item in the Project menu, which is the Add Window item:



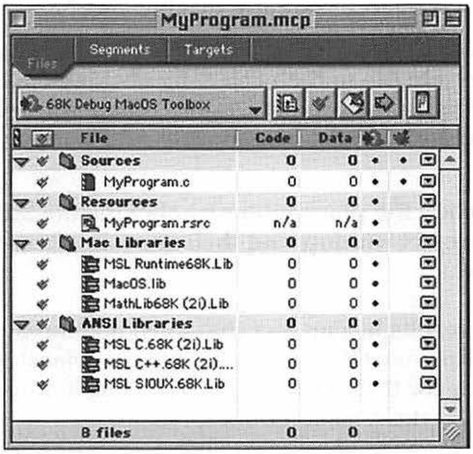
When you choose Add Window, CodeWarrior adds the open source code file to the project window right by the SillyBalls.c placeholder name:



File	Code	Data		
Sources	0	0	•	•
MyProgram.c	0	0	•	•
SillyBalls.c	0	0	•	•
Resources	0	0	•	
MyProgram.rsrc	n/a	n/a	•	
Mac Libraries	0	0	•	
MSL Runtime68K.Lib	0	0	•	
MacOS.lib	0	0	•	
MathLib68K (2i).Lib	0	0	•	
ANSI Libraries	0	0	•	
MSL C.68K (2i).Lib	0	0	•	
MSL C++.68K (2i)...	0	0	•	
MSL SIOUX.68K.Lib	0	0	•	
9 files	0	0		

Depending on which version of CodeWarrior Professional you're using, choosing Add Window may result in the display of an Add Files dialog box. If you see such a dialog box, go ahead and click its OK button. Chapter 9 mentions that this dialog box gives you the option of adding a file to only specific targets, or versions of the program you'll be creating. You're safe in adding files to all the targets.

Now you can remove the placeholder from the project window by clicking on the SillyBalls.c name in the project window and then choosing Remove Files from the Project menu. After you owners of CodeWarrior Professional do that for the SillyBalls.c placeholder, the MyProgram.mcp project window looks like this:



File	Code	Data		
Sources	**0**	**0**	●	●
MyProgram.c	0	0	●	●
Resources	**0**	**0**	●	
MyProgram.rsrc	n/a	n/a	●	
Mac Libraries	**0**	**0**	●	
MSL Runtime68K.Lib	0	0	●	
MacOS.lib	0	0	●	
MathLib68K (2i).Lib	0	0	●	
ANSI Libraries	**0**	**0**	●	
MSL C.68K (2i).Lib	0	0	●	
MSL C++.68K (2i)....	0	0	●	
MSL SIOUX.68K.Lib	0	0	●	
8 files	**0**	**0**		

Now your project is all set up, and you are ready to enter some source code.

Reviewing the Creation of a CodeWarrior Project

You've come a long way! Just to make sure you have all this CodeWarrior project stuff down pat, here's a quick review of the steps for creating a CodeWarrior Professional project to be used as the basis of a new Macintosh program:

1. **Run CodeWarrior Professional, of course!**
2. **Create a new project by choosing New from the File menu.**

3. **Launch ResEdit by double-clicking the SillyBalls.rsrc placeholder in the project window.**
4. **Create a new resource file and add the appropriate resources to it.**
5. **Add the new resource file to the project by returning to CodeWarrior, clicking on the SillyBalls.rsrc name in the project window, and choosing Add Files from the Project menu.**
6. **Remove the resource file placeholder by clicking the SillyBalls.rsrc placeholder in the project window and then choosing Remove Files from the Project menu.**
7. **Create a new, empty source code file by choosing New from the File menu.**
8. **Name and save the new source code file by choosing Save As from the File menu.**
9. **Add the new source code file to the project by clicking once on the SillyBalls.c placeholder in the project window, clicking once on the source code window, and, finally, by choosing Add Window from the Project menu.**
10. **Remove the source code file placeholder by clicking the SillyBalls.c placeholder in the project window and then choosing Remove Files from the Project menu.**

Following these ten steps results in a project that's just about ready to be turned into a Macintosh program. Of course, a couple of trivial steps remain, like writing the source code for the program and compiling it. Never fear. I cover both of these topics in the remainder of Part III.

Entering the Source Code

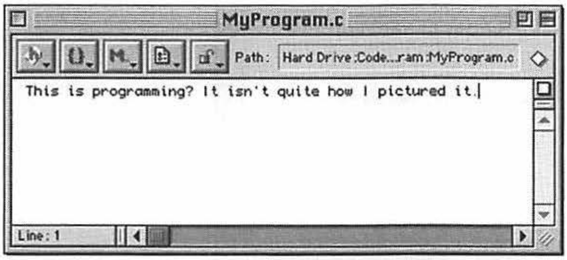
In Chapter 5, I introduce a very short Macintosh program that I call ExampleOne. Because I like to stick with familiar territory whenever possible, I use that very same source code in this chapter's example.

Opening a source code file

When a source code file is open, the source code appears in a standard window that has a close box in the title bar. You can close a source code file anytime. Once the file has become part of a project through the use of the Add Window menu option, it remains part of that project even if you close the file. Try closing an open file by clicking in its close box, or "go-away box," as some people call it.

To reopen a source code file that's part of a project, double-click the file's name in the project window.

While feverishly writing this chapter, I closed the MyProgram.c file so that it wouldn't be in my way on the screen. I used the technique of double-clicking its name in the project window to reopen it. When I opened it, I was back to where I left off:



```
This is programming? It isn't quite how I pictured it.
```

TIP

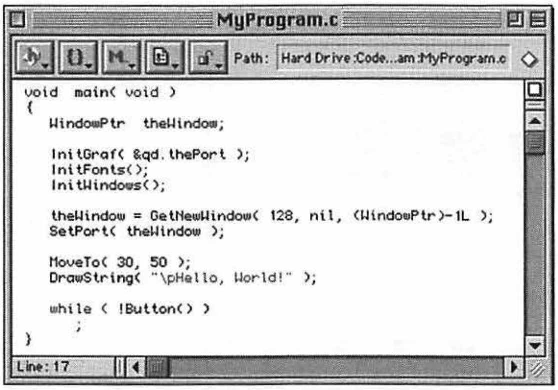
Recall that earlier in this chapter I stated that CodeWarrior Lite users can still view and add source code to an empty source code file — even though CodeWarrior Lite doesn't allow you to create new files. How do you perform this mysterious feat? If you haven't done so already, open the MyProgram.mcp project found in the C10 MyProgram folder (you can simply double-click its icon to open the folder). Now open the source code file by double-clicking the MyProgram.c name in the project window. I intentionally left this file empty so that you could experiment with it and then type in and save your own code.

If your window looks something like mine, or you have some other gibberish typed in it, get rid of the text now. Use any of the standard editing techniques you're familiar with to get rid of the text: You can click the mouse button, drag over all of the text to highlight it, and then release the mouse button and choose Cut from the File menu. Or, after highlighting the text, you can just press the Delete key. If you have a little time to kill, you can click the mouse at the very end of the text and just backspace over all of it. Whatever works for you is fine with me. Just make sure that you have an absolutely clean slate.

Typing in the code

The trick to becoming a programmer isn't becoming an accomplished typist — it's knowing *what* to type. In Part IV, I cover what to type when you create source code and what it means. But even without knowing quite what the source code means, you can still get a good feel for the process of writing code.

Click the mouse in the MyProgram.c window and type in the following sample source code. I show the code in a window so that you can compare it to your own window:



```
void  main( void )
{
   WindowPtr  theWindow;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   SetPort( theWindow );

   MoveTo( 30, 50 );
   DrawString( "\pHello, World!" );

   while ( !Button() )
      ;
}
```

As you type in the source code, you may have a few questions about what's important and what isn't. While some rules about writing source code are written in stone, other factors are left to the whim of the programmer. Now then, what should you be concerned with as you enter the code? Keep these points in mind:

- Most lines, but not all, end with a semicolon.
- Each parenthesis is crucial.
- Each brace is crucial.
- Proper use of uppercase and lowercase is essential in all words.

Relax. Not everything is critical when typing in source code. For example, you don't have to worry about:

- The number of spaces or tabs between words.
- The number of spaces or tabs used to indent lines.
- Whether or not blank lines are used occasionally.
- Whether or not spaces are between words that lie within parentheses.
- The font or the size of text used to display the source code.

If you've typed all the code, choose Save from the File menu to save it — you don't want your efforts to go for naught if your Mac unexpectedly crashes! After your code is all typed in, the next step is to compile it. I've talked about compiling several times already; it's about time you actually got to try it! I walk you through compiling some code in Chapter 11.

Chapter 11

Compiling and Running Your Source Code

In This Chapter

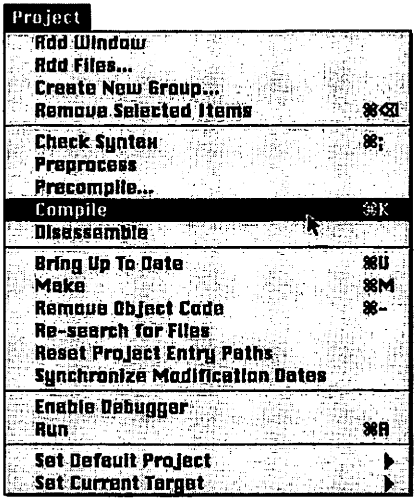
- Compiling a source code file
- Solving the problem if compiling doesn't work
- Testing a program by running it
- Turning source code into a program

You know what a compiler is, how to create a project, how to make a source code file, and how to enter the source code. (If you don't know these things, it's time to go back and read some important sections of this book.) You're only a few steps away from being face-to-face with a brand-new Macintosh program that you can call your own.

Compiling Your Code

Begin by opening your MyProgram.mcp project (you can read about opening a project at the start of Chapter 10). If you're using the full-featured version of CodeWarrior and you've created your own project, go ahead and open it now. If you're using the CodeWarrior Lite that comes with this book, open the Chapter 11 version of the MyProgram.mcp project, which you find in the C11 MyProgram folder.

Compiling is incredibly easy. After you open the MyProgram.mcp project, just tell the compiler which source code file to compile and then choose Compile from the Project menu:

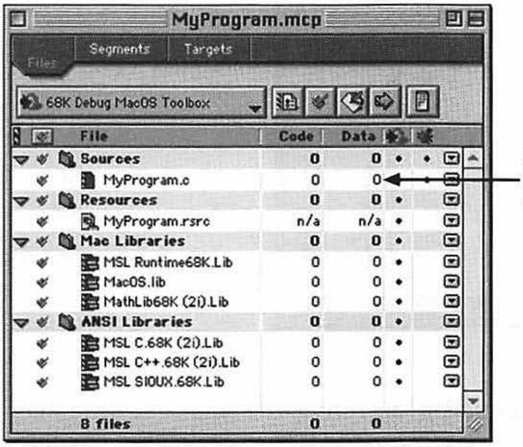


How do you tell CodeWarrior which source code file to compile? You can give CodeWarrior the signal in one of two ways. With your source code file open on the screen, make sure it's the frontmost file by clicking it. CodeWarrior then enables the Compile menu item so that you can compile the active source code file whenever you're ready. The second way to tell CodeWarrior which file you want to compile is to highlight the file's name in the project window. You can do that by clicking once on the name.

If the Compile option appears dim, you can't choose it. CodeWarrior won't *enable* this menu — that is, make it active — until you tell it what to compile. Makes sense, right?

Try to compile the MyProgram.c file. Use either of the techniques I describe in the preceding paragraph to make sure that CodeWarrior knows this is the file you want to compile.

Before you choose Compile, take a look at the numbers in the MyProgram.c row of the project window. They should both be zero:



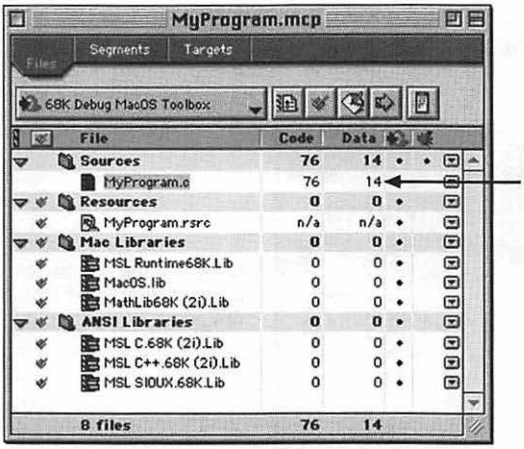
File	Code	Data		
Sources	0	0	•	•
MyProgram.c	0	0		•
Resources	0	0	•	
MyProgram.rsrc	n/a	n/a	•	
Mac Libraries	0	0	•	
MSL Runtime68K.Lib	0	0	•	
MacOS.lib	0	0	•	
MathLib68K (2i).Lib	0	0	•	
ANSI Libraries	0	0	•	
MSL C.68K (2i).Lib	0	0	•	
MSL C++.68K (2i).Lib	0	0	•	
MSL SIOUX.68K.Lib	0	0	•	
8 files	0	0		

Note that the numbers in the MyProgram.c row are zeros.

If the numbers in your project window aren't zeros, don't panic! It simply means that you or someone else already compiled some or all of the files in the project. That's okay — it just means you're a little bit ahead of me. Go ahead and choose Compile from the Project menu.

What happened?

If you made a mistake when you typed the source code, CodeWarrior responds by displaying a window titled Errors & Warnings. If you see that window, you should refer to the section "Can you type? The compiler lets you know" in this chapter. If you don't see that error message window, then things certainly didn't seem very dramatic, did they? If you typed all the code in correctly (or if you used the MyProgram.mcp project from the C11 MyProgram folder, which includes a MyProgram.c file with the code typed in), it may seem like not much at all happened, but the compiler really did do some work. Look at the project window. Some of the numbers change:



File	Code	Data
Sources	76	14
MyProgram.c	76	14
Resources	0	0
MyProgram.rsrc	n/a	n/a
Mac Libraries	0	0
MSL Runtime68K.Lib	0	0
MacOS.lib	0	0
MathLib68K (2i).Lib	0	0
ANSI Libraries	0	0
MSL C.68K (2i).Lib	0	0
MSL C++.68K (2i).Lib	0	0
MSL SIOUX.68K.Lib	0	0
8 files	76	14

Note that the numbers in the MyProgram.c row are no longer zeros.

So what *did* happen? CodeWarrior turned the source code in the MyProgram.c file into *object code,* which is code that the computer understands. If you have the MyProgram.c file open, though, you notice that the source code is still sitting in the file right there on the screen, unchanged and very readable. That's because the compiler does its work behind the scenes. It made a *copy* of your source code file and worked with that. How can you be sure it really did that? Look at the columns of numbers in the project window. The number in the row with MyProgram.c changed. The new numbers reflect the size of the code that is created from the source code file.

TIP

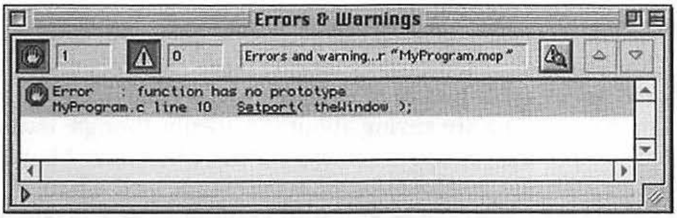
Don't waste your time keeping track of the exact numbers in the project window! I only point out these numbers to prove to you that compiling the MyProgram.c file really does produce a result. Understanding exactly what the values in the Code and Data columns mean only becomes important far, far down the programming road.

Compiling code doesn't actually *execute* — or run — the code. For example, the source code in the MyProgram.c file is supposed to put up a window and write a line of text to it. But it doesn't do that during the compiling stage. To get the code to run, you need to turn the code into a Macintosh program that shows up on your hard drive after you quit CodeWarrior. To see how to perform this feat, see "Running and Building Sounds Like Quite a Workout!" later in this chapter.

So that's it for compiling? Yup, that's it. Unless you make a *typo* — a mistake in typing the source code. Then you need to do some problem solving.

Can you type? The compiler lets you know

After choosing Compile from the Project menu, a window with a message in it *may* pop open on your screen. This is not good. But it also isn't the end of the world. This window only appears if you make one or more mistakes when typing your source code. When the compiler comes across a mistake, it makes a note of it in this window. The window looks like this, though the message (or messages) in it may be different depending on the error:

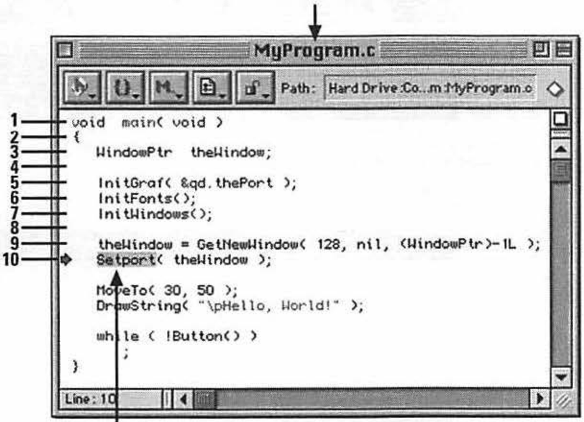


```
Error   : function has no prototype
MyProgram.c line 10   Setport( theWindow );
```

If you made a mistake entering source code, the Errors & Warnings window gives you a hint as to what went wrong. To get the above message, I purposely typed in `Setport` rather than `SetPort`. That is, I used a lowercase *p* rather than an uppercase *P*. And you thought I was being picky when I said that you had to be careful about upper- and lowercase!

The Errors & Warnings window tells you the file name and the line number at which the error occurred. My error is in the file MyProgram.c, at line 10. If I count down ten lines from the top of the source code file (including blank lines) I should come across the error. Take a look to see whether that's true:

Name of the file with the error.



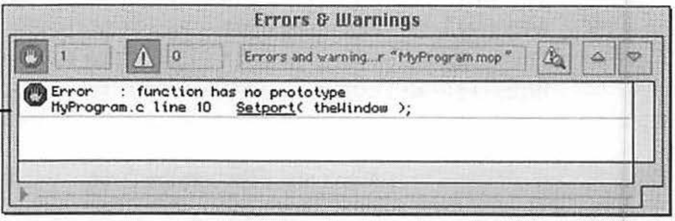
```
void  main( void )
{
   WindowPtr  theWindow;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   Setport( theWindow );

   MoveTo( 30, 50 );
   DrawString( "\pHello, World!" );

   while ( !Button() )
      ;
}
```

Here at line 10 is the letter *p* that the compiler finds offensive!

In Chapter 10, I say that blank lines don't count, yet in the preceding figure you can see that the compiler and I include two of them in the line count. Let me clarify. Blank lines don't count for anything when writing source code — you can put them anywhere and CodeWarrior won't complain. CodeWarrior only keeps track of the blank lines when reporting an error to you, which is very handy, because counting the blanks gives you a much more direct indication as to where the error took place.

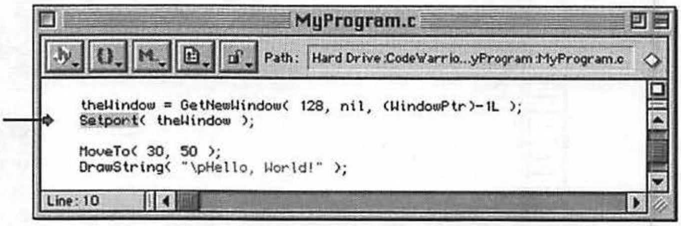
As the preceding figure shows, the error is in fact on line 10. The compiler is hoping to see `SetPort`, but instead, it finds `Setport`. A trivial difference to us humans, but not to a computer.

Counting lines of code is tedious, especially if the error occurs well past line 10. I counted lines earlier just to demonstrate that I can in fact count to ten — contrary to what people are saying about me. Really, though, I want you to realize that the compiler keeps track of your code by lines. What if an error occurs on line 103, and you just don't have the time or the patience to count all those lines? Or even worse — you start counting, but you get distracted at line 98 and have to start over? Don't worry. CodeWarrior provides an easier way to reach the problematic line of code. Just double-click anywhere on the highlighted section of the Errors & Warnings window. CodeWarrior scrolls your source code file right to the offensive line of code. CodeWarrior also adds an arrow and some highlighting to the line in which you made the mistake just to really rub it in your face:



```
Error    : function has no prototype
MyProgram.c line 10   Setport( theWindow );
```

Double-clicking anywhere on the highlighted area...



```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
Setport( theWindow );

MoveTo( 30, 50 );
DrawString( "\pHello, World!" );
```

...highlights the line with the error.

Yes, programmers really say things like *offensive* and *offending* when referring to a line with an error. Hey, don't shake your head at me — I didn't invent the terminology!

If you're reading this section because you encountered the Errors & Warnings window while compiling MyProgram.c, you now know exactly how to find your mistake. Feel free to go the error in your source code file and correct it. You can compare your source code to the figure near the end of Chapter 10, which is the figure that you used to initially type in this source code, to find out what you erroneously typed. If more than one error is listed in the Errors & Warnings window, correct each one.

After correcting your mistake (or mistakes), you need to *recompile* the source code file. Recompile is just the fancy word for saying *compile it again.* You can compile the same source code file as many times as you want. CodeWarrior only saves the result — the object code — of your most recent attempt. If you've corrected your mistake, choose Compile from the Project menu to recompile the source code file.

Running and Building Sounds Like Quite a Workout!

After you successfully compile your source code, the computer can understand it. Now you want the computer to *run* the source code so that you can see that you really have created a program. You can use two methods to get the compiler to run your code: You can run your *code* from within CodeWarrior, or you can run the *program* that CodeWarrior *builds* for you.

In Chapter 9, I explained that CodeWarrior is actually a programming environment, which means that CodeWarrior is capable of doing more than just compiling source code. It also allows you to add files to projects and to edit and save those files. It's actually a project manager, an editor, and a compiler all rolled into one program.

Besides all of the aforementioned tasks, CodeWarrior can perform one other trick you'll find very useful. It can run your compiled code right from within the CodeWarrior environment. That is, while CodeWarrior can take your code and create a standalone application that you can run at any time, it also lets you run your code without forcing you to go to your Mac's desktop to double-click the icon of the application it creates. On the surface, this may not seem like such a big deal, but it is. And that's due to the nature of programming.

It would be nice to sit down at your computer, type in all the source code for a program, compile it once, and be done with the whole business. But that's seldom, if ever, the way it works. When you compile and run your program for the first time, you're bound to see something you don't like. Here are a few possibilities:

- A window is the wrong size.
- A menu doesn't do what you thought it would.
- The words you wrote into a window don't look quite right.

The list goes on and on. So what do you do when something isn't quite right? Start over? Of course not. You make a change to the source code, recompile it, and then run it again and see how things look this time around.

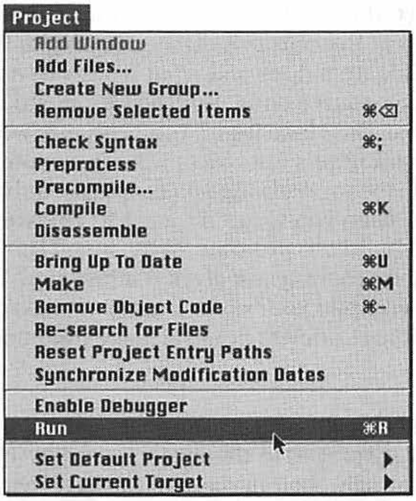
What about building a program? So far in this section, I've talked about letting CodeWarrior run your code. What about having CodeWarrior turn your code into a standalone, double-clickable application? In other words, when should CodeWarrior take your compiled code and *build* a program from it? You don't have to worry about that. Every time you run your code from within CodeWarrior, CodeWarrior builds, or makes, a new version of your program. Does that mean that if you run your code 20 times, you find 20 copies of your program sitting on your Mac's desktop? Not at all. Each time you run your code, CodeWarrior deletes the existing version of your program and replaces it with a new version. That way the program you find on your desktop is always the most recent version.

Running Code within CodeWarrior

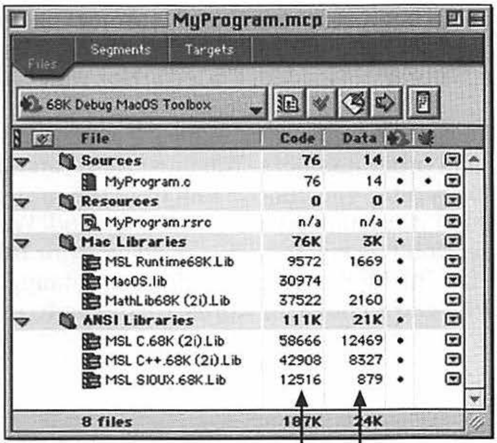
Knowing why you want to run source code is one thing, but actually doing it is another. You may also want to see an example of why you'd run, then rerun, your code. Your wish is my command.

Running the code

How to run your code? That's an easy one. Just choose Run from the Project menu:



When you make this selection, you probably notice some activity in the project window. Here's why. To build a program (which is one of the tasks of the Run menu item), all source code files in a project must be compiled, and all compiled source code, library code, and resources must be linked together. So while your project's window may start out with a bunch of zeros in its two columns of numbers, it finishes with all sorts of numbers in these columns after you choose Run from the Project menu:



File	Code	Data		
Sources	76	14	•	•
MyProgram.c	76	14	•	•
Resources	0	0	•	
MyProgram.rsrc	n/a	n/a	•	
Mac Libraries	76K	3K	•	
MSL Runtime68K.Lib	9572	1669	•	
MacOS.lib	30974	0	•	
MathLib68K (2i).Lib	37522	2160	•	
ANSI Libraries	111K	21K	•	
MSL C.68K (2i).Lib	58666	12469	•	
MSL C++.68K (2i).Lib	42908	8327	•	
MSL SIOUX.68K.Lib	12516	879	•	
8 files	187K	24K		

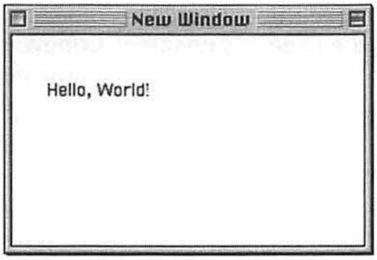
After selecting Run, these two columns have a variety of values in them.

Building a program is also referred to as *making* a program. And you may have even noticed, and guessed at the purpose of, the Make item that appears in the Project menu. This item does just what it says — it makes a standalone application from your source code. But the Run menu item already does that. So why choose the Make item? You probably won't, but there are a few cases when programmers choose to take this route. For example, if you write a program that does something so complicated that running it always takes several minutes, then you wouldn't want to choose the Run item if your intention is only to build the program. Why spend the time waiting while CodeWarrior builds the program *and* gives it a test run? While you probably won't have a need to compile your code and turn it into an application without running it from within CodeWarrior, the Make item does the trick.

After you choose Run from the Project menu, subsequent selections of Run have little impact on the values displayed in the project window. If you make changes to your source code, though, the numbers in the MyProgram.c row of the project window may change. That's because any time you make a change to the source code file, that file needs to be recompiled by CodeWarrior the next time you choose Run. Changes to your source code may affect the size of the compiled code, and thus the values in the project window may change.

What about all those libraries listed in the project window? If a library consists of precompiled code, why do the two numbers in the row of a library file start as zeros, and then change to other values when Run is selected — just as if the library is also being compiled. Very observant! You're correct in your thinking that a library doesn't need to be compiled by CodeWarrior. It does, however, need to be *loaded*. That just means that the project needs to figure out what's in the library and set up conditions so that it can access the code that's in the library. Once a library is loaded, CodeWarrior won't have to load it again.

Enough talk about zeros and other numbers. You must be scratching at the walls to run the code! Shortly after you choose Run from the project menu, your code runs. Why doesn't it run immediately? Because CodeWarrior has to compile your source code. Don't worry, though — CodeWarrior is fast. Your wait won't be long at all. For the MyProgram.mcp project, running the code means that a window opens and a couple of words are drawn to it, like this:



Hello, World!

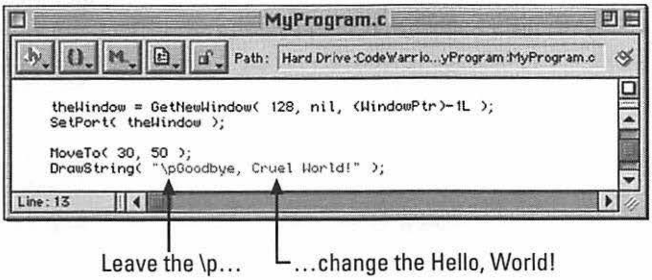
Mac programs really should contain a menu bar. The example program doesn't. If you're still running the program, you can verify this by looking at the blank menu bar on the screen. To read about adding menu bars to your programs, see Chapter 20. For now, I want to keep things really simple. Because no menu contains a Quit command, I designed the source code in such a way that the user simply clicks the mouse button to quit the program. If you are running the program, click the mouse to end the program.

When you quit the program that you're running from within CodeWarrior (or CodeWarrior Lite), you return to the compiler. The Hello, World! window disappears, and the menu bar returns to that of the compiler. You then get to pat yourself on the back — you successfully ran your first program!

Running it again. And again, and again . . .

The Run command is of most use while you are developing or making changes to your program. You should try making a simple change to the program and rerunning it.

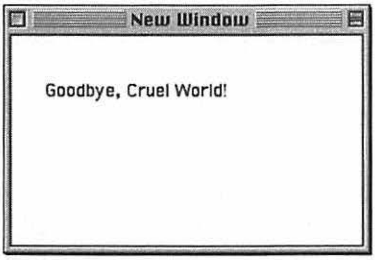
If the MyProgram.c source code file isn't open, reopen it now. Then go to the DrawString line and change the text that appears between the quotes. As tempting as it is to remove the funny-looking \p character, don't. But change any of the other text, as I do here:



```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );

MoveTo( 30, 50 );
DrawString( "\pGoodbye, Cruel World!" );
```

Leave the \p... ...change the Hello, World!

Choose Save from the File menu to save the change. Then choose Run from the Project menu. Because the source code has changed, CodeWarrior needs to recompile it. After just a moment, you see a window with the new text in it:



Goodbye, Cruel World!

Again, click the mouse to end the program and return to CodeWarrior. That brief example should give you a feel for the power and usefulness of the Run command. It allows you to change and test source code over and over again — and very quickly.

Checking Out the New Program

When you choose Run from the Project menu, CodeWarrior creates a stand-alone version of your program — a double-clickable application. CodeWarrior places this program in the same folder as the project from which it was created. In the case of the MyProgram.mcp project, that would be the C11 MyProgram folder. Go ahead and look for the program in the C11 MyProgram folder.

There it is! A program just like any you'd buy. Well, not *just* like any you'd buy. It doesn't do quite as much. But hey, if you can get someone to pay a few bucks for it, more power to you. Really, though. Does it behave as any other Mac program? Prove it to yourself by double-clicking its icon to run it. Then, after clicking the mouse to end the program, try copying the program to a disk. Or move the program out of the folder and try running it again. You see that your newly created program doesn't need the project, source code, resource file, or CodeWarrior to run. It is truly a stand-alone application.

You may notice a second new icon in your project folder, an icon with the same name as your project, followed by a .SYM ending. CodeWarrior creates this file when it compiles your source code. CodeWarrior uses it when the debugger is running. Because you don't use the CodeWarrior debugger in this book, you don't have to worry about this file; you can leave it in the folder or throw it in the trash. Don't be surprised to see it again, though. CodeWarrior creates a new version of this file every time you compile the MyProgram.c source code file.

That's it. In this chapter, you write source code, compile it, correct errors in the code, and build an application. I guess you don't need me anymore. Well, you can't get rid of me quite that easily. You're probably getting comfortable with the idea of using a compiler. But I bet you don't feel the same way about writing source code. After all, just blindly typing in what I tell you is much different than actually understanding what you're typing. After you get to know what those strange-looking combinations of characters in source code mean, you'll have this business of programming just about licked.

Part IV
Learning the C Language

In this part . . .

If you thought learning a computer language was a lot like learning a foreign language like, say, French or Japanese, you may have been a little intimidated. I don't blame you. I've never been good at foreign languages myself. Heck, the only thing I can say is *Eat my shorts!* in German. Obviously, a programming language can't be as complicated as a spoken language. It isn't, and here's why: A computer language usually consists of less than a hundred words. And to write a simple program, you only need to know a fraction of them.

In programming, it isn't how *many* words you know, it's how you *use* the words you do know. In this part, I describe several of the words in a computer language called C. And of course, I discuss how to use these elements of the C language. I also talk about the collection of miniprograms called the Toolbox. An easy-to-use Toolbox miniprogram, or function, turns a complex task, such as displaying a window, into a very easy one. The C language can be used to write programs for a number of operating systems (such as Macintosh, Windows 98, and so on). The Toolbox is used in conjunction with a language such as C to write programs exclusively for the Macintosh. While you *could* write a Mac program based only on the C language, mixing in the Toolbox allows you to write a Mac program that exhibits all the cool graphical elements (like windows, menus, and pictures) you'd expect to find in a Mac program.

Chapter 12

Choosing C over Other Languages

In This Chapter

- C is a good choice for Mac programmers
- C is called C for a reason
- C isn't the only programming language around

Many programmers know several computer languages. They may learn one language while in school, and another after they start working. Or they may learn different languages while working at different jobs.

You don't have to worry about any of that. Unless you're a professional programmer, one language is enough, and I think that C is the best one for you. So why pick the C language to do your Mac programming? And while I'm on the topic of languages, why is it called C? This chapter answers these questions and more.

Why Use C on the Mac?

Which is the absolute, one-and-only language to learn for programming the Mac? That depends on who you ask. Different programmers have different thoughts on that, and all of them are able (and willing) to provide valid arguments for their preference. In this book I use the C language, and in this section I explain why.

Arguing about a programming language with a programmer is a little like arguing about politics or religion — you make little headway, and you certainly can't win. If you *must* argue about it, my advice is to do so over the phone. That way you can cradle the phone on your shoulder, keeping both hands free, which allows you to ignore what the other person is saying and keep typing away at the C program on your Mac.

Everybody's using C

When the Mac came out a decade and a half ago, almost everyone programmed it using the Pascal language. Then, about a dozen years ago or so, many programmers switched to C. Most of them thought it was a little more powerful than Pascal, meaning that programmers thought they could get their programs to do more by using C.

If you decide to program in C, and if you need a little help at some point, you'll find a wealth of Mac programmers who can give you a hand. That's because just about all programmers know C. If you access the Internet, add `comp.sys.mac.programmer.codewarrior` and `comp.sys.mac.programmer.help` to your list of subscribed newsgroups. Post a Mac-related C programming question in either of these two newsgroups, and you're almost guaranteed to get a response or two within a day. The support you can garner from all these other C programmers may be enough reason to use it.

That's a good one! I just realized I said, ". . . *if* you need a little help at some point. . . ." Using the word *if* means you may, and then again, may not, need help. You *will* need help. I'm not criticizing you personally; it's just a fact. Programming can be a little tricky at times, and we all need to ask others for advice. I need to, and do, just about every day. One of the most important lessons you can learn about programming is to cast aside your fear of being ridiculed and *ask for help* whenever you don't understand something.

So, is it a sound practice to select a computer language just because everyone else is using it? Do you really want to be thought of as nothing more than a lemming heading for the sea? Of course not. Selecting C because the majority of programmers use it is only *one* of the reasons for choosing it — there are several other convincing reasons for using C. . . .

Other reasons for using C

To satisfy your curiosity, here are a few more reasons why you're being asked to learn C:

- It can be used to produce both mammoth, amazingly complex programs *and* short, extremely simple ones. This may not be important to you now, but if you stick with programming, you'll be happy that C is so versatile.
- The best-selling Macintosh compilers on the market allow you to program in C. The CodeWarrior compiler you're now familiar with is one.
- Because C is the current favorite, compiler vendors like Metrowerks update their C compilers sooner than they do compilers for other languages.

Why call it C?

I'd like to clear up a question that may have been nagging you ever since you heard about the C language — why is it called C? Because A and B were already taken! Very funny, right? But hold on — this time I'm serious. The C language was designed by Dennis Ritchie of Bell Labs back in 1972. Mr. Ritchie didn't, however, create it entirely on his own. He based it on the B language developed by Ken Thompson, an associate of his. So when Mr. Ritchie finished his new language, it was only natural to call it C.

- It's a highly *structured* language, meaning that it follows certain set rules. Learn the rules, and the rest falls into place. That's good for beginners.
- It is not as complicated a language as the next most popular Mac programming languages, the C++ and Java languages.
- All of the examples in this book — and on the book's CD-ROM — are in C.
- The single-letter name, C, sounds rather cryptic and mysterious, and so it impresses friends and coworkers when you inform them that you've mastered it.

Who could ask for more persuasion than all of the above? Hopefully you accept the notion that C is a good choice for Mac programmers. You can find out how to use C by reading the rest of the chapters in Part IV.

Those Other Languages

Pascal, C++, Java, FORTRAN, BASIC, and COBOL: you may have heard of them. They're all programming languages, just like C. Why so many different ones? There are a number of reasons.

Some languages are created with a specific use in mind. FORTRAN, for example, is used mainly by scientists and engineers who need a language that is geared heavily toward math. FORTRAN has a large number of code libraries that allow scientists to work easily with topics such as trigonometry and calculus. You can be thankful that you don't have to deal with FORTRAN when you're just starting to delve into Mac programming.

A *library* is a collection of code written by others. The code is placed in a library so it can easily be incorporated into your own programs.

Another reason for the proliferation of computer languages is time. Over time, computers have changed. So it makes sense that the languages used on computers would change, too. If the change to a computer is great, it is more practical to devise an entirely new language for it than to attempt to modify an outdated language.

One other reason for the existence of different computer languages reminds me of spoken languages. Computer languages, like spoken languages, are developed independently from one another by people living around the world in totally different environments. The English language and the Japanese language both exist because people in different parts of the world with different ideas developed them independently. Programming languages are no different. People developed different programming languages to suit their particular programming needs and circumstances.

As a Mac programmer, you can use just about any language you want. That's because a programming environment like CodeWarrior Professional actually includes a number of compilers — one compiler per language. So while I think you'll be quite content with the C language, I don't want to mislead you into thinking you have no choice in the matter.

When choosing a language, keep in mind that source code written in one language may look very different from source code written in another. That means that examples written in a language other than the one you choose probably won't be understandable to you. Because viewing example source code is a great way to learn about programming, you'll want to pick a language that has an abundance of sample code available. And that brings you right back to C. Because C is so popular, you'll have no problem finding books filled with C language examples (such as this one). And when people post example source code listings on the Internet (such as in the `alt.sources.mac` newsgroup), those listings are most often in C.

Chapter 13

Keeping Track of Things in C: Data Types and Variables

In This Chapter

- What a data type is
- What a variable is
- How data types and variables differ from one another
- Which C data types are most common
- How to keep track of whole numbers and fractional numbers
- How to give a variable a value

A computer program wouldn't be very useful if it didn't keep careful track of things. After all, that's primarily what a computer is used for — remembering things, calculating things, and ordering things. That's a lot of things for a computer to keep organized. Even a computer needs a system so that it can keep things straight.

In the last paragraph, I specifically used the word *things* instead of *numbers* because a computer works with much more than numbers. Many people think a computer only works with numbers, but it also works with words, pictures, sounds, movies, and more. And in the case of a computer with a GUI (graphical user interface), the computer also works with objects such as windows and menus. In this chapter, I take a look at the two primary programming tools a computer uses to keep track of all these things — data types and variables.

Data Types and Variables: Different, but Related

In Chapter 4, you get a peek at data types and variables. There, in my introduction to source code (see "Source Code"), I mention the `int`. The `int` is a C language abbreviation that stands for an integer (a whole number). The int is a symbol for an integer.

Besides these predetermined symbols, C allows you to make up your own symbols. In Chapter 4, I give the example of trucks as a made-up symbol. What I didn't say in Chapter 4 is that these two kinds of symbols — the predetermined C symbols and the symbols you make up yourself — are very different from one another, and that each type of symbol has its own special name.

Predetermined C symbols: Data types

Data is information, and information is what you're keeping track of when you program. Different types of data exist — numbers, words, pictures, and so on. So a computer language has different types of data to keep track of. Thus the phrase *data types*.

Just about everything can be divided into types. We're all people, but we can be divided into types by sex: the male sex, the female sex, and insects. Oops, let me start over: the male sex and the female sex. The same is true with numbers. Numbers without decimal points (whole numbers) are called *integers*. Because *integer* is such an incredibly long word, the C language refers to an integer as an int. Numbers with decimal points are called *floating-point numbers*. In C, a floating-point number is called — you guessed it — a float.

C language data types are *predefined* for you. By predefined, I mean that there are a set number of them you can use. The int and the float are just two of the dozens of types that exist. Don't worry, though; you only need to know about ten of the types in order to write some pretty interesting programs.

Do-it-yourself symbols: Variables

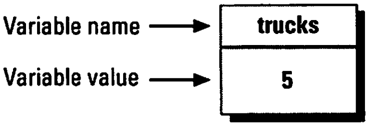
C lets you make up symbols of your own. I use trucks as an example back in Chapter 4. You create a symbol for the purpose of keeping track of something.

Say that you know an auto dealer, and you want to keep track of the number of trucks he has on his lot. You can make up a symbol like trucks. A made-up symbol like trucks is called a *variable*.

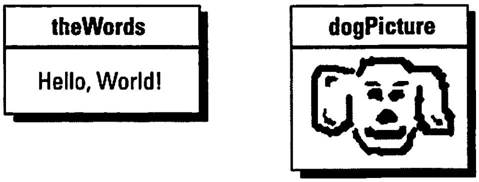
A variable always has a name; your auto dealer friend's variable is called trucks. A variable also always holds a value. Suppose the auto dealer you know has five trucks on his lot. You give the variable trucks a value of 5:

```
trucks = 5;
```

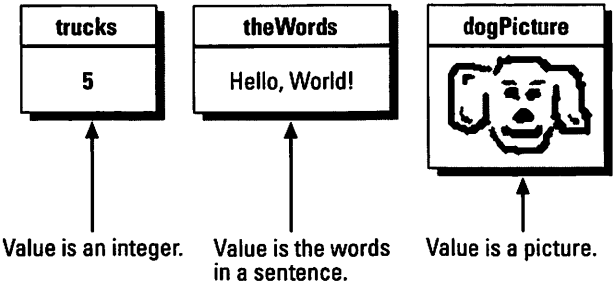
I describe how to *assign* a value to a variable in "Every Variable has a Value" later in this chapter. For now, take a look at a figure that shows the two parts of a variable — its name and its value:



The `trucks` variable has a value of 5. From my talks about numbers, you know that the number 5 is a whole number (an integer). Many of the variables you create hold integers, but a variable doesn't *have* to hold an integer value. In the figure below, I create two more variables. One is called `theWords`, and it holds, not surprisingly, some words. The second variable is named `dogPicture`. It holds a picture of a dog (at least, it's *supposed* to be a dog). Here they are:



You know that `trucks`, `theWords`, and `dogPicture` are all variables (well, variable names, actually). I said every variable has a value associated with it. In general terms, here are the values of each variable:



I draw different-sized boxes to hold each of the three variables. I had to because the things that the boxes hold are different sizes. That brings up an interesting point. How does the CodeWarrior compiler know how big this thing is that you are creating and holding in a variable? And how does it know what you're storing there? The answer is that you must *tell* the compiler what you're storing in a variable. That's where data types come in.

Every Variable Has a Type

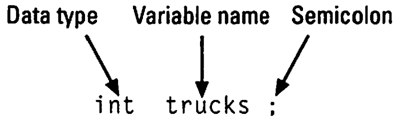
Every variable has a value, but not just any kind of value. You must let the compiler know what type of information (what kind of data) a variable holds. If you want a variable to hold an integer value, you tell the compiler:

```
int trucks;
```

This line of code does two things. It creates a variable named `trucks`, and it tells the compiler that `trucks` holds an integer — a value that is of the `int` data type. A line of code like the one above is called a *declaration*. You are declaring that a variable named `trucks` is being created, and you are declaring that it holds data of the `int` type.

Amazingly, sometimes computer terminology actually makes sense. When you come across a new term like *declare*, you may think of the normal definition of the word. Not the computer-related meaning, but one you use in everyday conversation. The dictionary says that to *declare* is to make clearly known, to state, or to announce openly. That definition fits squarely with the computer-related use of the word, which is to make a variable and its data type clearly and openly known to the compiler.

Here's the format you follow every time you declare a variable:



As you can see, to declare a variable, you first state the data type that the variable holds, then you name the variable, and then you end it all with a semicolon.

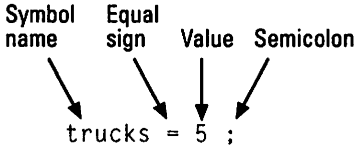
Every Variable Has a Value

Every variable has both a name and a value. In "Every Variable Has a Type," you see that the value of a variable must be of a particular type.

The declaration creates the variable and lets the compiler know what type of value the variable holds, but the declaration doesn't give the variable a value. That's done with an *assignment* statement like the following:

```
trucks = 5;
```

An assignment statement gives a variable a value. It assigns a value to the variable. But then, you may already know that — I stole the preceding line of code, and the following figure, straight out of Chapter 4:



```
trucks = 5 ;
```

This figure tells you that to give a variable a value, you first list the variable's name, followed by the equal sign, the value, and then you end the line with a semicolon.

Order Is Everything

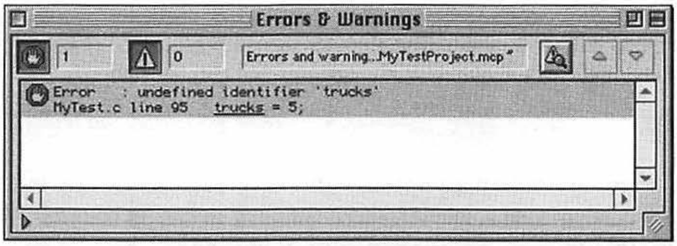
Now for today's quiz. Does the following source code look right to you?

```
trucks = 5;
int trucks;
```

You probably answered no. And you probably answered that way because you know by now that if I ask a question, I'm probably up to something. The answer is indeed no, and here's why. Before you can assign a variable a value, you must first declare it. When you compile your source code, the compiler looks at each line of source code *in order,* from top to bottom. So in the above example, the first thing the compiler would see is:

```
trucks = 5;
```

Because the compiler hasn't read the declaration on the second line yet, it doesn't know what the heck `trucks` is. Even if it is clever enough to assume that `trucks` is a variable, it doesn't know what *type* of data `trucks` holds. The compiler becomes so frustrated, in fact, that it stops compiling and pops open its Errors & Warnings window with a message like this:



```
Error   : undefined identifier 'trucks'
MyTest.c line 95   trucks = 5;
```

Why can't the compiler peek ahead somehow and see that `trucks` is declared on the very next line? Because you're dealing with software and computers, my friend. And with software and computers, *order is everything.* So the moral of this story is to declare your variables before using them. That is, before assigning them values. To remedy the offending code, simply switch the two lines around, like so:

```
int  trucks;
trucks = 5;
```

This simple change makes the compiler very happy, and it will perform its compiling duties with no further complaints.

Common Data Types

I said that C has dozens of data types. I also said that you'd only need a handful of them to create a Macintosh program. In the following sections, I mention a few of the most common data types. You encounter other data types throughout the book, but don't worry — I explain each data type when it comes up.

Data types for whole numbers

The three most common number data types are `int`, `short`, and `long`.

An `int` variable holds an integer, but not just any integer. The biggest number you can store in a variable of type `int` is 32,767. I know, I know. It would simplify things if an `int` held a number up to, say, 10,000 or 100,000. But it doesn't. You just have to use an `int` for your integer needs up to 32,767. That means that if you use an `int` to keep track of cars at an auto dealership, and car number 32,768 rolls onto the lot, you have to switch to another data type.

Why have any restrictions at all on how big a number you can store in a variable of a certain data type? Because a variable is held in computer memory. One variable occupies a certain number of bytes of memory. An `int` variable is stored in two bytes. In the system of math used by the computer, 32,767 is the biggest number that can fit in two bytes of memory.

A `short` variable is very similar to an `int`. Variables of either type hold whole numbers, and both hold numbers up to 32,767. So why have two data types that are essentially the same? Because there is a subtle difference between the two:

A `short` can *never* hold a whole number greater than 32,767.

An `int` can *sometimes* hold a whole number greater than 32,767.

Okay, okay. I know I said that an `int` can only hold whole numbers up to 32,767, but wait. The clever programmer can get variables of type `int` to hold numbers greater than 32,767; and the CodeWarrior compiler can be used to accomplish this magical feat. But such tomfoolery is a bit advanced for this book, and so for all practical purposes, you can consider the `short` and the `int` one and the same. Which one should you use? After looking at a lot of source code written by programmers at Apple, I realized that they seem to go bonkers about the `short` type. So that's the type I use in the remainder of this book.

If you want to create a variable that holds numbers larger than 32,767, use the `long` data type. A variable that is declared to be of type `long` can hold a number larger than two billion. That should definitely handle all your whole number needs.

I bet you just thought of something: If the `long` type can hold such a huge number, as well as much smaller numbers, why not just play it safe and *always* use a `long` ? Why ever bother with the `short` type? That's a good observation, but there is a reason why both types exist — memory management. In order to hold such a large number, a `long` reserves much more memory than a `short`.

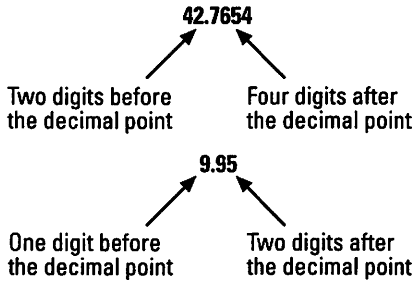
You still aren't sure whether to use a `short` or a `long`? You just have to think a little bit about what you want your program to count. If your program is keeping track of, say, the number of children a day care center takes care of, use a `short` — it's pretty unlikely the day care center will ever have more than 32,767 kids, right? On the other hand, if you're writing a program that keeps track of the number of people living in your state, use a `long`.

A data type for fractional numbers

You may find that the `short` data type and the `long` data type, both of which hold only whole numbers, are useful for most of your programming needs. After all, that auto dealership you're writing a program for won't be selling *fractions* of a truck. Regardless of how shady the salesman appears, he probably can't get away with that.

But if you were writing a program to track the dealership's inventory of automobiles, you would want to include the price of each car, and that involves fractions. More specifically, keeping track of prices involves decimals. If a number includes a decimal point, neither a `long` or a `short` cuts it.

A number that contains a decimal point is called a *floating-point number.* A number with a decimal point isn't required to have a certain number of digits before or after the decimal point. The position of the decimal point moves about — thus the term *floating-point.* For example:


42.7654
Two digits before the decimal point
Four digits after the decimal point
9.95
One digit before the decimal point
Two digits after the decimal point

Just as integers are represented by a C data type, floating-points are also represented by a C data type called the `float`. A variable that is of type `float` can hold a value much smaller than 1, such as 0.00000005. It can also hold an incredibly huge number, such as 5,000,000,000,000,000,000,000. And no, I don't know how to say that last number out loud, either!

You use the `float` data type for:

- Keeping track of things that don't come neatly packaged in whole numbers
- Keeping track of less than one thing — a fraction of something
- Keeping track of things that exceed the limit of a `long` type (which is just over 2 billion)

Common Variables

In the previous section, I showed you some common C data types. So it would make sense if I now showed you a few of the commonly used variables, right? Wrong! Data types are an established part of the C language. There are a limited number of data types, and each data type is spelled a specific way and used a specific way. Variables, on the other hand, aren't predefined for you. You make up variables according to your programming needs. Because of this fact, I can't give examples of variables that every programmer uses.

A Few Examples of Variables

In this chapter, I give you a head start on learning the C language by introducing two concepts very important to a programming language — declarations and assignments of variables. I assume that those two programming ideas made a little sense to you. If they did, here are a few examples.

Declaring variables

The declaration of a variable informs the compiler that you've just created a new variable. It also lets the compiler know what type of information the variable holds. Here are a few examples of variable declarations:

```
short  bookstores;
long   books;
float  price;
```

Most programs use more than one variable. For example, the three variables that I just declared could all appear in a single program that keeps track of books sales for this *...For Dummies* book. The `short` variable `bookstores` holds the number of bookstores that carry the book. The `long` variable `books` holds the number of copies of the book sold. The `float` variable `price` holds the cost of the book.

I made a couple of assumptions when I chose the data types for my variables. Variable `bookstores` is a `short`. Because the largest value a `short` can have is 32,767, I'm assuming that fewer than 32,767 bookstores will carry my book. For the `books` variable, I selected the `long` data type. If this book sells more than 32,767 copies, my program will still be able to keep track of sales. What about the `price` variable? Did I have a choice in what data type I would use? Not really. Book prices are often a dollar amount plus 95 cents, such as $29.95. This means two things:

- Because the number has a decimal point, I must use the `float` data type.
- You were tricked into thinking you paid less than you really did for the book.

The line of source code that declares a variable is called a *declaration statement*. You see this term in many computer programming books. I prefer to use the slightly less techno mumbo-jumbo phrase *declaring a variable.*

Assigning Values to Variables

You may already know what I'm about to say, but it's an important point, so please bear with me. You must declare a variable before you use it — that is, before you give it a value. Once declared, you give a variable a value by *assigning* it one. As an example, I want to declare some variables that pertain to my records for this book:

```
float  price;
short  bookstores;
long   books;
```

After I declare the variables, I can give any or all of them values:

```
price = 29.95;
bookstores = 295;
books = 40521;
```

Variable `price` is a `float`, so I can feel free to include a decimal point in its value. Variable `bookstores` is a `short`, so its value must be less than 32,767 — and it is. The `long` variable `books` can, but doesn't have to be, greater than 32,767. (Of course, I'm happy that the `long` variable is larger than 32,767.)

Notice that the value I gave `books` doesn't include a comma — like 40,521. Although the CodeWarrior compiler may let you get away with including a comma, your results won't be as expected. For example, if you try to assign `books` the value 40,521 (with the comma included), the resulting program would end up storing a value of 40 in variable `books`. Also notice I don't write:

```
price = $29.95;
```

The variable `price` was declared as a `float`, which is a number with a decimal. A symbol like a dollar sign isn't a part of a number; it's a label that precedes a number. Adding extra information, such as labels, causes the CodeWarrior compiler to complain. If you include a symbol like the dollar sign, CodeWarrior sends an error message when you compile your program.

The line of source code that assigns a variable a value is called an *assignment statement*. Like the term *declaration statement*, you see this phrase in some computer programming books. I just say that I'm *assigning a value to a variable* or I'm *giving a variable a value.*

In this chapter, I talk about data types, variables, declarations, and assignments — and you haven't even read the chapter on learning C. You're in way too deep to back out.

Chapter 14

Learning the Language — Just the Basics of C

In This Chapter

- Adding comments to source code
- Naming variables
- Using arithmetic with variables
- Looping to repeat source code
- Branching to allow choices

The entire C language can't be covered in a single chapter of a single book. But then, who needs the entire language? You just want to know enough to get a Mac program going, right? You're in luck — that much I can cover here and now.

Care to Comment on That?

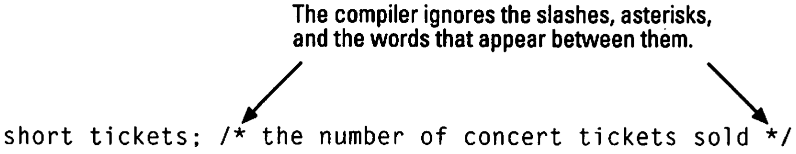
Sometimes source code looks confusing — I grant you that. You and I aren't the only ones who feel that way, though. Many, many people are befuddled by code, and so the people who make compilers came up with a way to help. Compilers let you add *comments* to your source code. These comments, or explanations, appear within your source code, but the comments aren't source code themselves. A comment is text that is readable by you, but is ignored by the compiler when it compiles the source code. Take this line of code for example:

```
short tickets;
```

What do you think the variable `tickets` is used for? You may have a few ideas, and one of them may even be right. But if I wrote the source code, and you're looking at it, you won't know for sure what `tickets` is unless you're a mind reader. Now look at the same line of code; this time I added a comment to it:

```
short tickets; /* the number of concert tickets sold */
```

Much clearer, right? And when you compile this line, CodeWarrior doesn't return an error message, even though the words that follow the semicolon obviously aren't valid C code. How does the compiler know the difference between code and comments? Glad you asked. Here's the answer:



The compiler ignores the slashes, asterisks, and the words that appear between them.

```
short tickets; /* the number of concert tickets sold */
```

A slash, immediately followed by an asterisk, signals the start of a comment. An asterisk, immediately followed by a slash, marks the comment's end. As an example, I add comments to a few lines of code I use in Chapter 14:

```
short  bookstores;/* number of stores carrying my book */
long   books;      /* number of books sold */
float  price;      /* price of a single copy of my book */
```

Sure, it takes a little extra time to add comments to your source code. But the clarity they add is well worth the effort.

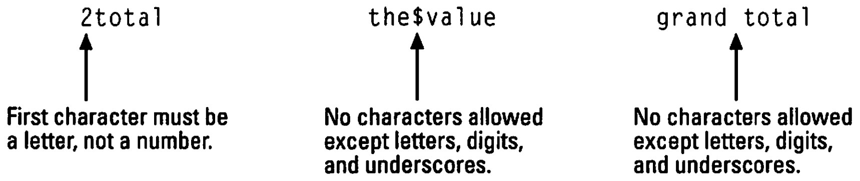
Variable Names

Your code may contain several types of variables. It's therefore time for the lowdown on variable names. (If you aren't sure what I mean by *several types of variables,* it's time for you to read Chapter 13.) Here are some guidelines for naming a variable:

- Use any combination of upper- and lowercase letters, digits, and underscores.
- Use a letter for the first character in the name.
- Keep in mind that C is case-sensitive.

According to these guidelines, `books`, `testScore`, `TotalScore`, `final_result`, and `total2` are all valid variable names.

Here's a look at a few *invalid* names and why they're illegal:


2total
First character must be a letter, not a number.
the$value
No characters allowed except letters, digits, and underscores.
grand total
No characters allowed except letters, digits, and underscores.

When picking a name for a variable, make it descriptive of what the variable represents. If a variable holds the total number of days you worked, name it `totalDays` rather than something obscure like `theNumber`. It makes it easier for someone else to figure out what your code is supposed to be doing. And, should you set unfinished code aside for a while and then return to it at a later date, using descriptive names makes things easier for you, too.

Operating without a License

Computers work with numbers, and the programs you write certainly do the same. A computer is a whiz at working with numbers, but you have to help it along by telling it what to do. That's what *operators* are all about. Take this simple case as an example:

```
trucks = 5;
```

To give a variable a value, you must *operate* on it. The word *operate* may sound a little pretentious for the simple act of assigning a variable a value, but it's not meant to. Here, *operate* just means *to work with.* A machine operator works with a machine, and a telephone operator works with telephone numbers, and no one thinks that they are pretentious. In C, certain symbols, such as the equal sign, are called *operators.*

In the previous line of code, the equal sign works with the variable `trucks` and the number 5. The equal sign is responsible for placing the number 5 in the variable `trucks`. Without it, the variable would not receive a value.

In C, the equal sign is called the *assignment operator.* The equal sign assigns a value to a variable. While the equal sign is by far the most commonly used C operator, you should know about some other types of operators, which I describe in the following sections.

Minimal Math

It's quiz time! In your Macintosh programming endeavors, which of the following objectives do you hope to accomplish:

A. Write a program that uses a menu and a window.

B. Write a program that disproves Einstein's Theory of Relativity.

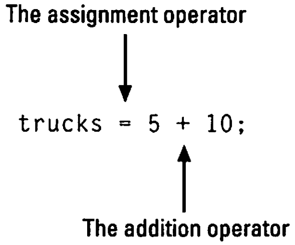
If you chose answer A, then the C language and this book are for you. If you chose answer B, then the C language may still be for you, but this book ain't. I stick to minimal math in this book; although C offers several operators that allow you to include a wealth of mathematical tricks and techniques in your programs, I only cover four of them in this book. Those four, however, should meet just about all your arithmetic needs while you are mastering the basics of Mac programming.

The addition operator

In programming, the act of addition is just as you'd expect it to be. You simply place the plus sign between two numbers to add them together. The result of the addition, the sum of the two numbers, is stored in a variable:

```
trucks = 5 + 10;
```

The above line of code performs two tasks. First, it adds the numbers 5 and 10. Second, it saves the result in the variable named `trucks`. Because two tasks, or operations, are performed in the line of code, two operators appear:



```
trucks = 5 + 10;
```

The *addition operator* is used to add two numbers together. That being the case, try to determine why this following snippet (group of code) is actual, real-live, functioning C code:

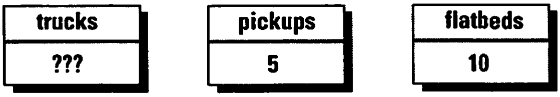
```
short pickups;
short flatbeds;
short trucks;

pickups  = 5;
flatbeds = 10;
trucks = pickups + flatbeds;
```

Let me interrupt myself here to point out that blank line between the variable declarations and the assignment statements in the preceding code. While not required, many programmers use *white space* in this manner. They feel that separating sections of code in this way achieves a little more clarity.

The last line in the preceding code appears to add two variables together, rather than add two numbers together. Keep in mind that a variable has a name *and* a value. The computer is more interested in the value of the variable. When you add two variables together, the values of the variables get added. Take a look at the source code up to, but not including, the line that adds the variables:

```
short pickups;
short flatbeds;
short trucks;
pickups  = 5;
flatbeds = 10;
```

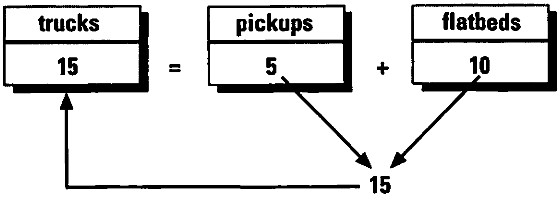
Here's a figure that represents the variables:



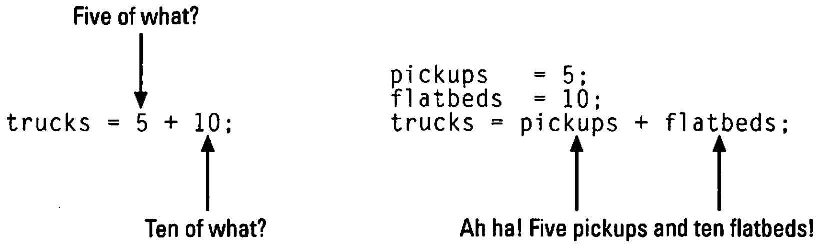
I put question marks as the value of `trucks` because I don't know what the value of `trucks` is — I haven't assigned it a value at this point. Now, the addition line:

```
trucks = pickups + flatbeds;
```

Watch what happens to the variables now:



What exactly is the purpose of using variables rather than just using numbers? The computer is great at keeping track of values, but people are better at keeping track of names:



```
trucks = 5 + 10;
```

```
pickups   = 5;
flatbeds  = 10;
trucks = pickups + flatbeds;
```

Now that you know that the compiler views a variable by the value it holds, you shouldn't be surprised to hear that all four of the following truck assignments give trucks a value of 15:

```
pickups  = 5;
flatbeds = 10;
trucks = 5 + 10;
trucks = pickups + flatbeds;
trucks = pickups + 10;
trucks = 5 + flatbeds;
```

The subtraction operator

As with the addition operator, the other three basic math operators are also a snap. To subtract one value from another and assign the result to a variable, use the minus sign, which is called the *subtraction operator:*

```
trucks = 10 - 5;
```

The subtraction operator works with variables just as the addition operator does. In the following examples, I add comments to provide you with the result of each subtraction operation:

```
pickups  = 5;
flatbeds = 10;
trucks = flatbeds - pickups;  /* The result of each of*/
trucks = flatbeds - 5;                  /* these four lines of code?*/
trucks = 10 - pickups;                  /* Variable trucks will*/
trucks = 10 - 5;                        /* have a value of 5 */
```

The multiplication operator

In C, multiplication is performed by placing the *multiplication operator* — an asterisk — between the two numbers:

```
totalDays = 7 * 10;     /* totalDays value will be 70 */
```

Take a look at a few examples using variables:

```
short daysPerWeek;
short weeks;
short totalDays;
daysPerWeek = 7;
weeks = 10;
totalDays = 7 * 10;
totalDays = daysPerWeek * weeks;
totalDays = daysPerWeek * 10;
totalDays = 7 * weeks;
```

The division operator

The *division operator* — a slash — is used to perform division in C. This operator divides the first number by the second:

```
dozens = 48 / 12;    /* dozens will have a value of 4 */
```

Like all the other operators, the division operator works on numbers, variables, or combinations of both:

```
short totalDozen;
short eggs;
short oneDozen;

eggs = 48;
oneDozen = 12;
totalDozen = 48 / 12;
totalDozen = eggs / oneDozen;
totalDozen = eggs / 12;
totalDozen = 48 / oneDozen;
```

Notice in the preceding examples the result is a whole number — one that has no decimal, or fractional, part to it. That is, 12 divides into 48 exactly 4 times. What do you suppose the value of `totalDozen` would be if `eggs` had a value of 51 and `oneDozen` still had a value of 12? Like this:

```
eggs = 51;
oneDozen = 12;
totalDozen = eggs / oneDozen;
```

My calculator tells me that 51 divided by 12 is 4.25. But totalDozen, which is defined to be a variable of type short, can only hold whole numbers. The result of such an "uneven" division is that the fractional part is simply ignored, or forgotten. That means that totalDozen would still have a value of 4 — the .25 part would be ignored.

Operators work together

You don't have to restrict your use of math operators to one per line. You can use them just as you would if you were figuring something out on paper:

```
short  grandTotal;
short  score1;
short  score2;
short  penalty;
score1 = 75;
score2 = 90;
penalty = 10;
grandTotal = score1 + score2 - penalty;
```

If your program is to include tricky, complicated operations that involve several of the operators, you need to know about *operator precedence.* That is, you need to know which operators the compiler looks at first as it performs a calculation. For example, does 5 + 2 * 3 equal 21 (5 plus 2 is 7, times 3 is 21), or 11 (2 times 3 is 6, plus 5 is 11)? Perhaps surprisingly, the answer is 11. Most intermediate-level programming books discuss operator precedence in detail. But I won't go in to it here.

All right, you caught me. What the heck do score1 and score2 refer to? And what kind of penalty does the penalty variable represent? I said it's a good idea to use descriptive words when choosing variable names. If the above code is for a game I'm programming, perhaps I should follow my own advice and select better variable names. Variable names such as scoreGame1, scoreGame2, and tooMuchTimePenalty would probably be better choices.

Operators work with floats, too

Though I haven't shown it, each of the four math operators works with floating-point numbers as well as with whole numbers. Here are a few examples:

```
float  hallLength;    /* entrance hallway length, in feet*/
float  hallWidth;     /* entrance hallway width, in feet */
float  hallArea;      /* entrance hallway floor area      */
hallLength = 6.2;
hallWidth  = 4.0;
hallArea = hallLength * hallWidth;  /* area is 24.8 */
```

Repeating Yourself by Looping

One of the greatest powers of a computer is its capability to do repetitious work, and do it at incredible speed. Computer programs perform this feat by running the same lines of code repeatedly — that is, by *looping* through the lines.

The need to loop

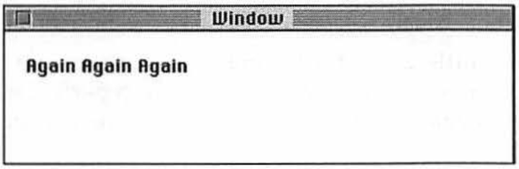
You know that `DrawString` is the command you use to draw some text into a window. Say you want to write the word *Again* three times in a row in a window. Don't ask me *why* you want to do this — just humor me. To write the word *Again* three times in a row, you could use this code:

```
DrawString("\pAgain Again Again ");
```

A second way to achieve the same result would be to use the `DrawString` command three times, like this:

```
DrawString("\pAgain ");
DrawString("\pAgain ");
DrawString("\pAgain ");
```

Either of the above methods results in text that looks something like this:



Again Again Again

That's all well and good. But what if you wanted to write the word *Again* ten times, or a hundred times? You could fill pages of source code with the `DrawString` command. A better method would be to have a line of code that tells the program to repeat the line or lines that follow, something like this:

```
Repeat the code between the following braces 10 times
{
   DrawString("\pAgain ");
}
```

You probably have figured out that the above example isn't real C source code. But if you replaced the first line (the one written out in English) with some C code that did what those words say, you'd be able to write *Again* as many times as you wanted without writing page after page of source code. You'd also have the technique of looping all figured out.

The while loop

Looping is an important part of any programming language, and most languages allow you to easily add looping capability to your source code. The powerful C language is no exception. To create a loop in C, you use a `while` statement.

Words that are part of C, like `short`, `float`, and `while`, are called *keywords*. The keywords make up the language itself. Because each keyword has a specific use and purpose, you can't use these words for other purposes. For example, you can't create a variable named `while`. If you try to, the source code won't compile — you just get an error message. The Metrowerks CodeWarrior documentation includes a complete list of the C language keywords to help you avoid such errors.

In noncomputer terms, here's what a `while` loop does:

```
while something is true...
{
...run the code that is between the braces
}
```

The word *something* is a little vague for a computer, so you won't be surprised to hear that instead of *something*, the `while` loop performs a *test*. Grades for this test don't range from A to F. Instead, the test is scored by either a pass or fail.

In C, a pair of braces defines the start and end of a *block* of code. A block of code is also referred to as a *compound statement*. The braces tell the compiler that the code nested between the braces all belongs together. In the case of a `while` loop, the code between the braces all belongs to the `while` loop.

In C, if a test passes, you say that the test is *true*. If the test fails, you say it is *false*. Take another look at — again, in noncomputer terms — how the `while` loop is shaping up:

```
while test is true...
{
...run the code that is between the braces
}
```

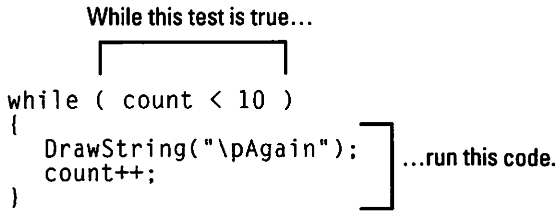
Now for the main event: Take a gander at an actual `while` loop. Before I write the loop, I need to declare a variable that will be used in the loop test. In the following example, the variable `count` serves this purpose. I then assign this variable a value. Why do that? So the loop has predictable results. If I don't know the value of the variable used in the `while` test, I won't know how many times the `while` loop will execute. In this next example, I assign variable `count` a value of 0. The explanation that follows makes it clear why I choose 0.

```
short count;
count = 0;
```

Now for the loop. Just take a look; I explain what's going on afterward:

```
while ( count < 10 )
{
DrawString("\pAgain ");
count++;
}
```

You're looking for the test, right? The test in the loop lies between the parentheses after `while` — you always find the test after the `while` statement. If the test is true, the code following the test runs:



```
while ( count < 10 )
{
   DrawString("\pAgain");
   count++;
}
```

- The less than operator (<) tests true if the left side is less than the right side.
- The greater than operator (>) tests true if the left side is greater than the right side.

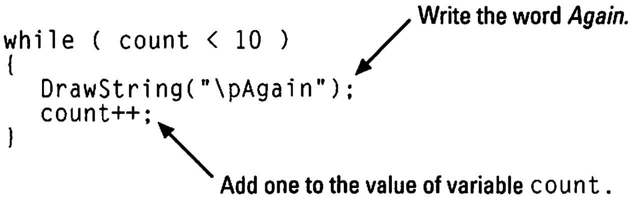
That figure helps, but it may not be descriptive enough. Take a closer look at what's going on in the test. You should recognize the < symbol. It, along with the > symbol, are called *comparative operators.* Like the basic math operators, comparative operators may look familiar to you. You discovered them early in school, though they probably weren't referred to as *comparative operators* at that time. The comparative operators in the `while` loop perform the following test:

The while statement in the example is read, *while* count *is less than 10.* While count is less than 10, do what? The answer is: Run the code that follows the test. Is the value of variable count less than 10? Yes, it is; I assigned it a value of 0, remember?

```
short  count;
count = 0;
```

Because count has a value of 0, the test passes. That is, it *evaluates* to true. That means the program runs the code within the braces. The word *Again* is written to the screen, and the variable count will *increment* by one.

Slow down, you say! What's this about *incrementing* a variable, and what's with the ++ symbol? You know that you can assign a variable a value:



```
while ( count < 10 )
{
   DrawString("\pAgain");
   count++;
}
```

Write the word *Again.*

Add one to the value of variable count.

```
count = 0;
```

But did I mention that later in your source code, you can assign that very same variable a *different* value? Just think about the meaning of the word *variable.* As is my habit, I quote Mr. Webster: "apt or likely to change or vary; changeable, fluctuating." Didn't I tell you that programming terminology actually makes sense sometimes?

Yes, a variable can take on different values during the running of a program. That's what's happening in my while loop — the variable count is being assigned a new value. To assign a new value to a variable, I use the *increment operator,* which is the two plus signs in a row. It's fine to breathe a sigh of relief; that's the last operator you need to know about to work with this book. Of course, this is a pretty big book, and so there *may* be one or two operators I haven't thought of yet. If there are, I explain them wherever they show up. Regardless, you can take comfort in the idea that you don't need to remember a whole lot of operators.

Now for a short review. Before the while loop, variable count has a value of 0. When the program compares count to the number 10, it is indeed less than 10. That means that the test passes and the code between the braces runs. When that code runs, the word *Again* is written to a window. Additionally, the variable count takes on a new value; its old value of 0 incremented by one. After one test, count has a value of 1.

So where's the loop? After the code between the braces runs, the program makes a U-turn and heads back up to the while statement. Code normally doesn't do that. When a loop *isn't* present, code runs one line after another:

```
short  bookstores;
long   books;
float  price;

price = 19.95;
bookstores = 295;
books - 40521;
```

With a loop, the program keeps ending up back at the while statement:

```
short  count;

count = 0;

while ( count < 10 )
{
   DrawString("\pAgain");
   count++;
}
```

Back at the while statement for the second time, the same test is performed again. This time, however, variable count has a value of 1, not 0. count is incremented in the loop *body*. The body is the code between the braces. Is count still less than 10? Yes. So the program runs through the body of the loop again. The very same two lines of code run. The word *Again* is written, and the variable count is incremented again — this time from 1 to 2. Then it's back up to the while statement for still another test.

When does a loop end? When the test fails. After the tenth running of the loop body, variable count has a value of 10. When the program loops back up to the while statement, the test fails. Variable count, with a value of 10, is compared to the number 10. Is 10 less than 10? No. The test evaluates to false, and the body of the loop is skipped.

What happens if you forget to increment the variable that serves as the loop counter? For example, what would the following code do?

```
short count;
count = 0;
while ( count < 10 )
{
   DrawString("\pAgain ");
}
```

This loop is called an *infinite* loop. It runs forever, or at least until you shut the computer off. That's because the variable `count` is first given a value of 0, and then it is never assigned a new value. The test, `count < 10`, is always true, and the loop body always runs. If you compile and then run one of your programs, and it just seems to sit there waiting, check your source code for infinite loops.

Loops are one way of changing the *flow* of a program as it runs. Without a loop, the program runs one line after another. With a loop, that even flow is broken up. Some lines of code — the ones in the loop body — run more than one time. The example in this chapter — repeating code to draw a line of text several times — was helpful but not particularly practical. Here are a few examples of when you may want to include a loop in a program. You can use a loop to:

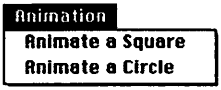
- Flash words or a picture on and off, as in the display of a blinking red light or stop sign to get the user's attention.
- Create animated effects, such as a shape or picture moving across a window (you see an example of this use in Chapters 18 and 19).
- Perform certain mathematical operations, such as cubing a number (multiplying a number to itself twice, as in 4 times 4 times 4).

Changing Directions by Branching

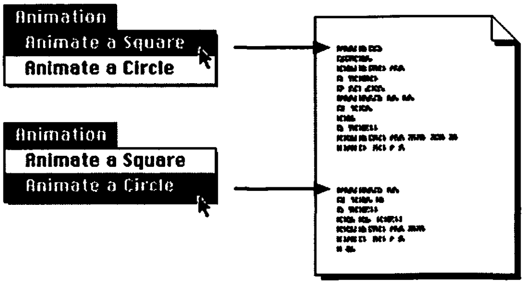
Computers are thought of as decision-making machines. Your Mac can't actually think for itself, of course. But you can write source code that makes it seem like a program running on the Mac is actually making a decision. In order to accomplish this amazing feat of magic, you need to know what *branching* is and how it works.

The need to branch

Imagine you've written a program that has a single menu in the menu bar. The menu has just two items in it:



If the user chooses the first menu item, the program draws a square, and the square races across the window from left to right. If the user chooses the second menu item, the program draws a circle, which then moves across the window. What draws the square and moves it? Your source code. What draws the circle and moves it? Again, your source code — but *different* source code:



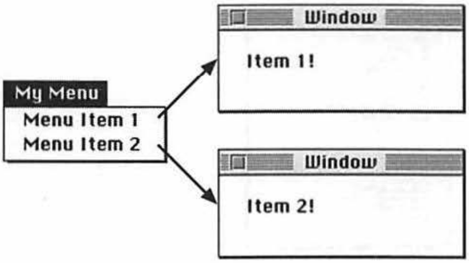
When the user makes a menu selection, how does the program know which source code to run? It determines that from the menu item that was selected. Somehow the program has a way to follow the instructions given from the menu choice and to run certain parts of code based on that choice. The program uses some technique to *branch* down different source code paths.

By the way, by the time you finish this book, you'll be able to write a program that has a menu similar to the one shown above. Not only that, your program will be able to move an object across the screen. This should be solid proof that programming the Mac is not such a hard thing to do after all.

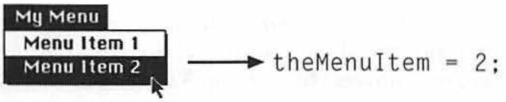
The switch branch

Handling a menu selection is a very practical application of a branching statement, and so I'll continue to use a menu selection to illustrate how branching works. Rather than talk about animation as I did in "The need to branch," I want to start very simply. Assume that I have a menu in my program with these two menu items in it:

When the user chooses the first item, the program writes the words *Item 1!* to a window. When the second item is selected, the program writes the words *Item 2!* to the same window:



From the user's perspective, that's all that happens with this program. From the programmer's viewpoint, a lot more action takes place. If the user chooses the second menu item, the program assigns a variable the value of 2. In the source code for this program, I called the variable `theMenuItem`:



```
theMenuItem = 2;
```

If the user chooses Menu Item 1, then the program assigns `theMenuItem` a value of 1. If you ever wondered why you need to be able to assign different values to a variable, this is a good example of why it's important. In Chapter 17, I discuss menus in detail, and there I demonstrate how a program assigns different values to the same variable. For now, it's just important that you see that it does.

Next, the code takes care of whatever needs to be done in response to the selection of the second menu item. Not by accident, I picked an example where not much has to be done after the menu selection. The program simply writes *Item 2!* to a window. (Please take a blind leap of faith with me and assume that this window was created earlier in the program.)

Here's where the decision-making comes in. You know that the program writes to the window regardless of which menu item the user picks. But which words should it write? Somehow the program must decide whether to write *Item 1!* or *Item 2!* What should the program base the decision on? Answer: The value of variable `theMenuItem`, which holds the number of the menu item. In plain English, the source code needs to do something like this:

```
what is the value of theMenuItem?
{
   in case its value is 1:
      DrawString("\pItem 1!");
   in case its value is 2:
      DrawString("\pItem 2!");
}
```

Once again, you probably realize that the preceding words aren't actual code. Here's the real source code for making a decision:

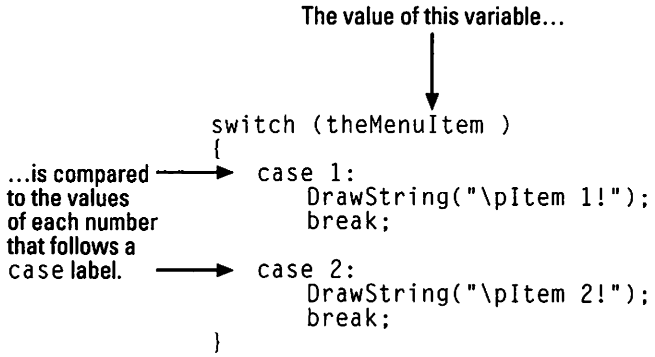
```
switch ( theMenuItem )
{
   case 1:
      DrawString("\pItem 1!");
      break;
   case 2:
      DrawString("\pItem 2!");
      break;
}
```

Hey, what's that `switch` stuff doing there? The `switch` statement is the C language way of selecting one group of source code to run from a collection of two or more groupings. The program doesn't always run the same group of code; it switches, depending on the circumstances at the time the program meets the `switch` statement. To write a `switch` branch, first write the C keyword `switch` followed by a variable name between parentheses:

```
switch ( theMenuItem )
```

The variable should be an `int`, a `long`, or a `short`. Following the `switch` line comes a pair of braces. In between the braces are two or more `case` *labels.* In my example, I use two labels, `case 1:` and `case 2:`. Each label is followed by one or more lines of code, and each label ends with a `break` statement. Like `switch` and `case`, `break` is a C keyword, a word defined to be a part of the C language.

A switch statement works by comparing the value of the variable between the parentheses on the switch line to each of the values associated with the case labels:



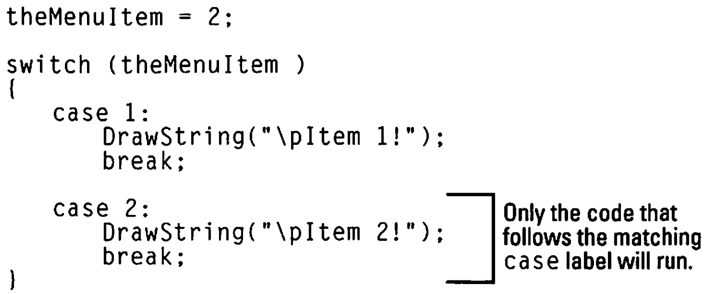
```
switch (theMenuItem )
{
   case 1:
      DrawString("\pItem 1!");
      break;
   case 2:
      DrawString("\pItem 2!");
      break;
}
```

The value of this variable...
...is compared to the values of each number that follows a case label.

Say the user of my program chooses the second menu item. My code then encounters an assignment statement like this:

```
theMenuItem = 2;
```

Then comes the switch statement. The value of theMenuItem, 2, is compared to the values listed in each case label. When a label with the same value as theMenuItem is found, the code under that label runs:



```
theMenuItem = 2;

switch (theMenuItem )
{
   case 1:
      DrawString("\pItem 1!");
      break;

   case 2:
      DrawString("\pItem 2!");
      break;
}
```

Only the code that follows the matching case label will run.

It's important to realize that not all of the code between the braces runs during a pass through the switch. Only the code that follows *one* label runs. After the program starts running some of the code, what signals the program that it's time to leave the switch? That's where the break statement comes in. No matter what case label code is running, when the program reaches a break statement, the switch ends. That is, the program jumps out from the braces and moves on to the line of code that follows the switch. So it's important that *every* group of code that follows a case label ends with a break:

```
switch (theMenuItem )
{
   case 1:
      DrawString("\pItem 1!");
      break;

   case 2:
      DrawString("\pItem 2!");
      break;
}
```

When the program reaches a break statement, the rest of the switch code is skipped.

Don't forget the break statements! If you do, the code under more than one case label can run. If I didn't include break statements in my menu example, and theMenuItem had a value of 1, both *Item 1!* and *Item 2!* would get drawn to the window. Without the break after the first DrawString, there would be nothing to tell the switch statement that it is time to bail out of the switch.

The if branch

The switch branch is very useful when you want your program to choose from several possibilities. But many situations arise when you only want your program either to run a section of code or not run it. Although the switch branch can be used for such occasions, a different type of branch is more practical — the if branch. If you've mastered the switch, the if branch may look very simple. Here's an example:

```
short vacationDays;

if ( vacationDays > 0 )
{
   MoveTo( 20, 30 );
   DrawString( "\pYou've got vacation coming!" );
}
```

An if branch uses a test condition to determine whether the code under the if should run or not run. In the preceding example, the if examines the value of the variable vacationDays to see whether it is greater than 0. If it is, the two lines of code between the braces run. If vacationDays isn't greater than 0, the program skips the code between the braces lines, and no message is written.

The if branch is used to handle situations that have two possible outcomes. In the preceding example, vacationDays is either greater than 0 or it's not. For one of the outcomes (vacationDays greater than 0), the example draws a message. What if I wanted to draw a message regardless of the outcome? If you want the if branch to respond to both possible outcomes, you can use an else in conjunction with the if. I'll expand upon the vacation example to provide a demonstration of what's referred to as an if-else branch:

```
short vacationDays;

if ( vacationDays > 0 )
{
   MoveTo( 20, 30 );
   DrawString( "\pYou've got vacation coming!" );
}
else
{
   MoveTo( 20, 30 );
   DrawString( "Sorry, no vacation days left." );
}
```

Like the code that makes up the body of the if section, the code that is to be used in the else section is defined by opening and closing braces. Unlike the if part of the if-else, following the else here is no pair of parentheses with a test between. The else doesn't need any kind of test — the code under the else runs every time the test following the if fails.

That's All There Is to C?

No, there's plenty more. If that was it, there wouldn't be much need for software engineers like myself. I know with that line I'm setting myself up for some rude comments from you, but show a little mercy.

Although this chapter doesn't cover all the C there is, it does cover the basics. The remaining bits and pieces that you need in order to write a Macintosh program are doled out throughout the remainder of the book. And if you have any quick questions about C, you can always refer to the C language reference in Appendix A. There I list several important elements of the C language along with examples of each, which may give you some ideas on how to construct or modify your own programs.

Chapter 15

To Build a Program, You Need a Toolbox

In This Chapter

- Becoming good friends with the Toolbox
- Finding the Toolbox
- Sampling the Toolbox

If every Mac programmer had to write every program completely from scratch, they'd all be in a world of hurt. It would just be too big a job. Long ago, Apple realized that programmers need some help, and so they kindly did something about it. Many of the strange and wonderful things that are done in Mac programs can now be achieved by Macintosh programmers with just a few lines of code. They, you, and I all have the Macintosh Toolbox to thank for that.

Why Have a Toolbox?

In Chapter 5, I give you a brief introduction to the Macintosh Toolbox. There I say that the Toolbox is a collection of miniprograms that you can incorporate into your own programs. You've surely heard the expression about reinventing the wheel. Now, I'm not usually really big on clichés, but this one does work perfectly when describing the Macintosh Toolbox.

Many of the things you want to accomplish as a Macintosh programmer are the very same things every other programmer wants to accomplish. All Mac programmers want to create and move windows, draw to them, and display menus, to name just a few common tasks. If you, I, and every other programmer want to do these things, we all must write the code to do it. Sounds like a lot of repetition, doesn't it? Apple thought so, too.

Apple programmers anticipated everyone trying to figure the same things out and writing the same code once they did. So Apple went ahead and wrote much of the code for you.

You may not have been aware of this thing called the Toolbox when you bought your Mac, but for some people it's a big selling point. Why is Apple so generous with the free programs in the Toolbox? Programmers like to program, of course, but they don't like to do a whole lot of the work involved in programming the mundane tasks — things such as making a menu drop down when the user clicks the menu bar, or closing a window when the user clicks the mouse in the window's close box. Programmers just want to type in a single command for these commonplace things to happen. A Mac programmer can then devote his time to matters of more importance, such as making his programs reverberate the Mac's speaker to scare the dog.

Miniprograms by Any Other Name

How about a dose of some correct terminology? Each of these things I've been calling a miniprogram is actually called a *function.* A function is a collection of instructions that generally serve a single purpose. I say *generally* because it is possible to write a function that does all sorts of not-necessarily-related things, although that's not the preferred way of doing it.

`GetNewWindow`, `MoveTo`, `DrawString`, and any other part of the Toolbox are each functions. Sometimes programmers refer to functions as *routines.* Yes, the terms *routines, functions,* and *miniprograms* mean one and the same and are interchangeable. I use the word *function* in the remainder of this book.

The Toolbox Gives and Receives

Using the Toolbox functions that were written by Apple is an essential part of programming the Mac. Toolbox functions are very useful and very powerful. Those Apple engineers are a pretty clever bunch, so I bet you think there's no way you could contribute to their efforts — right? Wrong! A Toolbox function carries out a task almost as if it were a small program; hence my reasoning for calling them miniprograms in other parts of the book. But just calling a Toolbox function usually isn't enough to get the function to do its work. That's where you get to contribute: You must supply the function with some additional information to help it do its work.

Function parameters

To use a Toolbox function in your program's code, you write its name and follow the name with a pair of parentheses. Between the parentheses lie the function *parameters.* The parameters provide the Toolbox with information it needs in order to properly run the Toolbox function. For example, the

Toolbox function DrawString needs to know what words to write to a window. You give it that information in the form of a single parameter. Without it, the Toolbox wouldn't know what to write. Here's a call to DrawString:

```
DrawString("\pTesting 1 2 3");
```

Even though several separate words appear between the parentheses, they are considered a single parameter. The DrawString function always requires just a single parameter:

One parameter

```
DrawString("\pTesting 1 2 3");
```

The word is pronounced *pah-ram-ah-ter.* I thought I'd mention that for you British readers, who may be tempted to say *para-meter,* as in centimeter!

The dictionary says that in math, a parameter is a quantity whose value varies with the circumstances of its application. That's about the same definition that's used in computer programming. Function parameters vary, depending on the circumstances. At the start of a program, a parameter to DrawString may be:

```
DrawString("\pWelcome!");
```

However, the parameter to DrawString may be:

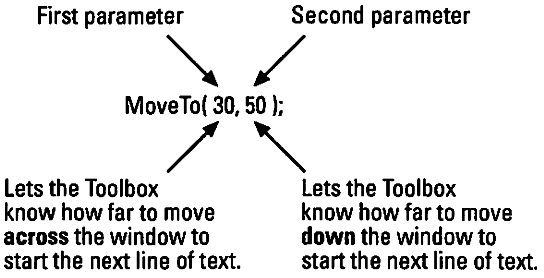
```
DrawString("\pGoodbye!");
```

at the end of a program.

Some Toolbox functions need more than one parameter. To let the Toolbox know where one parameter ends and the next begins, you separate parameters with a comma. Take the MoveTo function as an example. This Toolbox function specifies where in a window the next line of text should be drawn. I describe MoveTo in the next chapter in great detail. For now, here's a glimpse of the MoveTo function that you may recognize as a snippet from Chapter 5's ExampleOne:

```
MoveTo( 30, 50 );
DrawString( "\pHello, World!" );
```

The next figure shows the two parameters of `MoveTo`. It also gives you a hint at what they tell the Toolbox to do.


First parameter
Second parameter
MoveTo(30, 50);
Lets the Toolbox know how far to move **across** the window to start the next line of text.
Lets the Toolbox know how far to move **down** the window to start the next line of text.

Some Toolbox functions don't require any parameters. Functions that don't need additional information can perform only one, unvarying task. Here are a couple of lines of code I lifted from the ExampleOne program of Chapter 5:

```
InitFonts();
InitWindows();
```

The preceding two calls to Toolbox functions don't have parameters, but they still include the parentheses. In the C language, calls to functions always end with a pair of parentheses regardless of the number of parameters. Because these two functions don't need parameters, you know that they always do the same thing. `InitFonts` always initializes the fonts so that your program can make proper use of fonts. `InitWindows` always initializes things in the Toolbox so that it can work properly with windows.

By the way, another name for parameter is *argument.* You may find that some programming books use one, the other, or both terms. In this book, I use the term *parameter* — it's a little less abrasive-sounding, don't you think?

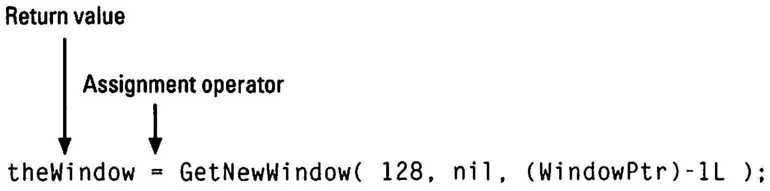
Functions return values

Because you're kind enough to help out the Toolbox by giving it parameters, the Toolbox occasionally reciprocates by giving something back to you. Well, back to your program, actually. Some Toolbox functions do this by providing your program with a *return value.* A value returned by the Toolbox can be of any data type — it all depends on the Toolbox call being used.

If a function *doesn't* have an equal sign (the assignment operator) in a call to it, it *doesn't* have a return value. For example, a call to MoveTo doesn't return a value:

```
MoveTo( 30, 50 );
```

If a call to a function *does* include the assignment operator, then it *does* return a value:



```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

To see how the preceding line of code works, take a look at the assignment operator from a different angle. To give a value to a variable, you first declare it, and then you assign it a value:

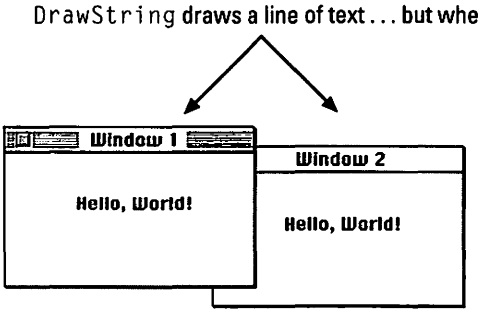
```
short  books;
books = 300;
```

Those are the same steps you take when writing the code for a function that has a return value:

```
WindowPtr  theWindow;
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

The first line is the declaration of a variable called theWindow. Its data type is WindowPtr. The second line is the assignment statement. The variable theWindow is assigned a value.

Data types such as the int, short, and the float are always numbers. There are plenty of other data types, and many of them *don't* represent numbers. The WindowPtr is one such type. A variable that is a WindowPtr represents a window. You create a different variable of the WindowPtr type for each window you create. This way, when you want to do something to a window, such as write text in it, you have a means of specifying which window to use:



In Chapter 16, I go into all the sordid details of how your program chooses which window to draw to by using a `WindowPtr`. For now, you should be aware that a `WindowPtr` variable gets its value when a window is created.

When you decide to give a `short` variable a value, you simply assign it one:

```
books = 300;
```

It's not quite that easy to do when the value of the variable isn't a number. That's when the Toolbox takes over and does the assignment for you in the form of a return value. Now the call to the Toolbox function `GetNewWindow` should be making a little more sense to you:

```
WindowPtr  theWindow;
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

`theWindow` is a variable. It gets its value when the Toolbox function `GetNewWindow` returns it right after the function creates the window. Now, what about the parameters of `GetNewWindow`? Sorry, this section deals with return values, not parameters. But if you keep reading, you'll find out all about the parameters for `GetNewWindow`.

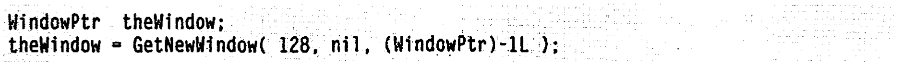
Sampling the Toolbox

There are several thousand functions in the Toolbox, so I'm sure you'll forgive me if I don't cover them all. I do want to, however, cover one call in detail here, which is `GetNewWindow`, an important Toolbox function used in just about every Mac program.

Some Toolbox functions have no parameters, others have one, and still others have more than one. To make matters worse, functions require all different *types* of parameters. `DrawString` requires one or more words as its one parameter. `MoveTo` requires two numbers for its two parameters. How on earth can one person memorize which functions require which parameters? It's another one of those good news/bad news situations. The bad news is, there is no possible way you can memorize all the parameter types. The good news is you aren't expected to!

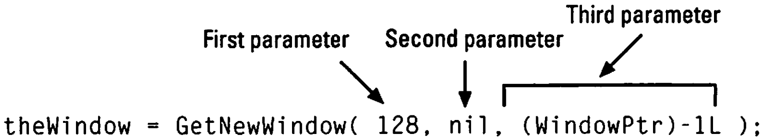
Macintosh programming books *tell* you what parameters to use. For example, in the next chapter, I list several commonly used Toolbox functions along with a brief description of what each function is used for. I also list the parameters for each. I provide that same information for several other Toolbox functions in Appendix B. So, how did *I* memorize all of this information so that I could pass it on to you? Feel free to tell people that I'm incredibly brilliant and was blessed with a photographic memory. But the truth of the matter is that I just looked them up in reference material supplied by Apple.

But hold on — I need to get back to the `GetNewWindow` function. `GetNewWindow` creates a new window for your program to use. The window has the characteristics, such as size and location on the screen, that you specify in a 'WIND' resource. A call to `GetNewWindow` returns a `WindowPtr` to your program. That is, the function assigns a `WindowPtr` value to the variable on the left side of the assignment operator. Here's a declaration of a `WindowPtr` variable and a typical call to `GetNewWindow`:



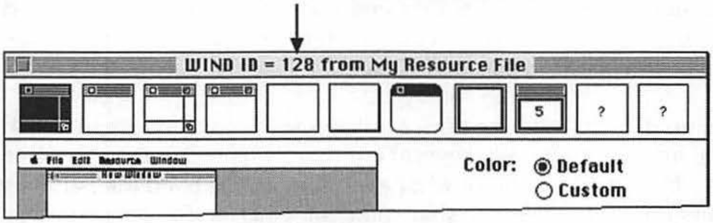
```
WindowPtr  theWindow;
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

You've had some pretty heavy exposure to `GetNewWindow` in this book. But I never did describe the mumbo-jumbo that's tucked in between the parentheses. You may have guessed that the stuff between the parentheses are parameters. `GetNewWindow` requires three of them:



```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

The first parameter to `GetNewWindow` is the ID of the 'WIND' resource for the window you want to create. In Chapter 8, I show you how to create a 'WIND' resource using ResEdit. When you make a resource in ResEdit, the program gives it an ID. ResEdit usually gives the first resource of each type an ID of 128 — that's why you see the number 128 scattered about Macintosh source code. If you want to double-check on a resource ID, run ResEdit and open the resource file. I did that to verify that my 'WIND' resource did indeed have an ID of 128:



WIND ID = 128 from My Resource File

Color: ● Default ○ Custom

The ID of a resource can be found using ResEdit.

The second parameter to GetNewWindow is used to reserve memory for the window. Reserving memory is a tricky business, so it's good that you have the option of letting the Mac do it for you. If this second parameter has a value of nil, the Toolbox figures out where the new window should be stored. You can think of nil as your means of getting out of the business of supplying a particular parameter and allowing the Mac to handle the task. Obviously, not all parameters give you that option — if that were the case, we'd write nil just about everywhere! The nil value is used mostly for parameters that have something to do with memory. You don't have to determine when it's okay to use nil — I'll let you know.

The third and final parameter to GetNewWindow specifies whether the new window should open in front of any other open windows or behind them. I'm not exactly sure why you would ever want a new window to be hidden by other windows on the screen, but I guess it's nice to know you have the option to do so. If you want your window to open up in front, as I always do, always use (WindowPtr)-1L for this parameter.

No, I'm *not* going to explain what the heck (WindowPtr)-1L means. Trust me. You don't want to know! Just make sure to type it correctly. Include the parentheses, a minus sign, the number one, and an uppercase letter *L*.

Let me say two things: Reserving memory for a window is a very advanced programming technique. It's very unlikely that you'll ever do it. And in any program I've ever worked with, a new window always opens up on top of any existing windows. Yours should, too.

What do the preceding two points mean to you? You can essentially forget about what the second and third parameters to GetNewWindow are all about. If you always use nil for the second parameter and (WindowPtr)-1L for the third, you're safe. That means you only have to remember what the first parameter stands for — the ID of the 'WIND' resource of the window you want to open.

In this chapter, I concentrate on explaining the GetNewWindow function. That makes sense, because every Mac program displays at least one window. But the Toolbox is capable of much more than making windows — as you see in Chapter 16.

Chapter 16

Drawing with C: Why Have a Mac If You Can't Draw?

In This Chapter

- Finding QuickDraw in the Toolbox
- Specifying the window location to draw to
- Drawing lines
- Drawing rectangles
- Specifying which window to draw to

The Macintosh is best known for its GUI (graphical user interface). Note the word *graphical* in that phrase. Graphics may not be the most serious side of programming, but then, who wants to always be so serious? If you can't play around on the Mac by drawing a few lines and shapes now and then, why have one? Besides, you can have fun drawing with the Mac and still learn some important programming and C language concepts. So stop feeling guilty.

Quick on the Draw

The Macintosh Toolbox is divided into separate areas. One area holds functions that work with windows — that area is called the Window Manager. Another area holds functions that work with menus — that's the Menu Manager. And still another area contains functions used for drawing. Stop right there! That's the one you're interested in — at least in this chapter. As you've just seen, these different parts of the Toolbox have names, and the part that holds the drawing functions is called *QuickDraw*.

Remember, the Toolbox is nothing more than a huge collection of functions. So each part of the Toolbox is simply a set of functions. When you refer to QuickDraw, you're referring to one or more of the functions that are used to draw in a window.

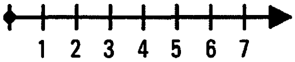
The Coordinate System

Before I get down to the business of drawing, there's one issue I have to clear up. I mention it several times in different parts of the book: When you tell the Mac to draw something to a window, how does it know *where* to draw that something? You know that you use the Toolbox function MoveTo before you use DrawString, right?

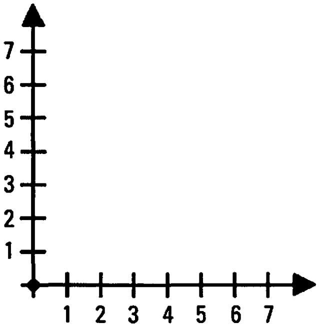
```
MoveTo( 30, 50 );
DrawString( "\pHello, World!" );
```

But that only hints at what's happening. What do the numbers that lie in the parentheses of the call to MoveTo mean? For the answer to that question, you must return to your grade school days . . .

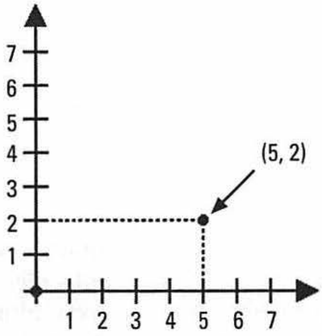
Remember the number line? In school, you used it to count and to prove to yourself that numbers lined up in order. I don't know about your grade school, but in mine it looked something like this:



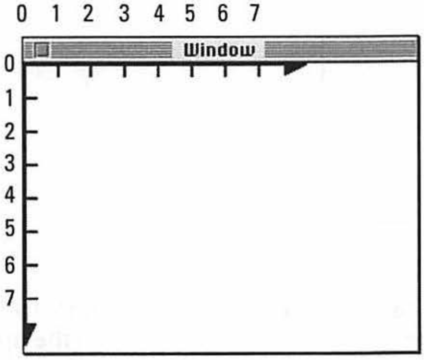
You felt pretty good after you mastered the number line. That is, until a few years later when you were introduced to the coordinate grid. Math teachers loved the ol' number line so much, they stuck two of them together to form something like this:



Math teachers had to next think up something to *do* with all their coordinate grids. So they had you plot points, remember? In the figure below, I plot the point (5, 2). That notation means that I move five places along the horizontal line and two places up the vertical line.

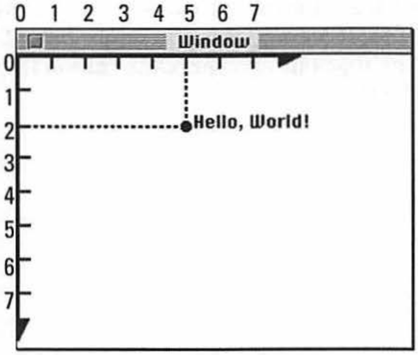


By now you're surely wondering whether I introduced this subject just for the sheer fun of it, or for the sake of nostalgia. Neither. It applies directly to the topic at hand, which is how the Mac draws things in a window. Don't believe me? Let me make the connection and wrap things up. Imagine flipping the coordinate grid and placing it in a window. It would look like this figure:

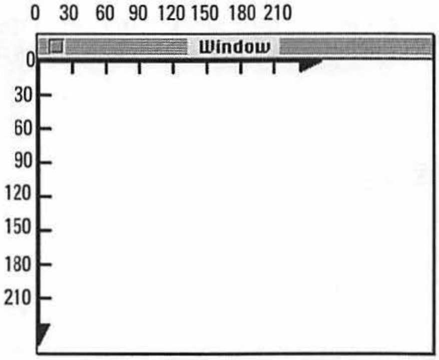


Now, if you want to move to a particular point in a window, you use `MoveTo` just like you did on the coordinate grid in school. Here I've moved to the point (5, 2):

After a call to `MoveTo`, I call `DrawString`. Where does the text start? Right where I moved to:



The preceding figure is very close to the truth, but not quite. You see, I used a number line and coordinate grid with nice big numbers and a nice wide spacing between numbers. That was for your benefit so that everything would look nice and clear. On a Mac, the coordinate grid is much, much smaller. The distance between points on the grid of the Mac is so small that I can't draw it accurately in a figure. Instead, I only label every 30 points:



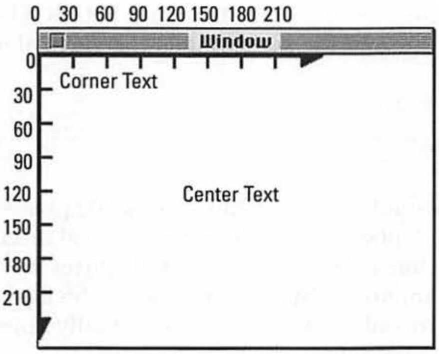
The small grid size is good because it gives you more freedom to pick and choose just where you draw things. If I want to write text in the upper-left corner of a window, I write this code:

```
MoveTo( 15, 30 );
DrawString("\pCorner Text");
```

If I want text to start near the center of the window, I use this code:

```
MoveTo( 150, 120 );
DrawString("\pCenter Text");
```

Take a look at the result of both of these `DrawString` calls:



The preceding figure is now accurate except for one obvious point. The coordinate grid you see superimposed on the window doesn't appear on a real window. You have to do a little guesswork to determine just where to place the text. It should help, however, if you realize there are 72 grid marks per inch of window.

The screen is composed of thousands of individual dots that can be turned on and off, and each one of these dots is called a *pixel*. Just now when I told you that there are 72 grid marks per inch of window, I was referring to pixels. So pixels are very small. If you touch your nose to your monitor, you may be able to see the individual pixels.

Let's Draw!

I didn't make you do your number line schoolwork as a punishment. The idea of a coordinate grid, or *coordinate system* as some call it, is very important when it comes to drawing on the Mac. That's because the QuickDraw functions rely on it.

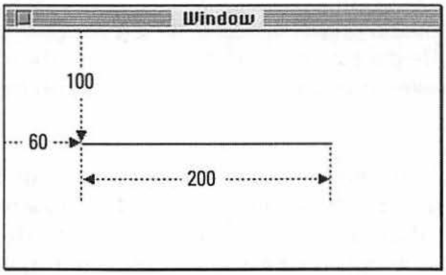
Drawing a line

I always like to start simple, so let me begin the discussion of shape drawing by using the simplest shape I can think of — a line. Is a line really a shape? I'm not sure, but a few of them together make up a shape, and that's good enough for me.

To draw a line, you move to the window location where the line should start, and then you call the Toolbox function `Line`. You have the freedom to draw a line of any length and in any direction. To let the Toolbox know what your line is to look like, you give the `Line` function two parameters. The first tells the horizontal length of the line and the second tells the vertical length of the line. Here's an example:

```
MoveTo( 60, 100 );
Line( 200, 0 );
```

The preceding call to `Line` would result in a line that is 200 pixels in length in the horizontal direction, and 0 pixels in length in the vertical direction. The next figure shows where the line is eventually drawn. I clutter up the window with a few extra arrows and numbers, but they're only in the figure to help point out what's going on. The only thing that would actually appear in the window is the one solid, horizontal line.

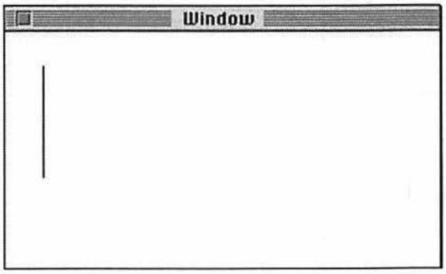


To draw a horizontal line, as I do above, set the second parameter of `Line` to 0. That tells QuickDraw to go zero pixels in the vertical direction.

How about a vertical line? To draw a vertical line, set the first parameter of `Line` to 0. That tells QuickDraw to draw zero pixels in the horizontal direction:

```
MoveTo( 30, 30 );
Line( 0, 100 );
```

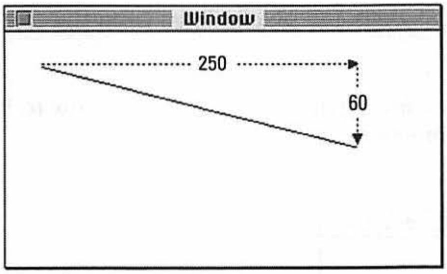
The previous code first uses `MoveTo` to move 30 pixels in from the left and 30 pixels down from the top. Then `Line` draws a line 0 pixels in the horizontal direction and 100 pixels in the vertical direction:



A line doesn't have to be vertical or horizontal. It can be drawn at just about any angle. For example, if you type this code:

```
MoveTo( 30, 30 );
Line( 250, 60 );
```

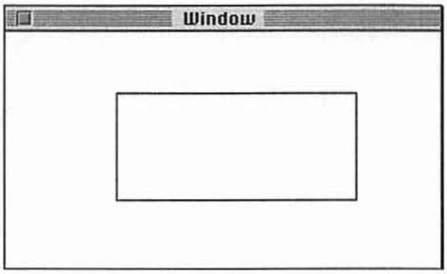
You get a line that looks like this:



Drawing a rectangle

Drawing a rectangle, like drawing a line, requires knowledge of the Mac's coordinate system. It also requires that you be familiar with a data type you haven't seen yet, which is the `Rect`. The `Rect` holds information about a single rectangle. It's an interesting data type in that it holds four different numbers at the same time.

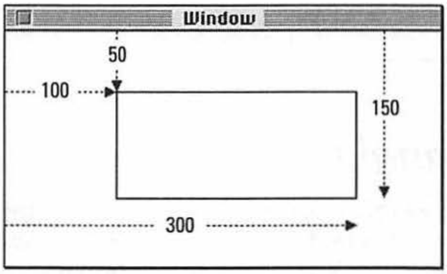
A variable that is of the `Rect` type holds the four *coordinates* of a rectangle. Any rectangle can be defined by using the coordinate system to specify the left, top, right, and bottom of the rectangle. Say you want to draw a rectangle that looks like this:



If you want to define the above rectangle using the numbers of the coordinate system, you could say something like this:

- The left side of the rectangle is 100 pixels in from the left of the window.
- The top side of the rectangle is 50 pixels down from the top of the window (the window's title bar doesn't count).
- The right side of the rectangle is 300 pixels in from the left of the window.
- The bottom of the rectangle is 150 pixels down from the top of the window.

In the following figure, I add a few dashed lines to the window to illustrate where these numbers are coming from:



The Toolbox exists to make your life easier, and `SetRect` is one example of how that's true. The `SetRect` function performs four separate assignments, all in one call. Here's how I would assign the four coordinates of the preceding rectangle to a `Rect` variable named `theRect`:

```
Rect   theRect;

SetRect( &theRect, 100, 50, 300, 150 );
```

Order is important! If you switch numbers around, the result will not be what you wanted. Think of SetRect like this:

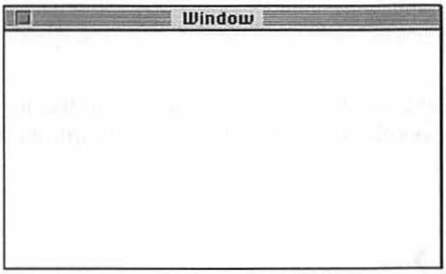
```
SetRect( &theRect, left, top, right, bottom );
```

Here's another way to remember the order: litterbug. The word *litterbug* has the first letter of left, top, right, and bottom in the correct order — **L**I**T**TE**RB**UG.

Notice the & symbol that precedes theRect in the SetRect function call. That symbol is important. How important? This important:

Without including that innocent-looking & symbol in the SetRect, you get an error message when you try to compile the program. SetRect and several other QuickDraw functions require it — so keep an eye open for it as you type in source code.

After calling SetRect, here's what my window looks like:

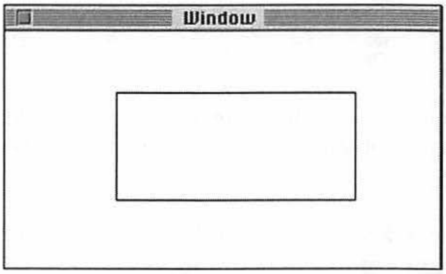


Misprint? Nope. SetRect doesn't draw a thing. But then, it's not supposed to. All SetRect does is assign a Rect variable the coordinates of a rectangle. An assignment statement just gives a variable a value; it doesn't draw anything.

To do the drawing, you use another Toolbox function, which is called FrameRect. It's possible to have more than one Rect variable in your program, so you need to help FrameRect out by telling it which rectangle you want to draw, or frame. Here's how you draw a rectangle, from start to finish:

```
Rect  theRect;

SetRect( &theRect, 100, 50, 300, 150 );
FrameRect( &theRect );
```

And here's the result of my efforts:



Lines and rectangles are just a small sampling of the drawing capabilities of QuickDraw, but they should be enough to give you an idea of how drawing works. Make sure to check out Appendix B for a description of several other QuickDraw drawing functions that you can easily incorporate into your own programs.

Drawing to a Port

In Chapter 15, I promised that I would go into the details of the `WindowPtr` data type and how it's used to select different windows. Hoping I forgot, huh? No such luck.

Why have ports?

Here's the dilemma: Your program opens two windows, and you want your program to write to one of them. You could write code like this:

```
MoveTo( 50, 50 );
DrawString("\pHello, World!");
```

But which window is the text drawn to?

The solution to the problem comes in the form of *ports*. Every window has one port. It's a kind of identifier that makes each window unique and allows you to pick and choose among open windows.

Normal people (people who don't program computers) think of a port, or portal, as a gateway or a place of entry. That definition fits in pretty well with the programmer's use of the word. A window's port allows you to draw in the window and move it around on the screen. Without a window's port, you're locked out. By the way, the word *port* also is the name of a sweet, dark-red wine, but I don't *think* Apple was referring to that when they selected the word.

Before you write or draw to a window, you tell the Mac which port you want to work with. Only one port can be *set* at any given time, which prevents writing or drawing from taking place in two windows at once. Before writing a line of text to a window, you want to write something a little like this:

```
Set the port to the port belonging to Window 1
MoveTo( 50, 50 );
DrawString("\pHello, World!");
```

You may have already figured out that the first of the above three lines isn't valid C code. But it does give you an idea of what you're shooting for. To find out *exactly* how you do it, read on.

WindowPtrs and Ports

In Chapter 15, you see that some Toolbox functions return a value. You also see that `GetNewWindow` is one such function. It assigns a value to the `WindowPtr` variable that appears in the same line of code as `GetNewWindow`. Before the call, variable `theWindow` had no value. After the call, it does. What value isn't important — be content to know that the value is a collection of information about the window that just opened.

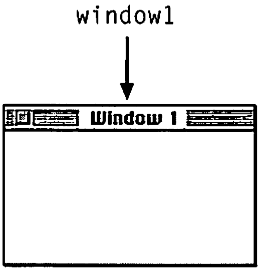
Imagine a program that opens two windows. This program would have two `WindowPtr` variables declared; I cleverly call these windows `window1` and `window2`:

```
WindowPtr  window1;
WindowPtr  window2;
```

First, one window is created and displayed:

```
window1 = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

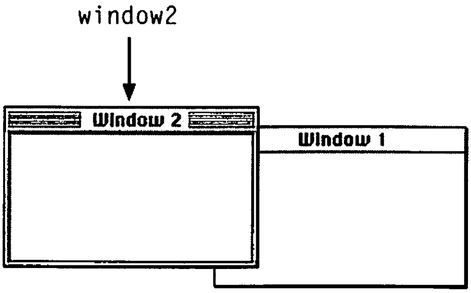
Notice that I used the name of one of the `WindowPtr` variables, `window1`, in the preceding line. After the call is complete, `window1` points to the newly opened window. It serves as a reference to it:



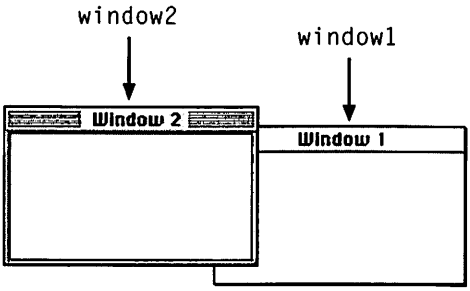
Next, I open and display the second window. Here I use the other `WindowPtr` variable, `window2`:

```
window2 = GetNewWindow( 129, nil, (WindowPtr)-1L );
```

The new window opens up in front of the old window, and the variable `window2` points to the new window:



Now the program has two windows and two separate pointers, each pointer referencing one window:

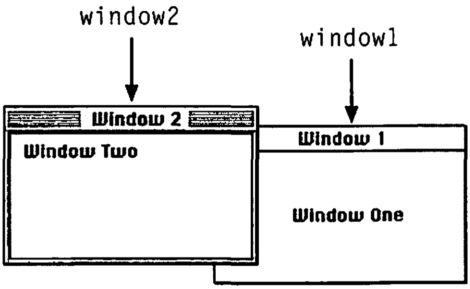


Now the setup is over. It's time to write to one of the windows. Which one? That's up to me because now that I have a WindowPtr for each, I have the means to pick the window I want to use. You tell the Mac which window you want to work with by calling the Toolbox function SetPort. This function is simple to use; just pass it the WindowPtr variable for the window you want to draw to. Here's an example that sets the port to the port of the second window I opened and then draws to it:

```
SetPort( window2 );

MoveTo( 50, 50 );
DrawString("\pHello, World!");
```

Just to reinforce things a little, here's a more comprehensive example. In the following code, I open two windows, just as I did above. Then I write one line of text to one window and a different line of text to the other window. After the code is a figure that shows what the windows should look like.

```
WindowPtr   window1;
WindowPtr   window2;

window1 = GetNewWindow( 128, nil, (WindowPtr)-1L );
window2 = GetNewWindow( 129, nil, (WindowPtr)-1L );

SetPort( window1 );
MoveTo( 50, 50 );
DrawString("\pWindow One");
SetPort( window2 );
MoveTo( 10, 20 );
DrawString("\pWindow Two");
```



What if your program only uses one window? Sorry, you still have to call SetPort before drawing or writing to it. Sure, it may seem ridiculous to tell the Mac to set the port to your window when that's the only window on the screen. But the Macintosh is a computer, and computers never do things without some kind of reason. In this case, there's a pretty darned good one. It turns out that the entire screen of your Macintosh also has a port, just as each window on the screen has one. So if you don't call SetPort when your one window is on the screen, subsequent drawing may miss your window and take place right on the screen! Think about the ExampleOne program from Chapter 5; I use SetPort even though the program opens a single window. Here's that entire short program again, just in case you don't have the memory of an elephant. By the way, this code shows why comments are really important. You don't have to remember the functions and meanings of each line of code from ExampleOne because my comments tell you exactly what's going on:

```
void  main( void )
{
   WindowPtr  theWindow;       /* the window */

   InitGraf( &qd.thePort );    /* standard initializations */
   InitFonts();
   InitWindows();              /* next line opens a window */

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   SetPort( theWindow );              /* set the drawing port */

   MoveTo( 30, 50 );           /* set the drawing position */
   DrawString( "\pHello, World!" );    /* draw some words */

   while ( !Button() )         /* do nothing until */
      ;                        /* button is clicked */
}
```

I think I have already said this 100 times, but the point is worth repeating. Even if your program uses only one window, you should still call SetPort before writing text or drawing shapes in it. If you don't, what was supposed to be drawn may not get drawn, and you may spend a lot of time trying to figure out why.

That concludes Part IV, which means it's on to bigger and better things, including the creation of your first program that includes both a window and a menu.

Part V

The Moment of Truth: Writing a Program!

The 5th Wave
By Rich Tennant
ROCKY XI - ROCKY BALBOA BECOMES A Real Programmer
©RICH TENNANT
GIVE IT UP, ROCK!! THIS VIRUS IS TOO TOUGH TO CRACK! YOU BEEN WORKIN' AT IT SO LONG YOUR EYES HAVE SWOLLEN SHUT!
CUT ME, MICK. CUT ME.

In this part . . .

What do I mean by the moment of truth? Heck, you see a Mac program way back in Chapter 5. True enough. But you haven't seen the source code for a real Mac program. One that has a menu that allows the user to make choices. That's what Mac programs are all about — giving the user the power to decide what to do.

The program that's covered in the chapters of this part has a functional menu and a movable window — two important program features not covered up to this point. It also demonstrates how to effectively use the part of the Toolbox that draws shapes in windows — QuickDraw. And, as if that weren't enough, this part shows you exactly how to turn your source code into an honest-to-gosh application, a program that you can save on your hard drive or copy to a floppy disk.

Finally, this part ends with a few pointers on what to do next. That is, what's in store for you if you care to carry on with your Mac programming endeavors.

Chapter 17

Examining a Simple Mac Program

In This Chapter

- Looking at the source code for MyProgram
- Functions — with and without the Toolbox
- Initializing the Toolbox
- Events — how a program responds to the mouse
- Getting closer to writing a real Macintosh program

In this chapter, I give you a detailed look at the MyProgram program — a program I've described in the last few chapters. If you've read the previous few chapters, you know that MyProgram contains many of the elements that are key to writing a Macintosh program. A good hard look at each line of the program may crystallize your thoughts about those programming elements. But I won't be content to simply rehash things you've already worked on, though. In this chapter, I'll also add a few new lines of code to MyProgram. And of course, I'll describe exactly what this new code does.

It is important to keep in mind that while the MyProgram program demonstrates several of the important concepts of Macintosh programming, it isn't a program to really brag about. It's not exactly the fanciest Mac program in town. In this chapter, I also rectify that situation by adding elements to the source code to get a little bit closer to what a Macintosh program is really all about. You have to read the chapter to find out what these new elements look like!

The MyProgram Program Source Code

Because I discuss the MyProgram program in this chapter, it may be helpful to see it in its entirety, which I show you in the following code. If you've been reading the book sequentially, you've already seen this listing.

After you read this chapter, you may think that MyProgram is a thing of the past, a done deal. Or is it? You see, almost each line of code that makes up MyProgram is used in every Macintosh program you write. So bits and pieces of the infamous MyProgram live on to be seen again. And now, without further ado, MyProgram:

```
void  main( void )
{
   WindowPtr  theWindow;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   SetPort( theWindow );

   MoveTo( 30, 50 );
   DrawString( "\pHello, World!" );

   while ( !Button() )
       ;
}
```

If you look in the *...For Dummies* Examples folder, you see a few folders with names that begin with C17. The C17 MyProgram folder holds a copy of the MyProgram project from Chapter 11. If you open the MyProgram.c file that is a part of the Chapter 17 project, you find that the source code in that file matches the code I just showed you here.

In the next few sections, I discuss some modifications you can make to the source code in this file. Go ahead and give these changes a try. If you're afraid of messing things up, don't be. I'm sure you'll be able to follow along just fine. If you *should* get hopelessly lost though, open the BoxedText.mcp project in the C17 BoxedText folder. The BoxedText.c file in this project contains the changes to the MyProgram source code detailed in this chapter.

Functions Aren't Just for the Toolbox

Every miniprogram in the Toolbox is a function. But the Toolbox isn't the only place you find functions. They also exist in the source code of every C program, and that includes the programs you write. The MyProgram program has a single function, and its name is *main*. (For more information on functions, see Chapters 5 and 15.)

If the Toolbox has functions, why does your program also have to have them? Keep in mind that the primary purpose of a function is to group source code together in an attempt to better organize it. Apple has done that with its own source code and then placed the source code in the Toolbox. Now you have to do the same with your own source code within your source code file.

You can approach functions in a couple of different ways. If your program consists of more than a page full of code, you may want to divvy it up into separate functions. You then have to learn about calling your own functions just as you call Toolbox functions. You also have to learn a lot more about parameters, which are those variables and numbers that appear between the parentheses of a function name. The second approach — used if your program doesn't consist of a whole lot of code — is to just pack all your code into one function and forget about function calling and parameters. Which way sounds easier to you?

All right, one function per program it is! Here's how you do that. All programs written in C must have a function named `main`. That's not just a whim or a preference of mine — it's The Law. So, if your program is going to have just one function, and all C programs must have a function named `main`, guess what the name of your function is? You guessed it: `main`. Your prize is that you get to create a function named `main`.

How do you learn more about writing functions and the parameters that are passed to your own functions? Chapter 21 provides an introduction and a few examples. Chapter 21 also provides a couple of references to Mac programming books that are a little more advanced than this one.

Before creating `main`, look at the form that all functions take. On the first line are all of the following:

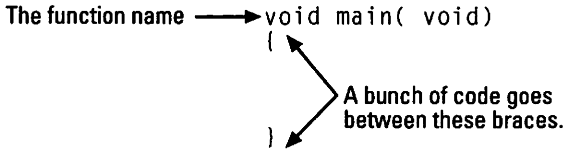
- A return type
- The function name
- An open parentheses
- One or more parameters
- A close parentheses

A function is said to run, or execute. When it's finished running, it can return information to the program. If it does return information, then the return type is the C data type of the returned value. If the function doesn't return a value, then the return type is listed as `void` (as in "the function is void, or without, a return value"). In this book's examples, I'm only writing a single function, and it never returns a value. So until you progress to writing more complex programs, you don't really need to know much more about function return values.

The function name is just that — the name you've elected to give your function. Programs can consist of many functions, though mine just includes one. All programs *must* include a `main` function, so in my example I really didn't have much of a choice in selecting a name for my function.

The opening and closing braces are used to mark the beginning and end of a list of parameters. Previous chapters mention parameters as used in Toolbox functions. Your own functions can also optionally include parameters. A parameter is a value that the function can make use of. You *can* write a function that includes any number of parameters, but my `main` function doesn't include any parameters. Once again the word `void` is used to specify that nothing is here. In this case the function is void, or without, any parameters.

All the source code that follows the first line of the function is nested between a pair of braces. Straight out of Chapter 5 comes this figure showing the basic layout of the function:



```
void main( void)
{

}
```

Look back at the source code for the MyProgram program. Does it follow this format? Yes. It has a single function named `main`. Between the braces that signify the start and end of the function you find ten lines of code that initialize the Toolbox, open a window, and write some text to the window. Yes, it's true: Functions aren't just for the Toolbox.

Initializing the Toolbox

You know what the MyProgram program does, but you probably aren't exactly sure how it does it. The next several sections of this chapter sum up the inner workings of the MyProgram program, starting with the initialization of the Toolbox.

A Mac program starts to run when the user double-clicks its icon. During the running of a program, the program communicates with the Toolbox. And the Toolbox, in turn, communicates with the program. As with humans, before any serious communication can take place, the program and the Toolbox have to get to know each other. That's what the *initialization* process is all about. Calls to the Toolbox that begin with `Init` get this communication under way. MyProgram carries out its initialization with this code:

```
InitGraf( &qd.thePort );
InitFonts();
InitWindows();
```

MyProgram uses three initialization calls, but others exist. For example, to prepare your program to work with menus, you call the Toolbox function named InitMenus. For using dialog boxes, you have InitDialogs. If your program does fancy things with the cursor, you call InitCursor. And if your program has text editing capabilities, you call TEInit.

Most programs also call a Toolbox function named FlushEvents. FlushEvents doesn't have Init in its name, but it is still a part of the program initialization process. I cover events in detail later in this chapter in the section "Making MyProgram More Eventful." For now, realize that events linger in the computer's memory from one running of a program to the next. When a program starts, you want it free of old events — so you flush them. Please don't make me explain this concept with an analogy.

A program with windows, menus, dialog boxes, and all that other fancy stuff I mentioned in the last paragraph has initialization source code that looks like this:

```
InitGraf( &qd.thePort );    /* standard initializations */
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();
```

That seems like a lot of initializations, doesn't it? Are they all really necessary for every program you write, even a simple one like MyProgram? That's an easy one — you should include these eight initializations in *every* Mac program you write. You can't overdo it. If an initialization is unnecessary, it won't hurt your program, and it won't hurt your Mac. What about unnecessary code? You may have heard somewhere that source code should be efficient so that a program runs quickly. That's true. But a call to a Toolbox function such as InitWindows takes only *microseconds* to run — that's *fast.* No one, even your computer, would ever notice.

No matter what your program does, always start it off with these eight lines:

```
InitGraf( &qd.thePort );    /*standard initialization*/
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();
```

When programmers talk about efficient code, they often talk about time-consuming programming tasks, such as redrawing all of the graphics that appear in a window. An issue like that is important in programs that work with large, complex graphics. If you're working with a program that displays photos, for example, you don't want to wait minutes for the Mac to redraw a picture that you just rotated.

Working with a Window

I cover opening a window and the GetNewWindow Toolbox function several times in this book — so I just summarize things for you here. But even if you think you've got this window business licked, read on. Over the course of the next few pages, I'm going to add a little new code to MyProgram regarding windows.

Opening a window

To open a window you first create a 'WIND' resource in your program's resource file. Then, in your source code, you declare a WindowPtr variable. Finally, you call the Toolbox function GetNewWindow to open and display the window. The code for opening a window looks like this:

```
WindowPtr    theWindow;

theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

Writing to a window

Once a window is on the screen, feel free to work with it. But first, don't forget to let the Mac know that you want to write to the new window:

```
SetPort( theWindow );
```

After you clue in the Mac about which window you'd like to write to, you tell the Mac to move to the desired location in the window by calling MoveTo. You then type the following code:

```
MoveTo( 30, 50 );
DrawString( "\pHello, World!" );
```

What about that \p that always precedes the words you want to write? The \p code has to do with *strings,* which are the computer's way of keeping track of several characters — letters, digits, and symbols — in one grouping. In the beginning (about 15 years ago), programmers almost exclusively used the Pascal language to program the Mac. To this day, the Mac still looks for strings in a Pascal format. So the \p before the start of a string tells the Mac, "Here comes a string from the C language, but feel free to put it into the Pascal format you are more familiar with."

Planning an addition to the window

Mac programs contain graphics, and so I want to add a graphic to MyProgram. How about drawing a rectangle to the window that MyProgram opens?

Programming requires planning. Sometimes a little, sometimes a lot. Adding a rectangle is easy, but still, I want to take a moment to plan the rectangle — how it looks, and where it appears on the screen.

To add a rectangle, I need a Rect variable and the use of two QuickDraw functions, which I create with the following code:

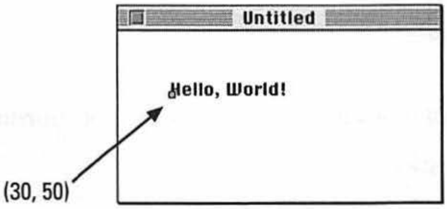
```
Rect   theRect;

SetRect( &theRect, 100, 50, 300, 150 );
FrameRect( &theRect );
```

In MyProgram, I want to have a box around *Hello, World!* to draw the user's attention to it. I want to draw my rectangle in such a way that it frames the text that's written with DrawString.

As you saw in the "Writing to a window" section of this chapter, the following source code moves to a location and then draws a line of text:

```
MoveTo( 30, 50 );
DrawString( "\pHello, World!" );
```

The result is shown in the following figure. I mark the starting point of the text:



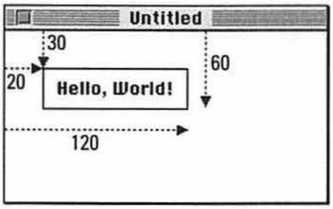
My job is to determine the coordinates for a rectangle that surrounds the *Hello World!* text. I use my knowledge of the Mac's coordinate system and do a little calculating to come up with where the rectangle should be placed. (For more on the Mac's coordinate system, see "The Coordinate System" in Chapter 16.) Whenever Jethro of *The Beverly Hillbillies* needed to do a little math he *commenced to ciphering.* I do the same. I use the following reasoning to come up with the rectangle's coordinates:

- The top must be less than 50 — I choose 30.
- The bottom must be greater than 50 — I choose 60.
- The left side must be less than 30 — I choose 20.
- The right side must extend beyond the end of the text — I choose 120.

Why must the top be less than 50? Because the second parameter in the call to `DrawString` has a value of 50. Recall that the second parameter to `DrawString` tells the Mac how many pixels down from the top of a window drawing should start. Because drawing will begin 50 pixels from the top of the window, I certainly want the top of the framing box to start at a pixel value less than, or higher up in the window than, this value. The same logic applies to the bottom of the framing rectangle, which should start at a pixel value greater than 50 — a pixel value below the bottom of the text.

The same reasoning also applies for choosing the top and bottom values, which explains why I select 20 for the left side of the framing rectangle and 120 for the right side. The call to `DrawString` starts the text 30 pixels from the left of the window, so the left edge of the framing rectangle must start to the left of that value. Any number less than 30 would suffice. As for the right side of the framing rectangle, I admit that was a bit of a guess. But I will give you a little more information about that guess in just a bit.

A rectangle of the dimensions I've discussed looks like this:



Here's what my `SetRect` call looks like, using the previous coordinates:

```
SetRect( &theRect, 20, 30, 120, 60 );
```

Don't forget the order that SetRect expects those four coordinates to appear in — left, top, right, bottom.

For years, I thought that Jethro was just incorrectly using a word when he said he had do some *ciphering*. I just found out that *cipher* means *to solve arithmetical problems*. I have to admit it's more than a little humbling to realize a hillbilly who barely gradjeated the third grade knows things I don't!

More planning for the addition

Now I want to incorporate the rectangle-drawing code into my program. Programmers list a program's variables right near the top of their programs so that the compiler is sure to find them first. In Chapter 5, I do this with the WindowPtr variable named theWindow:

```
void  main( void )
{
   WindowPtr  theWindow;

   InitGraf( &qd.thePort );
   InitFonts();
   /* rest of the code here */
```

I add the Rect variable up front, right by the WindowPtr variable. By the way, the order in which the two variables appear doesn't matter. So it isn't important which of the two variables I list first. Because WindowPtr was there first, I put Rect right underneath it:

```
void  main( void )
{
   WindowPtr   theWindow;
   Rect        theRect;

   InitGraf( &qd.thePort );
   InitFonts();
   /*rest of the code here */
```

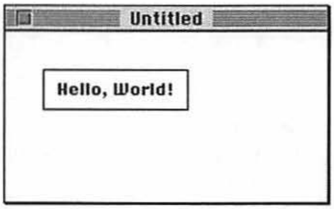
Next, I look for the appropriate place to add the SetRect and FrameRect calls. Because I'm drawing to a window, I want to keep a couple of things in mind as I determine where to add the new code:

- Have I opened a window?
- Have I told the computer that I want to draw to the newly opened window?

The preceding two points mean that my new code should go *after* the call to GetNewWindow and *after* the call to SetPort, as shown in the following code. If I placed the new code before the call to GetNewWindow, there'd be no window to draw to. And if I placed this same new code after the call to GetNewWindow, but before the call to SetPort, I wouldn't be guaranteed that the drawing would end up in the new window. It may, in fact, appear on the screen outside of the window! The following code also adds the remaining Toolbox initializations, as I recommended (several times) earlier in this chapter.

```
void  main( void )
{
   WindowPtr   theWindow;
   Rect        theRect;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( nil );
   FlushEvents( everyEvent, 0 );
   InitCursor();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
    SetPort( theWindow );

   MoveTo( 30, 50 );
   DrawString( "\pHello, World!" );

   SetRect( &theRect, 20, 30, 120, 60 );
   FrameRect( &theRect );

   while ( !Button() )
       ;
}
```

After compiling and running the preceding code, a window just like this appears:



Hello, World!

Fabulous! This is just what I wanted, and on the first try, too. But, as you may already know, the best made plans of mice and programmers sometimes go astray. Sometimes things don't go right on the first try, and then what do you do? Read the next section, of course!

In the code listed previously, I only called SetPort once even though I draw to the window twice. That's because I'm not doing anything between the drawing of the text and the drawing of the rectangle. What if I opened a new, second window (which I call theWindow2) between these two drawing operations:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );

MoveTo( 30, 50 );
DrawString( "\pHello, World!" );

theWindow2 = GetNewWindow( 129, nil, (WindowPtr)-1L );

SetRect( &theRect, 20, 30, 120, 60 );
FrameRect( &theRect );
```

Don't get nervous: It's quiz time. Which window, theWindow or theWindow2, would the rectangle be drawn to?

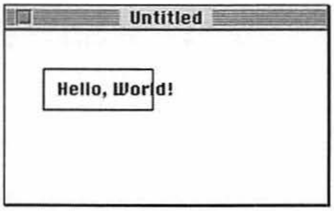
The answer is theWindow. Why? Because its port is the current, or active, port. Recall that each window has a port — that's what the Mac is actually drawing to. Why is the port belonging to theWindow the active port? Because the most recent call to SetPort specified that theWindow is the current port. Now, part two of the quiz. How would you modify the preceding code to have the rectangle drawn to theWindow2? The answer is shown here:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );

MoveTo( 30, 50 );
DrawString( "\pHello, World!" );

theWindow2 = GetNewWindow( 129, nil, (WindowPtr)-1L );
SetPort( theWindow2 );

SetRect( &theRect, 20, 30, 120, 60 );
FrameRect( &theRect );
```

Add another call to SetPort, this time specifying that theWindow2 is the current port.

Solving problems with your box

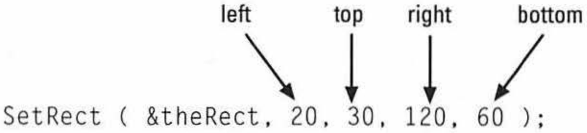
What if I run MyProgram and the window looks like the one in the following figure, rather than the one you see at the end of the section "More planning for the addition"?



Hello, World!

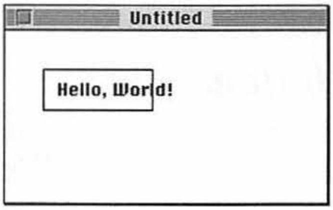
That would mean that I guessed wrong about the size of the rectangle. I based the size of the rectangle on the fact that there are about 72 pixels to an inch. How did I know there are 72 pixels in an inch? I've been programming the Mac for 10 years; I just know stuff like that. How would *you* know there are about 72 pixels to an inch? You may have read about it in Chapter 16 where I mentioned this fact. Otherwise, don't feel bad — there's really no other way you could know that.

Looking at the words *Hello, World!,* I see that they're about an inch long. That's 72 pixels. I then add a little clearance to each side, making the total length of my rectangle 100 pixels. Take a peek at the following figure to double-check the math.



```
SetRect ( &theRect, 20, 30, 120, 60 );
```

The left side of the rectangle is 20 pixels in from the left side of the window. The right side of the rectangle is 120 pixels in from the left side of the window. Jethro could quickly tell us that 120 minus 20 is 100. Now, back to my original question: What would it mean if I run the program and the window looks like this?



Hello, World!

It would mean that my rectangle wasn't big enough. How can I fix this tight fit? I can make the rectangle bigger by setting it bigger during the call to `SetRect`. Here's the rectangle's current code:

```
SetRect( &theRect, 20, 30, 120, 60 );
FrameRect( &theRect );
```

To make the rectangle larger, I make it 130 pixels in length by changing the right side to 150 pixels in from the left side of the window rather than 120. The code for this ground-breaking change looks like this:

```
SetRect( &theRect, 20, 30, 150, 60 );
FrameRect( &theRect );
```

And now you can bet that the rectangle frames the *Hello World!* text nicely. Go ahead and run the new code if you're not sure. I bet you'll be pleasantly surprised!

Making MyProgram More Eventful

I keep telling you that MyProgram is a real Mac program, but it certainly isn't the most exciting program around. I tried to improve things a little bit by adding some more drawing code — the code that draws a rectangle around the program's text. But that really isn't enough. In this section, I show you how to add a few simple lines to MyProgram so that it becomes a more eventful program.

Introducing events

Through some mystical process, your Macintosh is aware of everything you do. Well, everything that involves the Macintosh. Whether you move the mouse, click the mouse button, press a key, or insert a disk, the Mac knows. This power to spy on you, in the hands of an evil computer, could be disastrous. Fortunately, the Mac is a user friendly computer. Because it's friendly, it's willing to share its powers with you. So when the Mac notices that a user has taken some action — like pressing the mouse button — it passes this information on to whatever program is currently running. And if that program happens to be yours? Well then, you can have your program make note of this fact and respond accordingly.

In Macintosh lingo, an action such as a mouse click is called an *event*. Events are the heart and soul of a Macintosh program; events make a Macintosh program run. For example, if the Mac wasn't aware of events, it would never know when the user clicked the mouse on a menu item. It would never know when a key was pressed. It would never . . . well, you get the picture. The Mac would not be the Mac. Events drive a program to do certain things. Not surprisingly, Macintosh programs are called *event-driven* programs.

Looking at the MyProgram event loop

All Macintosh programs have an *event loop*. The purpose of a program's event loop is to repeatedly look for and process events until the user quits the program. At each pass through the loop, the program checks to see if an event has occurred. If an event did occur, the program usually responds in some way. If an event didn't occur, nothing happens. In either case, a moment later, the program is back looking to see if another event has occurred. (For more information on loops, see Chapter 15.)

Two lines of code in the MyProgram program have not yet been discussed in this chapter. They just happen to represent the event loop of the MyProgram program:

```
while ( !Button() )
   ;
```

The code beneath a `while` loop is usually enclosed in braces, like this:

```
while ( booksSold < 100 )
{
   MoveTo( 10, 30 );
   DrawString("\pPoor sales! Pick a new line of work!");
}
```

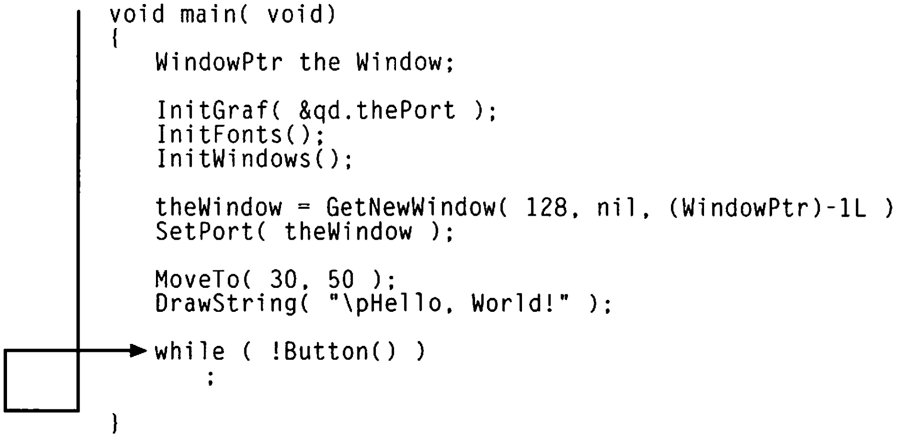
There is an exception to this, however. If one and only one line of code appears beneath the `while` statement, then the braces are optional. I omit the braces in MyProgram for this reason. I can write the MyProgram `while` loop this way:

```
while ( !Button() )
{
   ;
}
```

and the code still has the same effect.

Why did I try to pawn off a second-rate event loop on you? I used it because the MyProgram program is the simplest of programs, and the two-line event loop I used is the simplest of event loops. Even if the event loop in MyProgram isn't highly complex, it's still enough for me to use to explain how event loops work and where it fits in the framework of MyProgram.

The event loop of a program is generally the last piece of source code in the program's `main` function. Once the event loop is reached, that's where the program remains until the user quits the program. In MyProgram, you may think of the flow of the program this way:



```
void main( void)
{
   WindowPtr the Window;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L )
   SetPort( theWindow );

   MoveTo( 30, 50 );
   DrawString( "\pHello, World!" );

   while ( !Button() )
      ;
}
```

Just like any loop, the event loop checks the condition between the parentheses to determine if it's true. In this case, the odd-looking text between the parentheses is a call to a Toolbox function called `Button`. Any time it's called, `Button` tells your program whether or not the mouse button was just clicked by the user.

If the mouse *isn't* clicked, the loop runs. Here the exclamation point in front of the word *Button* assists in the operation. Yes, I know that's an evasive explanation of how the loop works, but trust me — you don't want all the details. What happens when the loop runs? This line of code runs:

```
;
```

What does a semicolon, all alone, do? Absolutely nothing. And for this simple program, that's all I want to happen: nothing. Once MyProgram places a window on the screen and writes to it, the program doesn't do anything at all. It just loops continuously, waiting for the user to click the mouse button. What happens when the user finally clicks the mouse button? If not clicking the mouse causes the loop test to pass and the loop to run, then clicking the mouse must cause the loop test to fail and end the loop. And that's exactly what happens. Where does the program go after a loop ends? To the next line of code that follows the loop, the closing brace:

```
    MoveTo( 30, 50 );
    DrawString( "\pHello, World!" );

    while ( !Button() )
        ;
}
```

The last line of the program

As long as the mouse button isn't clicked, the `while` loop continues to cycle. Once the button is clicked, the `while` test condition fails and the line following the loop code is run. That line is the closing brace of the program. There's nothing left to run, so the program quits.

The MyProgram event loop isn't nearly as powerful as the full-fledged event loop you see later in this chapter. But you can learn a couple of important points about event loops from the MyProgram event loop:

- An event loop is usually based on a C `while` loop.
- Once the event loop is reached, it repeats itself until the program ends.

Holding onto an event

The event loop of MyProgram only looks for a click of the mouse button. But a mouse click isn't the only kind of event the Mac recognizes. The computer also notices when the user presses a key, inserts a disk, and several other actions. As a programmer, you aren't only interested in *if* an event occurred, you want to know *what* event has occurred. Why? You want your program to react in different ways to events, depending on the type of event. For example, if the user clicks the mouse button in the menu bar, you want your program to drop down a menu. If the user inserts a disk in the Mac, that's also an event, but you won't want a menu to drop down for this event.

Obviously, you need some means of storing information about an event. If you were a Mac, you might choose to log information about events like this:

- The event is a mouse click in the menu bar.
- The event is a key press that doesn't involve the mouse

I got tired of listing event types and quit after just two. Fortunately, the Mac doesn't get tired; it stores all the information I show above, and a lot more. It doesn't use a table, of course. Instead, it uses a C data type called the

EventRecord. When you declare a variable to be of the EventRecord type, you then have the means to hold all sorts of information about one event.

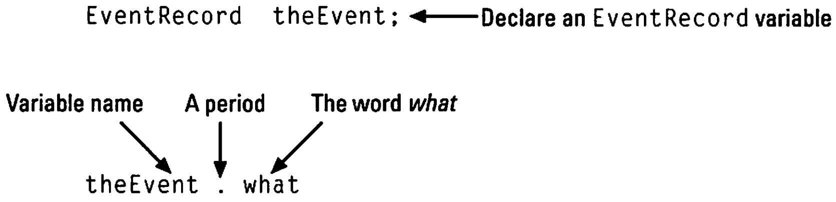
In the next line of code, I declare an EventRecord variable named theEvent:

```
EventRecord  theEvent;
```

An EventRecord variable holds lots of information about an event, but the most important bits of info are the two I list in my chart: the event type and where the event took place.

With some data types, the clever C language allows you to dig two or more separate bits of information out of one single variable. The EventRecord is one such data type. How do you get information about what the event is? You follow the variable name with a period and the word *what* as I do here:

```
theEvent.what
```



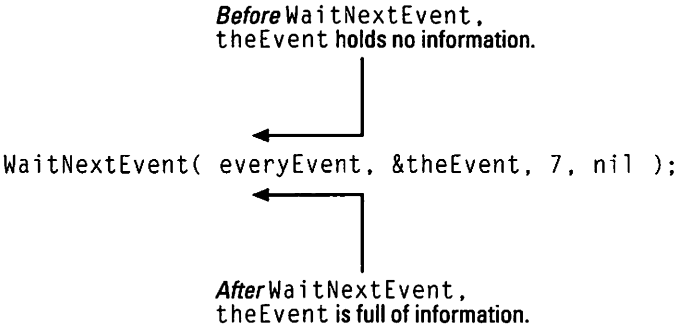
How do you suppose you go about finding out where an event took place? Use the same method as above, but instead of the word *what,* use *where:*

```
theEvent.where
```

Now you know how to get information out of an EventRecord variable, but how does the information get in the variable in the first place? As with most tasks on the Mac, a call to a Toolbox function does the trick. In this case, the appropriate Toolbox call is called WaitNextEvent. The following is a typical call to WaitNextEvent:

```
WaitNextEvent( everyEvent, &theEvent, 7, nil );
```

When your program calls WaitNextEvent, the Mac performs its spy work to see if the user is up to anything. If an event has just occurred, WaitNextEvent gathers all the important information about the event and stores it in the variable named theEvent. Here's what's going on:

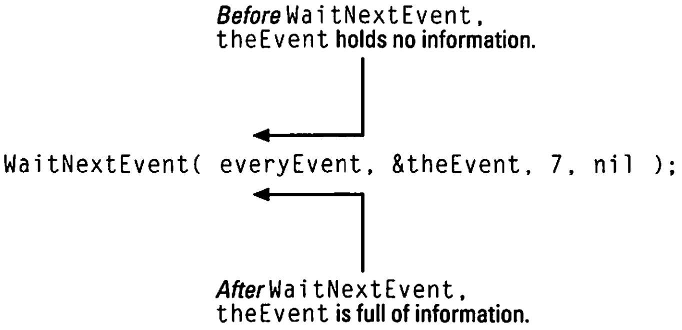


Before `WaitNextEvent`, `theEvent` holds no information.

```
WaitNextEvent( everyEvent, &theEvent, 7, nil );
```

After `WaitNextEvent`, `theEvent` is full of information.

As long as you declare an `EventRecord` variable named `theEvent`, you can use the same four parameters that I did whenever you call `WaitNextEvent`. Recall that the parameters are the items between the parentheses of a call to a function. You don't have to know anything about the other three parameters, but if you're feeling adventurous you can examine this next figure:



Before `WaitNextEvent`, `theEvent` holds no information.

```
WaitNextEvent( everyEvent, &theEvent, 7, nil );
```

After `WaitNextEvent`, `theEvent` is full of information.

Don't forget to type in the & character before the `EventRecord` variable name in the call to `WaitNextEvent`. CodeWarrior won't like it if you forget! If you do forget, CodeWarrior opens up the dreaded Message Window with an error message in it when you attempt to compile the code.

A call to `WaitNextEvent` informs the Toolbox that it should grab hold of the next event it sees and tuck all event-related information into the `EventRecord` variable. But that only takes care of one single event. In a Mac program, events could be happening all the time. If the user clicks the mouse twice, that's more than one event. To make sure that your program doesn't miss events as they happen, it would make good sense to call `WaitNextEvent` over and over again. In fact, that is exactly what your program should do. Here's what you need to accomplish:

- Get information about events.
- Loop through code that calls WaitNextEvent over and over.

In the next section, I show you how to get the job done.

Improving the MyProgram event loop

About the only thing my new-and-improved event loop can't do is fight stains. Where does my new event loop get all its amazing new powers? From good use of the EventRecord data type and the WaitNextEvent function, as you soon see.

Imagine that I've written a program that initializes the Toolbox and then opens a window just as MyProgram does. So that I can concentrate on just the new event loop, I just show the new loop in the following code:

```
allDone = 0;
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   switch ( theEvent.what )
   {
      case keyDown:
         MoveTo( 10, 20 );
         DrawString( "\pKey pressed" );
         break;

      case mouseDown:
         allDone = 1;
         break;
   }
}
```

The new event loop is much larger than the old one, but don't be shocked — some of its elements may seem familiar to you. Take a close look at what the new event loop is doing. First the loop test. I use a short variable called allDone and compare it to the value 1. Is allDone less than 1? Of course — I assign it a value of 0 just before the while statement. When does the loop stop looping? When allDone has a value of 1 or greater. Near the end of the loop is a line that assigns allDone a value of 1, and that takes care of ending the loop. More on ending the loop later in this chapter.

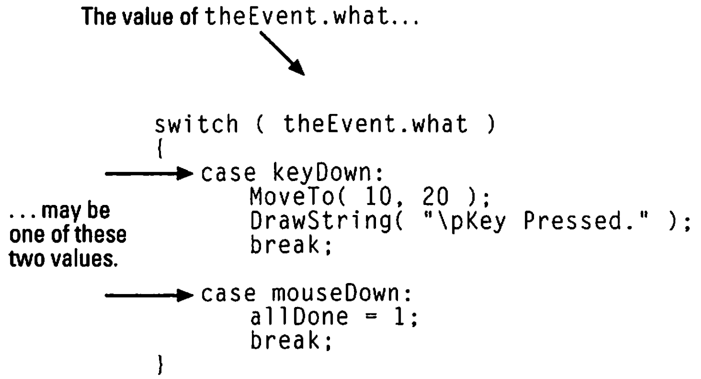
The first thing that happens inside the loop is a call to WaitNextEvent. (Notice that the four parameters of this function are just as I described in the preceding section.) After the call to WaitNextEvent is complete, the variable theEvent holds information about whatever event just occurred.

The primary purpose of a program's event loop is to recognize an event and *handle* it. That means it should take some action appropriate to the type of event that has occurred. That's the purpose of the switch statement. A switch statement is a branching statement that allows the program to run only one section of two or more groups of code (for more on switch statements, see Chapter 14). The switch statement begins with the word *switch* followed by a variable name in parentheses.

For the switch statement in the new event loop, I use the EventRecord variable theEvent, but not the whole variable. At this point, I'm only interested in what type of event occurred (I can wait until later to get any additional info on the event). So naturally I want to examine the what part of the variable, which I can do in the following line of code:

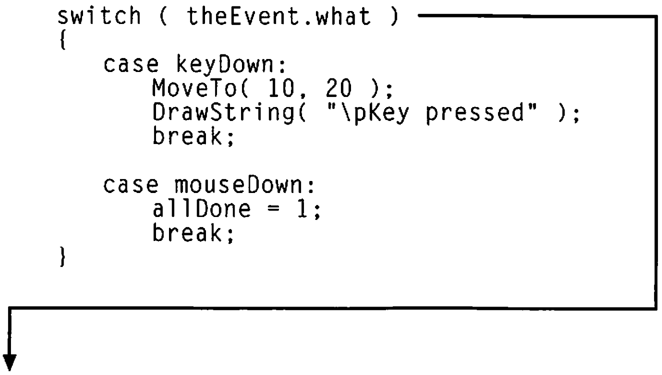
```
switch ( theEvent.what )
```

After the code determines what type of event has occurred, it's on to the case labels. (If case labels seem unfamiliar, see Chapter 14.) Where did the words keyDown and mouseDown come from, and what are they? Both are part of the Toolbox, and can be used in your C source code; they give programmers a way to refer to different event types, and each event has its own name. You can use these names in the case sections of the switch statement by using this code:


The value of theEvent.what...
...may be one of these two values.

```
switch ( theEvent.what )
{
   case keyDown:
      MoveTo( 10, 20 );
      DrawString( "\pKey Pressed." );
      break;
   case mouseDown:
      allDone = 1;
      break;
}
```

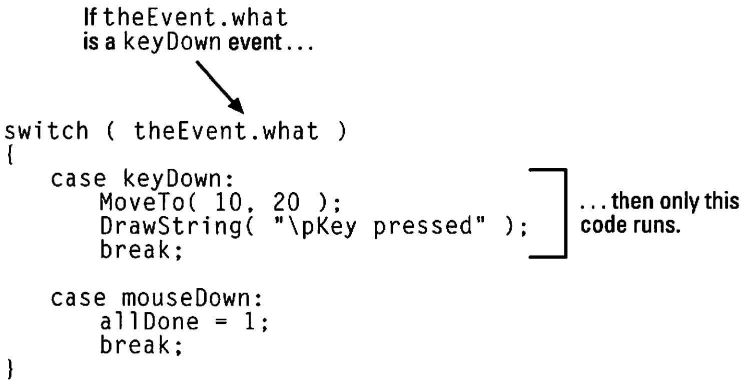
In this figure, I say that the variable theEvent.what may have a value that matches one of the two case values. But what if it doesn't? The two case labels only account for two types of events — a key being pressed (keyDown) and the mouse button being clicked (mouseDown). What if the user inserts a disk? (That's called a diskEvt.) My switch statement doesn't look for an event of this type; if the user performs any other event besides keyDown or mouseDown, nothing happens:



```
switch ( theEvent.what )
{
   case keyDown:
      MoveTo( 10, 20 );
      DrawString( "\pKey pressed" );
      break;

   case mouseDown:
      allDone = 1;
      break;
}
```

If the event doesn't match any of the event types listed in the case labels, the program returns to the top of the loop to see if it should run the loop again. Will it? Let me answer that question with another question. Did the value of allDone change? No. It is still 0. Because 0 is less than 1, the loop runs again, and once again WaitNextEvent is called to capture another event.

Assume that the user presses a key on the keyboard. That constitutes a keyDown event. The call to WaitNextEvent picks up on that and sets theEvent.what equal to keyDown. When the switch statement is reached, the keyDown case runs. Assuming a window is open, the words *Key pressed* are then written to it:



If theEvent.what
is a keyDown event...

```
switch ( theEvent.what )
{
   case keyDown:
      MoveTo( 10, 20 );
      DrawString( "\pKey pressed" );
      break;

   case mouseDown:
      allDone = 1;
      break;
}
```

...then only this
code runs.

What happens if instead of pressing a key, the user clicks the mouse button? The event type is then mouseDown. In my example, when a mouseDown event occurs, allDone assumes a value of 1. Now, what happens when the program returns to the top of the loop statement and makes its test? allDone has a value of 1, which is not less than 1, and the test fails. The entire while loop is skipped, and the program ends.

Examining an Even More Eventful Program

There's no better way to really understand some code than to see it placed in a complete program. In this section, I show you code for a program that features a new-and-improved event loop like the one I discussed in the previous section, but with a twist.

Looking directly into the source code

Because I'm creating a new program, I need to think of a new program name. The purpose of the program is to demonstrate that the Mac knows about events, and so I aptly call it EventTest. Hold on to your hats: I'm about to show you the complete source code listing for the program. In the sections that follow, I break the code down and describe the functions of its various parts.

```
void  main( void )
{
   WindowPtr    theWindow;  /* A window to write to */
   EventRecord  theEvent;   /* Hold info about one event  */
   short        allDone;    /* Tell the program when to end */
   Rect         whiteRect;  /* A rectangle to cover text */
   long         count;      /* A loop counter */

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( nil );
   FlushEvents( everyEvent, 0 );
   InitCursor();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   SetPort( theWindow );

   allDone = 0;
   while ( allDone < 1 )
   {
      WaitNextEvent( everyEvent, &theEvent, 7, nil );
      switch ( theEvent.what )
      {
         case keyDown:
            MoveTo( 10, 20 );
            DrawString( "\pKey pressed" );
            count = 0;
            while ( count < 2000000 )
               count++;
            SetRect( &whiteRect, 10, 10, 100, 25 );
            FillRect( &whiteRect, &qd.white );
            break;
         case mouseDown:
            allDone = 1;
            break;
      }
   }
}
```

Extending a friendly reminder

You can find the source code for the EventTest program in the EventTest.c file in the C17 EventTest folder. If you're using CodeWarrior Lite, open the EventTest.mcp project so that you can take a look at the code.

If you own the full-featured version of CodeWarrior, now is as good a time as any to try your hand at creating a Mac program from scratch. If you don't recall just how that's done, go back and review a few chapters. EventTest uses one window, and so it needs a resource file with a single 'WIND' resource. If you don't remember the exact steps to creating a resource file, you can refer to Chapter 7. Chapter 8 discusses the 'WIND' resource in detail. A CodeWarrior program starts out as a project. Creating a project is covered in Chapter 9. If this all sounds too confusing, read Chapter 19 before tackling the task of creating a new project. In Chapter 19, I fully describe creating a complete Macintosh program from start to finish. That includes the basics — from conjuring the resource file to making the source code file.

Examining the basic stuff

Like all programs, EventTest starts with the declaration of variables. Here I comment on all of the variables, and I describe each variable as I get to the code that uses it:

```
WindowPtr    theWindow;  /* A window to write to        */
EventRecord theEvent;   /* Hold info about one event  */
short       allDone;    /* End of program */
Rect        whiteRect;  /* Used to cover text */
long        count;      /* A loop counter */
```

After the variable declarations come the Toolbox initializations. These eight lines are lifted directly from another program I wrote. Remember, I advise that you use these same eight Toolbox initializations in every program you write:

```
InitGraf( &qd.thePort );
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();
```

The last bit of code for the basic stuff is for the opening of a window. Calling `GetNewWindow` provides my program with a `WindowPtr` that can be used in other Toolbox calls, like `SetPort`. These two lines take care of that job:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );
```

Examining the event loop

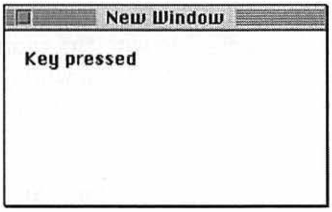
The event loop of EventTest should be a welcome sight to you. It is the very same event loop I showed you earlier in this chapter — with just five extra lines added to it:

```
allDone = 0;
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   switch ( theEvent.what )
   {
      case keyDown:
         MoveTo( 10, 20 );
         DrawString( "\pKey pressed" );
         count = 0;
         while ( count < 2000000 )
            count++;
         SetRect( &whiteRect, 10, 10, 100, 25 );
         FillRect( &whiteRect, &qd.white );
         break;

      case mouseDown:
         allDone = 1;
         break;
   }
}
```

The new lines appear under the `keyDown` `case` of the `switch` statement. Here's how the old `keyDown` section looked:

```
case keyDown:
   MoveTo( 10, 20 );
   DrawString( "\pKey pressed" );
   break;
```

In the old keyDown code, pressing a key caused the program to write to the window like this:



Key pressed

All fine and dandy, you say. But what happens if the user again presses a key? The same code runs, and the same words write to the window in the exact same spot. After the first press of a key, the window goes from one that is blank to one that has words written in it. But after that, the user has no way of knowing that words are being written. The solution? Erase the words soon after displaying them. A key press then results in the words *Key pressed* flashing on and off. Take a look at how EventTest does that.

After writing the words *Key pressed* to the window, the variable count assumes a value of 0. Then the program goes into a very simple loop:

```
case keyDown:
   MoveTo( 10, 20 );
   DrawString( "\pKey pressed" );
   count = 0;
   while ( count < 2000000 )
      count++;
```

The loop doesn't appear to be doing much except counting. Variable count starts with a value of 0, and increases by a value of 1 each time through the loop. The ++ symbol is the increment operator, and when it appears next to a variable name, the variable is incremented by one. So, what good is a loop that does nothing but count from 1 to 2,000,000? Not much, unless you want to kill some time. That's exactly what the loop is doing. While the Mac is busy counting up to two million, it doesn't do anything else. That means the loop is delaying the rest of the program from running. So what's really important is what *follows* the loop, that is, what code is being delayed.

The loop has the extra benefit of showing just how fast a Mac runs. The loop runs 2,000,000 times and takes a few seconds to run on an older Mac II model. It takes only about a quarter of a second on a much newer Power Macintosh. In either case, that's a pretty fast way to count to 2,000,000!

The two lines of code that follow the loop are calls to the Toolbox functions `SetRect` and `FillRect`. `SetRect` establishes the coordinates of a rectangle, but doesn't draw it (for more on `SetRect`, see Chapter 16). The coordinates I select create a rectangle that surrounds the words *Key pressed*. But I don't want to place a frame around these words as the Toolbox function `FrameRect` would. Rather, I want to fill this rectangle with the same color of the window. The window is white with black text. If I draw a white rectangle over the black text I effectively obscure the text — it will be erased. I use `FillRect` to do this.

The `FillRect` function draws a solid rectangle with no frame. You give `FillRect` two parameters, which are the `Rect` variable that should be filled and the shade to fill it with. Here's the code that sets up a rectangle and then fills it with `white`:

```
SetRect( &whiteRect, 10, 10, 100, 25 );
FillRect( &whiteRect, &qd.white );
```

The second parameter to `FillRect`, the shading for the rectangle, must be preceded by an ampersand (&), the letters *qd*, and a period. Failure to do so results in an error when you attempt to compile the source code!

By the way, you can use a few other shades with `FillRect`. The `FillRect` Toolbox function allows you to fill a rectangle with white, light gray, gray, dark gray, or black. Here's an example of each. If you use any of these shades, make sure to use the same combination of uppercase and lowercase characters as shown here:

```
FillRect( &theRect, &qd.white );
FillRect( &theRect, &qd.ltGray );
FillRect( &theRect, &qd.gray );
FillRect( &theRect, &qd.dkGray );
FillRect( &theRect, &qd.black );
```

Events and the event loop are two of the most important and powerful concepts in Macintosh programming, which is why this chapter is one of the longest in the book. The event loop gives a Mac program the ability to react differently in different circumstances. One of those circumstances is a mouse click in the menu bar.

Now that you've gained knowledge about the event loop, it's time to go on to Chapter 18 where you look at a sample program that includes a menu.

Chapter 18

Menus That Drop and Windows That Move

In This Chapter

- Dissecting the different parts of a window and the screen
- Dragging a window around on the screen
- Closing a window
- Examining a program with a movable, closable window
- Displaying a menu bar
- Displaying a pull-down menu
- Handling a user's menu selection
- Examining a program with a functioning menu

Sometimes a chapter in a book ends with a review. Just to be different, I'm starting a chapter with one. Here's a review of what you can make your Mac programs do:

- ✓ Initialize the Toolbox.
- ✓ Open a window.
- ✓ Draw text and graphics in a window.
- ✓ Respond to user actions, such as mouse clicks.

The points I just listed are very important topics in Macintosh programming. But there are a still a couple of topics you need to have under your belt in order for your programs to have the true look and feel of a Mac program.

The first missing feature deals with windows. Yes, you can open a window. But it sits lifeless on the screen as if it were frozen. The user can't move it or close it. In fact, if the user tries to do anything with the window by clicking the mouse on it, the program ends. The second missing feature is a biggie — menus. To avoid the scorn and ridicule of other Mac programmers, your programs need to feature menus. I address both of these issues in this chapter.

Bringing a Window to Life

A true Mac program doesn't just display a window, it also lets the user work with the window. A user expects to be able to drag, or move, a window. And when done with the window, the user wants to be able to close that same window. Those are the two topics I cover in the next few sections.

Dissecting the parts of a window

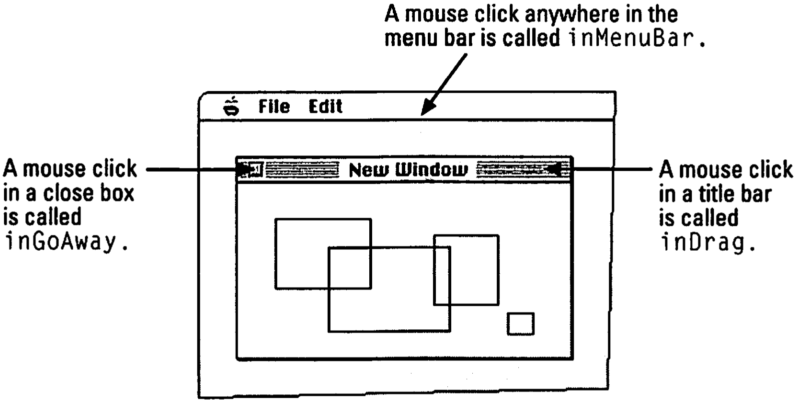
Each of the different event types has a name, such as `mouseDown` and `keyDown`. Knowing that, it may not come as a surprise to you to discover that the different parts of the screen, and windows on the screen, also have names.

When the user clicks the mouse button, a `mouseDown` event is reported to your program. What should your program do next? In programs in other chapters, I simply use a mouse click as a signal to quit the program. A better response would be for the program to determine where the mouse click took place and then act accordingly. If the mouse click took place in the menu bar, the program should drop a menu. If it took place in the close box of a window, the program should close that window. And if the mouse click took place in the title bar (the drag bar) of a window, my program should start dragging the window.

The Mac screen and any window on it has several different parts. The following list details the names of the three most common parts:

- ✓ `inMenuBar` is anywhere in the menu bar.
- ✓ `inGoAway` is the go-away, or close box, of a window.
- ✓ `inDrag` is the drag bar, or title bar, of a window.

Notice that the part names begin with *in*. That gives you a hint as to how these part names are used by your program. If the mouse is clicked *in* the menu bar, drop a menu. If the mouse is clicked *in* a window's close box, close the window. Here's a figure that emphasizes where the three main parts are located:


A mouse click anywhere in the menu bar is called inMenuBar.

A mouse click in a close box is called inGoAway.

A mouse click in a title bar is called inDrag.

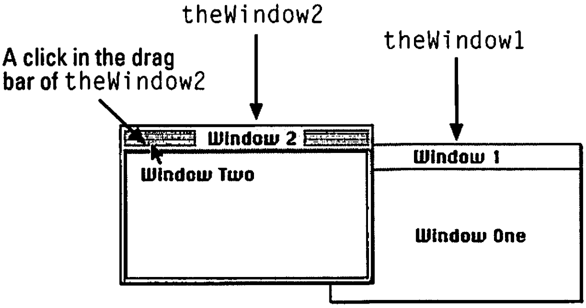
Clicking different parts of a window

Knowing where a mouse click took place is very useful information that your program can and does take advantage of. But for some mouse clicks, still more information is needed. When the user clicks the mouse anywhere in a window, your program wants to know not only where the click took place, but also in which window it occurred.

Take a look at the case of a program that has two windows open. For two windows to open, you have to write the following code:

```
WindowPtr   theWindow1;
WindowPtr   theWindow2;

theWindow1 = GetNewWindow( 128, nil, (WindowPtr)-1L );
theWindow2 = GetNewWindow( 129, nil, (WindowPtr)-1L );
```

To drag one of the windows, a user clicks the mouse on the window's drag bar. In the figure below, you can see that the cursor is over the drag bar of theWindow2.


theWindow2

theWindow1

A click in the drag bar of theWindow2

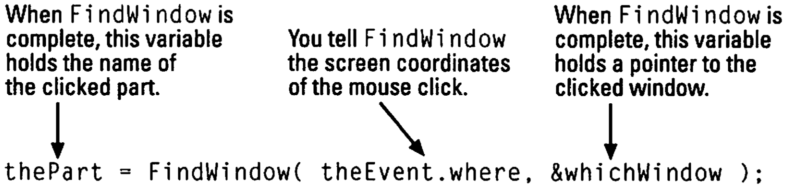
A click in the drag bar of theWindow2 tells the program that an inDrag part has been clicked. Your program then wants to start dragging the window as the user moves the mouse. But one important bit of information is missing — which window should be dragged? You know it's the window that theWindow2 is pointing to because you've looked at the previous figure. But the program isn't reading this book, so it doesn't know. But, just like a good international spy, it has ways of finding out.

The program needs several things in this situation: a way to determine what part is clicked on, and, if a window is involved, the window in which the click took place. Naturally, the Toolbox offers the answer.

The Toolbox function FindWindow gives your program the mouse click information it needs. FindWindow isn't psychic, though; you need to help the function by supplying the location of the mouse click. Thankfully, coming up with this location requires no effort on your part. The EventRecord data type and a call to WaitNextEvent fill FindWindow with information about an event. One bit of information is the type of event that occurred. Another piece of information filled in for you is where the cursor is when the event occurs. That *where* information is of high value to you at this time. The following code shows a typical call to FindWindow. For clarity, the code also shows the declarations of the variables that it uses.

```
EventRecord   theEvent;
WindowPtr     whichWindow;
short         thePart;

thePart = FindWindow( theEvent.where, &whichWindow );
```

After the call to FindWindow, the short variable thePart holds the name of the part that was clicked. That is, thePart has a value such as inDrag or inMenuBar. The last parameter to FindWindow also takes on a value it didn't have before the function call. Before calling FindWindow, you declare a new WindowPtr variable, but you don't open a window. Instead, you use this valueless variable as the second parameter to FindWindow. If FindWindow determines that the mouse click occurred in a window, it gives this variable a value. The value is a pointer to the clicked-in window, a WindowPtr. Here's a breakdown of the call to FindWindow:



When FindWindow is complete, this variable holds the name of the clicked part.

You tell FindWindow the screen coordinates of the mouse click.

When FindWindow is complete, this variable holds a pointer to the clicked window.

```
thePart = FindWindow( theEvent.where, &whichWindow );
```

How can a short variable (which is a number) end up having a value that is a name? That seems to be the case when thePart is declared to be a short, and is then given a value such as inDrag by the FindWindow function. The answer lies in the way the Mac keeps track of some names, like inDrag, inGoAway, and inMenuBar. While you see these names as words, the Mac associates a number with each name. So when thePart gets a value of inDrag, the Mac is secretly doing something like this:

```
thePart = 4;     /* The Mac views inDrag as the number 4 */
```

Tricky, no? The thing to remember is that the Mac does all this converting of names to numbers behind closed doors. You don't see it, and you don't have to worry about it. Now that's reason to celebrate.

After a call to FindWindow, your program knows

- Which part of the screen, or window, is clicked.
- Whether a window is clicked.
- Which window, if any, is clicked.

Variable thePart holds the first two pieces of information. If thePart has a value that pertains to a window, such as inDrag, your program knows that a window was clicked and where in that window the click took place. The third piece of information is held in variable whichWindow. Ifthe mouse click was in a window, FindWindow gives this variable a value, a pointer that tells the Mac which window received the mouse click.

That does it for the theory behind the determination of where a mouse click took place. It's time to put the theory into practice.

Working with windows can be a drag

Wait! Wait a second. Before you jump right into the code for dragging a window, you need to figure out where the code should go. You don't have a clue as to where it should go? Well here's a hint: A look at the code for the event loop may come in handy very quickly:

```
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   switch ( theEvent.what )
   {
      case keyDown:
         /* do something */
         break;
      case mouseDown:
         /* do something */
         break;
   }
}
```

Instead of actually doing anything when a key or the mouse button is pressed, I've just stuck a comment in the code. That doesn't make the code very useful, but it makes it nice and short and easy to look at.

A window gets dragged when the user clicks the mouse button on the drag bar of the window. That tells you the code is going to go under the `case` label `mouseDown`. I insert the appropriate code under the `case` label in the following listing of code. Take a look at it, scratch your head if you need to, and then move on to the explanation.

```
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   switch ( theEvent.what )
   {
      case keyDown:
         /* do something */
         break;

      case mouseDown:
         thePart = FindWindow( theEvent.where, &whichWindow);
         switch ( thePart )
         {
            case inDrag:
               DragWindow( whichWindow, theEvent.where,
                           &qd.screenBits.bounds );
               break;
         }
         break;
   }
}
```

The first line of new code is the call to `FindWindow`. This Toolbox function returns the part of the screen or window that was clicked, along with a pointer to the window that was clicked, if any. The only other code I add is a `switch` statement. The `switch` examines the value of `thePart`. It has a value such as `inDrag`, `inGoAway`, or `inMenuBar`. At this time, I'm only demonstrating how to respond to a click in the drag region, so that's the only `case` label I put in the `switch`. A mouse click anywhere else is simply ignored by the program.

The following figure shows what goes on in the event loop when the user clicks the mouse on the drag bar of a window. I don't like working in cramped quarters, so I've taken the liberty of inserting a few blank lines that give me room to add those arrows I like to draw:

Program ends up here if the mouse button is clicked.

Determine at what part of the screen or window the click took place.

```
case mouseDown:

    thePart = FindWindow( theEvent.where, &whichWindow );

    switch ( thePart )
    {

        case inDrag:

            DragWindow( whichWindow, theEvent.where, &qd.screenBits.bounds );
            break;
    }
```

Compare the part of the screen (or window) with the case label (or labels) that appear below.

If thePart has a value of inDrag, the program ends up here.

Drag (move) the window in response to the user moving the mouse.

When it comes right down to it, only one line of code actually drags a window. The one important line contains the call to the Toolbox function DragWindow. When a user moves a window on the screen, it is DragWindow that's doing all the work.

DragWindow needs three parameters in order to work. The first, the one I called whichWindow, is the window that is to be dragged. Remember, whichWindow isn't a window that you opened using GetNewWindow. It's a WindowPtr variable that you declared with the express purpose of using it in the call to FindWindow. When the user clicks on a window, whether there are one, two, or ten windows on the screen, the FindWindow function figures out which one received the mouse click and puts a pointer to it in the whichWindow variable. Now is the time, when a window is being dragged, to use this WindowPtr variable.

The second parameter of DragWindow is the screen coordinates of the mouse click. That helps DragWindow get started as it moves the window. The last parameter is a strange-looking one called &qd.screenBits.bounds. The purpose of this parameter is to tell DragWindow just how far a window can be dragged. When you set this parameter to &qd.screenBits.bounds, you're telling the Toolbox that it can feel free to move the window anywhere on the screen — the entire area of the screen is available.

When you call a Toolbox function, the function typically runs in a blink of the eye. For example, when a program calls FrameRect to draw a rectangle, the rectangle is drawn almost immediately. DragWindow is an interesting function in that it stays running for as long as the user holds the mouse down on a window's drag bar. When the user starts moving a window around the screen, it's the DragWindow function that's doing the work. If you were to drag a window around the screen for 10 seconds, that's how long the `DragWindow` function would take to run.

Closing a window

After you know the procedure for dragging a window, closing one is a breeze. A window gets closed when the user clicks the mouse on the window's close box (or go-away box to some). That means your program must first determine whether a `mouseDown` event occurs. You know how to get your program to do that. Next, a call to `FindWindow` is in order to see whether the mouse click happened `inGoAway` (in the go-away box of the window). That you know how to do. Next, a `case` label is needed inside a `switch` statement. That you know how to do. Last, code has to be added to actually close the window. That you don't know how to do. But after looking at this one line, you will know:

```
DisposeWindow( whichWindow );
```

Adding this one line of code does the trick. The Toolbox function `DisposeWindow` closes a window. All you have to do is tell `DisposeWindow` which window to close. And that information you already have from the call to `FindWindow`. I add a `case` label `inGoAway` to the very code I used when I demonstrated how to drag a window. I put the new code in bold type so that you can see it better. You can thank me later:

```
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   switch ( theEvent.what )
   {
      case mouseDown:
         thePart = FindWindow( theEvent.where, &whichWindow);
         switch ( thePart )
         {
            case inDrag:
               DragWindow( whichWindow, theEvent.where,
               &qd.screenBits.bounds );
               break;
            case inGoAway:
               DisposeWindow( whichWindow );
               break;
         }
         break;
   }
}
```

Working windows and breaking out of the loop

I'm sure that you now have a pretty good idea of how to drag and close a window. Just to reinforce the concepts, here's a complete listing of the code for a program that does everything I talk about in this chapter.

This program, which I call WindowWorks, has only one point that may need to be explained. In order to write a program that doesn't run forever, you must always add a line of code somewhere that breaks the program out of the event loop. In my programs, that line always looks like this:

```
allDone = 1;
```

Where you put that line depends on the program you're writing. It makes most sense to use it when the user selects Quit from a menu, but my program doesn't have a menu. So I elect to place this line right after the call to `DisposeWindow`. That means that when the user clicks the window's close box, the window closes and the program ends.

```
void  main( void )
{
   WindowPtr    theWindow;
   EventRecord  theEvent;
   short        allDone;
   WindowPtr    whichWindow;
   short        thePart;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( nil );
   FlushEvents( everyEvent, 0 );
   InitCursor();

   theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
   SetPort( theWindow );

   allDone = 0;
   while ( allDone < 1 )
   {
      WaitNextEvent( everyEvent, &theEvent, 7, nil );
      switch ( theEvent.what )
      {
         case mouseDown:
            thePart = FindWindow( theEvent.where, &whichWindow );
            switch ( thePart )
            {
               case inDrag:
```

(continued)

(continued)

```
               DragWindow( whichWindow, theEvent.where,
                           &qd.screenBits.bounds );
               break;
            case inGoAway:
               DisposeWindow( whichWindow );
               allDone = 1;
               break;
         }
         break;
      }
   }
}
```

It's often said that simplicity is beautiful. But there are certainly other important features that you need to be able to add to your programs to make them look and act like real, for-certain Mac programs. Adding menus has to be at the top of your list at this point.

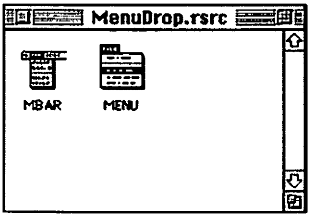
Dropping That Menu

A person can write tons of source code to show off many Macintosh programming concepts without ever including the code for a menu. I know this because I have accomplished this amazing task after writing 17 chapters of this book. Now, it's finally time to create a program that's much closer to being a real Mac application. That's right: It's finally time to belly up to the menu bar for some menus.

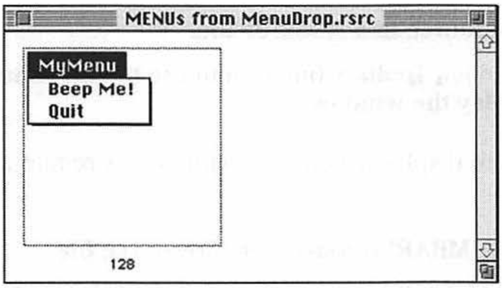
Running through the menu resources

The program that I show you at the end of this chapter is called MenuDrop. Before looking at code for a menu, I want to give you a quick peek at the resources that the program uses.

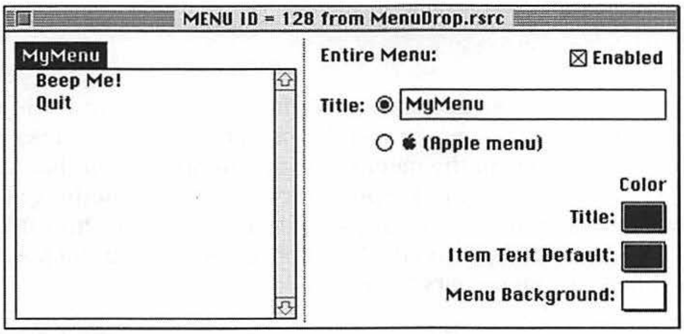
The first thing you may notice about the resource file shown below is that there is no 'WIND' resource. My MenuDrop program doesn't use one because it only displays a single menu in the menu bar.



Clicking the MENU icon once shows the 'MENU' resources in this resource file. You find only one:



Double-clicking the menu in the previous figure opens the 'MENU' editor. Here's what the one 'MENU' in MenuDrop looks like:



Entire Menu: ☒ Enabled

Title: ● MyMenu

○ (Apple menu)

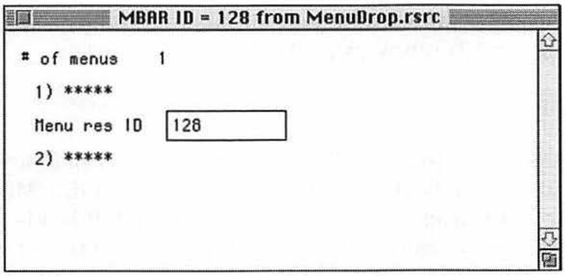
Color

Title: ■

Item Text Default: ■

Menu Background: □

Whether your program has one menu or ten, it should have an 'MBAR' resource that lists each one. Here's the 'MBAR' for MenuDrop:



```
# of menus    1
  1) *****
  Menu res ID  128
  2) *****
```

Displaying the menu bar

To get a Mac program to display a window, you use two steps:

1. **Create a 'WIND' resource in a resource file.**
2. **Call the** GetNewWindow **Toolbox function to use the information in the resource and display the window.**

Getting a Mac program to display a menu bar with menus requires two similar steps:

1. **Create 'MENU' and 'MBAR' resources in a resource file.**
2. **Call the** GetNewMBar **Toolbox function to use the information in the resources and display the menu bar.**

Here's a call to GetNewMBar, which is the Toolbox function that takes information from an 'MBAR' resource and uses it to display a menu:

```
Handle  menuBarHandle;

menuBarHandle = GetNewMBar( 128 );
```

Take note of the variable declaration. Variable menuBarHandle is a Handle type, a type that may be new to you. To explain Handle, you may recall that a lot of things get stored in the memory of a computer. When that happens, your program needs a way of keeping track of where something is stored in memory. The data type Handle does just that. Thankfully, how it keeps track of memory isn't important to you. The bottom line is that the Mac keeps track of things like menu bars by using a Handle.

GetNewMBar accepts a single parameter — the resource ID of the 'MBAR' that holds information about the menus that are to appear in the menu bar. If you create one 'MBAR' resource (as is usually the case), ResEdit automatically assigns it an ID of 128. That's why I've used 128 as the parameter to GetNewMBar in my example.

After a call to GetNewMBar, the variable menuBarHandle has a value. That value is used in a second Toolbox call, which is SetMenuBar:

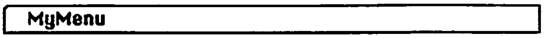
```
SetMenuBar( menuBarHandle );
```

When you call GetNewMBar from the Toolbox, it does a lot of fancy things to get all the information from both the 'MENU' resource and the 'MBAR' resource located in your program's resource file. Though it holds on to all this information, it doesn't really do much with it. The call to SetMenuBar is the thing that establishes that yes, this information should and will be used for the menu bar for this program.

After GetNewMBar and SetMenuBar, you'd think that the menu bar would be there on your screen. Not quite. True, everything is safely stored in memory and ready, but the Toolbox still wants you to call DrawMenuBar in order to display the menu bar at the top of the Mac's screen. Here's the call to DrawMenuBar, along with the other two Toolbox calls necessary for displaying a menu bar:

```
Handle  menuBarHandle;

menuBarHandle = GetNewMBar( 128 );
SetMenuBar( menuBarHandle );
DrawMenuBar();
```

If I use this code along with the 'MENU' and 'MBAR' resources I created several pages back, I get a menu bar that looks like this:



You know that my 'MBAR' holds the ID of one 'MENU' resource. That 'MENU' is for a menu that has two items — one called Beep Me! and the other called Quit. You may be wondering why I didn't show MyMenu dropped down to display these two items. I could have included that in my figure, of course. Heck, I can draw a menu bar and menu in my graphics program any way I want! But I wanted to be realistic. The code that I show in this section only displays a menu bar and the names of the menus in that bar. It doesn't do anything to make the menu bar functional. That is, if you include the code I just showed you in a program, you'd see a menu bar, but you wouldn't be able to use it. You know what's about to happen: You need to add code to make the menu usable.

Pulling down a menu

You know about the parts of the screen and the parts of the windows on the screen such as inDrag, inGoAway, and inMenuBar. You also know that through the use of the Toolbox function FindWindow, you can find out in which of these parts a mouse click occurs. You also know that because the moving and closing of a window both involve a click of the mouse, the code that is used to carry out these actions is added to the event loop under the case label mouseDown. (If any of this sounds unfamiliar, refer to the sections at the beginning of this chapter.) All this knowledge serves you well when you want to understand how menus work.

The only time your program pulls down a menu is when the user clicks the mouse in the menu bar. So it's important that your program be aware of when the user clicks the mouse. To clue your program in on mouse clicks, you call

FindWindow from the Toolbox when working with a window. The code below should look familiar because it's from this chapter. I've added a case label for inMenuBar, but I haven't added any menu-handling code just yet.

```
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   switch ( theEvent.what )
   {
      case mouseDown:
         thePart = FindWindow( theEvent.where, &whichWindow);
         switch ( thePart )
         {
            case inDrag:
               DragWindow( whichWindow, theEvent.where,
                           &qd.screenBits.bounds );
               break;
            case inMenuBar:
               /* handle a click in the menu */
               break;
         }
         break;
   }
}
```

If there's a click in the menu bar, FindWindow assigns variable thePart a value of inMenuBar. Because no window is involved in a menu bar click, variable whichWindow is left without a value.

What happens when a mouse click turns out to be in the menu bar? The code under inMenuBar runs. I show you most of the code for running inMenuBar here:

```
case inMenuBar:
   menuAndItem = MenuSelect( theEvent.where );
   if ( menuAndItem > 0 )
   {
      theMenu = HiWord( menuAndItem );
      theMenuItem = LoWord( menuAndItem );
      switch ( theMenu )
      {
         /* handle each menu item here */
      }
      HiliteMenu( 0 );
   }
   break;
```

You may notice in the preceding code that rather than address the selection of each menu item, I simply insert a comment. How's that for getting off easy? Actually, I have my reasons for omitting some code. First, I want to keep the code minimal so it's easier to look at and easier to explain. Second, each program handles menu items differently. That is, code for a menu item is dependent on what that menu item is supposed to do. But don't feel like you're being cheated. I show you the code for my two menu items, Beep Me! and Quit, a little later in this chapter.

On to the explanation of the inMenuBar code. After a click in the menu bar is detected, the code calls the Toolbox function MenuSelect. This is one of those Toolbox functions Mac programmers go ga-ga over. Why? Because it's one of those functions that does a lot, and thus saves you a lot of work. Once called, the astonishingly functional MenuSelect does the following:

- Follows the mouse as it moves about in the menu bar.
- Shows and hides menus as the mouse moves over them.
- Flashes a selected menu item a few times.
- Highlights the name of the menu that's been selected in the menu bar.
- Tells your program which menu item has been selected — and from which menu.

Wow! Hopefully the Toolbox pays MenuSelect at least time and a half! The only things MenuSelect needs in order to do all this work are the screen coordinates at which the mouse button was clicked. Just pass theEvent.where as the parameter and MenuSelect does the rest.

After dragging the mouse here and there across the menu bar, the user eventually settles on a menu item and selects it. At that time, MenuSelect considers its work done. As the Toolbox function ends, it returns a number to your program. This number is a code that represents both the selected menu item and the menu from which the item was selected. Save the number as a long variable — I name mine menuAndItem. Here's the call to MenuSelect, along with a reminder of how to declare a long variable:

```
long   menuAndItem;

menuAndItem = MenuSelect( theEvent.where );
```

Remember, a long is a whole number that can have a value larger than 32,767. And as a further reminder, a short and an int are two other C data types that hold whole numbers.

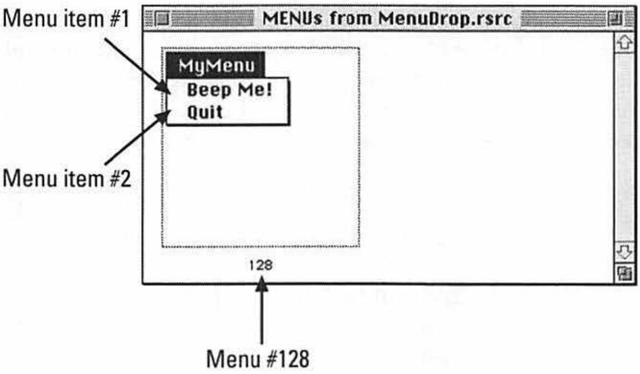
Making the menu usable

If I stopped writing code right now, my program would appear to behave much as any Mac program should. If the user clicked on the one menu in my program's menu bar, the menu would drop down. The user could select either item and the menu item would flash a few times and the menu would disappear back into the menu bar. All thanks to a call to MenuSelect. What would happen next? Nothing. That's because I haven't written any code to handle the menu item selection.

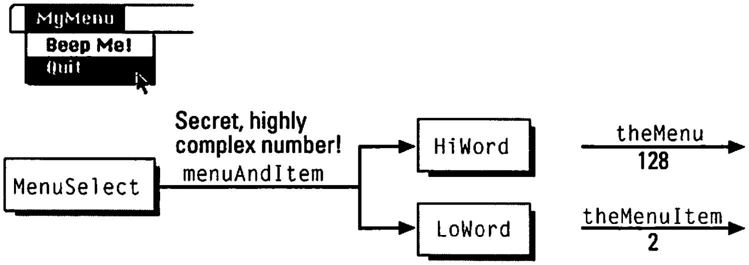
Any of a thousand different things can happen when a user of a program makes a menu selection — but you decide what, not the Mac. That implies that your program has to become aware of which menu item was selected so that it can respond accordingly. Your program does have that information — kind of. It was returned to variable menuAndItem by MenuSelect. But the two values, the menu and the menu item, are both bundled into this one number. To break the code and separate them you use two Toolbox functions — HiWord and LoWord. Here's how these two functions are used:

```
short  theMenu;
short  theMenuItem;

theMenu = HiWord( menuAndItem );
theMenuItem = LoWord( menuAndItem );
```

You pass both HiWord and LoWord the variable that holds the combined menu/menu item value — menuAndItem. As each function ends, it returns a new number to your program. HiWord returns a number that represents the menu selected, LoWord returns a number that represents the menu item selected. I use the 'MENU' resource I developed earlier as an example of what these numbers mean. Here's that resource:



The Mac views a menu by the resource ID of its 'MENU' resource. It gives each item in a menu a number associated with its order in the menu. The first menu item is 1, the second is 2, and so forth. If a user selected Quit from my example menu, here's what would happen:



The previous figure shows that MenuSelect spits out menuAndItem, a number in a form far too complicated for mere mortals to understand. menuAndItem is then passed to both HiWord and LoWord. HiWord returns the resource ID of the selected menu, which is 128 for my example. LoWord returns the number of the selected menu item, which is the second item, or number 2 for my example.

I didn't get the Official Menu Spy Decoder Ring with my Mac, so I'm not exactly sure how HiWord and LoWord manage to extract both the menu and the menu item from this one number. But from my experience with programming the Mac, they always seem to get it right.

Let's take another look at the case inMenuBar, and then cover the sections of it that haven't yet been discussed:

```
case inMenuBar:
   menuAndItem = MenuSelect( theEvent.where );
   if ( menuAndItem > 0 )
   {
      theMenu = HiWord( menuAndItem );
      theMenuItem = LoWord( menuAndItem );
      switch ( theMenu )
      {
         /* handle each menu item here */
      }
      HiliteMenu( 0 );
   }
   break;
```

Did you ever start to make a menu selection in a program and then change your mind? When a user does that, MenuSelect returns a value of 0 to the program; it sets menuAndItem to 0. That tells the program that, yes, menus were dropped and looked at, but no selection was made. In a case such as this, a program won't want to go through the work of deciphering menuAndItem with HiWord and LoWord — no menu or menu item numbers are embedded in this variable. In fact, because the user decided not to do anything, the program won't want to do anything either. That's the reason for the if statement after MenuSelect.

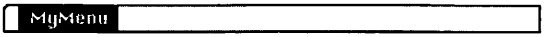
If no menu choice is made, menuAndItem is 0 and that if statement test fails because menuAndItem is not greater than 0. That means all the code that handles a menu selection is skipped:

If no menu selection is made, MenuSelect sets menuAndItem equal to 0.

```
case inMenuBar:
   menuAndItem = MenuSelect( TheEvent.where );

   if ( menuAndItem > 0 )
   {
       theMenu = HiWord( menuAndItem );
       theMenuItem = LoWord( menuAndItem );
       switch ( theMenu )
       {
          /* handle each menu item here */
       }
       HiliteMenu ( 0);
   }
   break;
```

The preceding figure shows what happens when the user clicks in the menu bar but doesn't end up making a menu selection. What happens if the user does make a choice? The program enters the if loop. HiWord and LoWord are called to determine the menu and menu item selected. Then a switch statement is entered. The code inside the switch (which I haven't yet shown) handles whatever tasks are expected of each menu item. Bear with me — I'll cover all that soon enough.

After the switch handles the menu selection, HiliteMenu is called. When a menu selection is made, MenuSelect highlights the name of the menu in the menu bar. The name stays highlighted even after the user releases the mouse button:



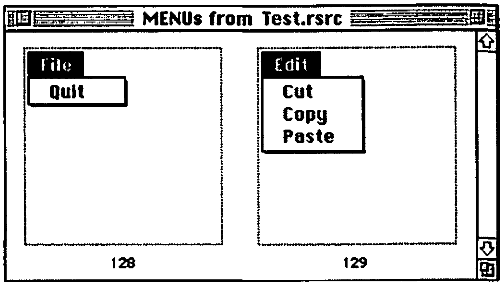
After the code that handles a menu selection is complete, the menu name returns to its normal condition. HiliteMenu is the Toolbox function that does this. Always pass HiliteMenu a value of 0 as its one parameter.

Handling a menu selection

In the previous section, I show how a menu is made to drop down and how a program can get the number of both the menu item selected and the number of the menu that holds that item. But I stopped short of showing you the details of handling the menu selection. Now I want to fill in the missing code that should be under the switch statement. I've put the switch in bold type so that you can see just where I'm about to add code:

```
case inMenuBar:
   menuAndItem = MenuSelect( theEvent.where );
   if ( menuAndItem > 0 )
   {
      theMenu = HiWord( menuAndItem );
      theMenuItem = LoWord( menuAndItem );

      switch ( theMenu )
      {
         /* handle each menu item here */
      }

      HiliteMenu( 0 );
   }
   break;
```

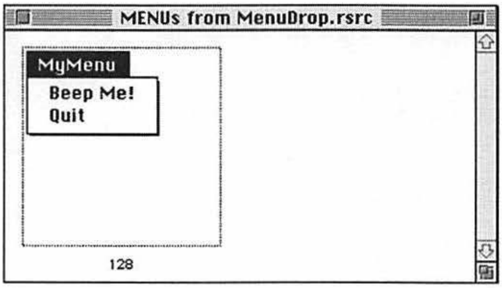
If I had a program with two menus (one with a 'MENU' resource ID of 128 and one with a 'MENU' resource ID of 129), I'd add a case label under the switch statement for each of them. Under each case label I would add more code that handled the tasks necessary for any item selection from that menu. Say that the 'MENU' resources for a program I'm writing look like this:



Then my code would look something like this:

```
switch ( theMenu )
{
   case 128:
      /* handle item 1, Quit,  from menu 128 */
   case 129:
      /* handle item 1, Cut,   from menu 129 */
      /* handle item 2, Copy,  from menu 129 */
      /* handle item 3, Paste, from menu 129 */
}
```

Fortunately, and not by any accident, I may add, the example program I've been working on in this chapter is even easier! Here's the 'MENU' resource for the example program:



So the `switch` statement for it (in general terms) looks like this:

```
switch ( theMenu )
{
   case 128:
      /* handle item 1, Beep Me!, from menu 128 */
      /* handle item 2, Quit,     from menu 128 */
}
```

Now it's time get rid of the comments and add the real code. Whenever decisions need to be made, such as which one of several menu items are to be handled, expect to see a `switch` statement. So here once again a `switch` is used.

```
switch ( theMenu )
{
   case 128:
      switch ( theMenuItem )
      {
         case 1:
            SysBeep( 1 );
            break;
```

```
        case 2:
            allDone = 1;
            break;
    }
    break;
}
```

The first `switch` uses `theMenu` to narrow down which menu was selected. In this example, there is only one — menu 128. The second `switch` uses `theMenuItem` to narrow down which menu *item* was selected. My example has two items, which are item 1 and item 2. Take a look at how each item is handled.

When the user selects the first item, Beep Me!, the code under the first `case` label runs:

```
case 1:
    SysBeep( 1 );
    break;
```

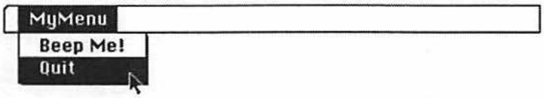
All this menu item does is sound the Mac's built-in speakers. Usually that means that the speaker emits a single beep, but other things could happen. If the user uses a control panel to change the system alert sound to something other than a beep, whatever sound they choose plays once. Or, if the user has the speaker volume set to 0, the menu bar flashes instead. `SysBeep` is the Toolbox function that does the beeping. Just pass `SysBeep` the number 1 and it gives the speaker a beep. While this menu option doesn't do anything terribly exciting, it does at least provide you with verification that the code is working. Every time the user selects this item, the speaker should beep.

Now look at how the second menu item, Quit, is handled. Even without the use of menus, you know how to end the running of a program. Just set variable `allDone` to a value of 1, and the event loop ends. That's exactly what I do here:

```
case 2:
    allDone = 1;
    break;
```

Examining a program with a menu that drops

In this section I show the code for a short program that demonstrates how all the menu handling code fits together. The program, which I call MenuDrop, uses all of the menu code that you see in this chapter, including the 'MBAR' and 'MENU' resources. A user who runs the program doesn't see a window, but instead sees one menu in the menu bar:



So, where's the Apple menu? Including the Apple menu in the menu bar of a program is a little extra work. But not so much work that I don't think you'll eventually be up to the challenge. Chapter 22 demonstrates how to add the Apple menu.

As described earlier, the first menu item beeps the Mac's speaker and the second menu item ends the program. Here's the complete source code listing for MenuDrop. You'll find the CodeWarrior project, source code file, and resource file for MenuDrop in the C18 MenuDrop folder in the *...For Dummies* Examples folder on this book's CD-ROM.

```
void  main( void )
{
   EventRecord   theEvent;
   short         allDone;
   WindowPtr     whichWindow;
   short         thePart;
   Handle        menuBarHandle;
   long          menuAndItem;
   short         theMenuItem;

   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( nil );
   FlushEvents( everyEvent, 0 );
   InitCursor();

   menuBarHandle = GetNewMBar( 128 );
   SetMenuBar( menuBarHandle );
   DrawMenuBar();

   allDone = 0;
   while ( allDone < 1 )
   {
      WaitNextEvent( everyEvent, &theEvent, 7, nil );

      switch ( theEvent.what )
      {
         case mouseDown:
           thePart = FindWindow( theEvent.where, &whichWindow );
           switch ( thePart )
           {
              case inMenuBar:
                 menuAndItem = MenuSelect( theEvent.where);
                 if ( menuAndItem > 0 )
                 {
                    theMenu = HiWord( menuAndItem );
                    theMenuItem = LoWord( menuAndItem );
                    switch ( theMenu )
                    {
                       case 128:
                          switch ( theMenuItem )
                          {
                             case 1:
                               SysBeep( 1 );
                               break;
                             case 2:
                               allDone = 1;
                               break;
                          }
                        break;
                    }
                    HiliteMenu(0);
                 }
                 break;
              }
              break;
      }
   }
}
```

The source code for MenuDrop contains no surprises — you've seen it all before. As a matter of fact, you've seen all the code you need in order to write a complete Macintosh program. And while you've done that here with MenuDrop, the result has been a program that doesn't let the user do anything except beep the Mac's speakers. In the next two chapters, you use CodeWarrior Professional or CodeWarrior Lite to use what you've learned here and apply it to the development of a still more advanced program.

Chapter 19

Writing a Very Mac-Like Program — Part I

In This Chapter

- Looking at Animator, a sample program
- Creating a resource file for Animator
- Adding resources to Animator's resource file
- Creating the CodeWarrior project for Animator
- Adding Animator's source code file to the project file

In the next two chapters, I create a Macintosh program from start to finish — without skipping a step. Developing a Mac program involves some steps that aren't source-code related and some steps that are. In this chapter, I cover the steps that don't involve writing code. Here I use CodeWarrior to create a new project to hold the source code file, resource file, and libraries used to create the program. Then I create the resource file to hold the program's resources. After creating the resource file, I use ResEdit to create the three different resource types used in the program. Finally, I create a new, empty source code file and add it to the project. After that, it's time for a short break. After a couple cans of diet soda, it's on to Chapter 20, where I take care of the steps that involve the source code itself. In Chapter 20, I type in the source code for the new program and explain just what the code does. Then I compile the source code and run it to make sure that it works. When I test-drive the code, CodeWarrior is kind enough to turn the code into a final program. That is, it builds the code and resources into a Mac application that I, or anyone I give it to, can run. After that, Animator is a fully functional program.

Now that you know what's in store, it's time to get started writing a very Mac-like program — the ever-popular Animator program.

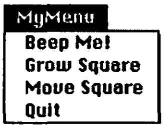
Regarding the Animator Program

In this chapter and the next, I develop a program I call Animator. You find all the files for this program in the C19 Animator and C20 Animator folders — both of which are housed in the *...For Dummies* Examples folder on this book's CD-ROM. As you read this chapter, feel free to create your own project if you own CodeWarrior (I show you how in this chapter), or open the included Animator.mcp project in the C19 Animator folder if you're using CodeWarrior Lite.

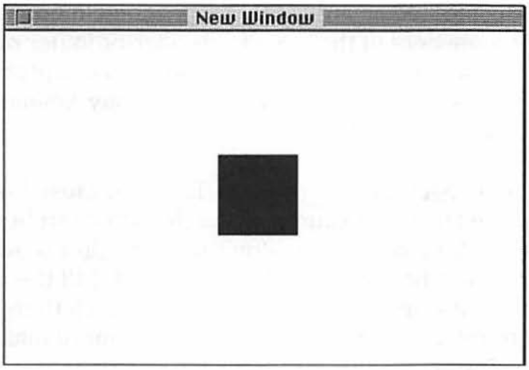
While Animator won't win the Most Useful, Exciting, and Intricate Piece of Mac Software award this year, I think you may find it's just right for you, the new Mac programmer, because in less than 100 lines of code, Animator does several of the things you want many of your own Mac programs to do. Animator, in all its glory, does all of the following:

- Initializes the Toolbox.
- Opens a window.
- Allows the user to drag the window.
- Allows the user to close the window.
- Displays a menu bar with one functional menu.
- Draws a moving shape in the window.

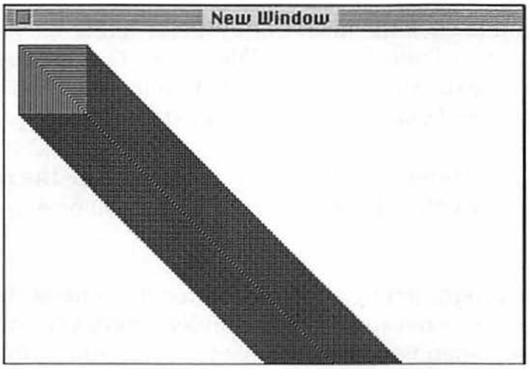
Running the Animator program displays an empty window and menu bar with one menu in it. If you click the menu, you see four items in it:



The first menu item simply sounds the Mac's speakers once. You've seen that trick before. The second menu item, Grow Square, draws a very tiny solid black square in the center of the window and then quickly enlarges it to fill most of the window. While this trick won't rival the special effects of *Star Wars,* it does give you an introduction to simple animation. I captured the square as it starts to grow — here's a look at it now:



The third menu item, Move Square, creates a small framed square in the upper-left corner of the window. The square immediately and quickly slides diagonally down the window, leaving a path of squares as it goes. Here's what the window looks like after the journey of the square is completed:



The fourth and final menu item, Quit, does just what you'd think it would do.

Assembling the Folders Needed to Create Animator

Just about every Mac program consists of a resource file, project file, and source code file. Animator is no exception. In this chapter, I use ResEdit to create the resource file, and CodeWarrior to make the project file and source code file. In the next chapter, I again use CodeWarrior to make a fourth file — the Animator application itself.

As always, I store all the files associated with my program in one folder. Next, I create a new folder somewhere in the main CodeWarrior folder on my hard disk for this purpose. Because this example accompanies Chapter 19, I name my folder C19 Animator. As I create the files needed for my Animator program, I make sure that they end up in this folder.

TIP

Keeping a program's files together is important. That's because CodeWarrior looks for the source code file and resource file so that they can be included in the project file you're working with. If you don't keep the files together in one folder, CodeWarrior may not be able to find them. Keeping all the files related to one program together also makes it easier for you to keep things organized, and to be able to refer back to these files if you want to make changes to your program in the future.

TIP

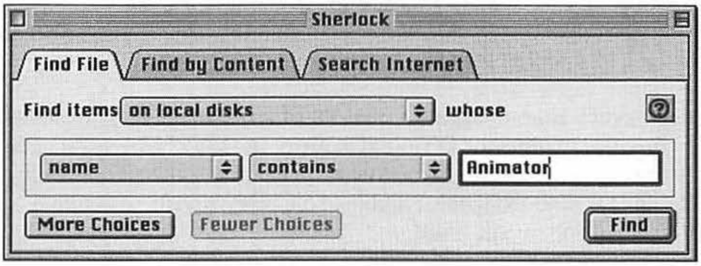
If you're using the full-featured version of CodeWarrior, you can create your own versions of the project, resource, and source code files. To do this, just follow along in the book and do as I do. If you're using the CodeWarrior Lite program that comes with this book, refer to the already completed versions of these three files in the C19 Animator folder. (You find the C19 Animator folder in the *...For Dummies* Examples folder on this book's CD-ROM.)

To start things off, double-click the main CodeWarrior folder on your hard drive and then choose New Folder from the File menu. That creates a new, untitled folder. If you give the folder the name C19 Animator, then all of your work will match the figures I use in this and the next chapter.

After you create the new folder, it won't match mine because the previous figure is a peek at what lies ahead. Instead, your folder will be empty, but not for long.

TIP

At some point in your programming, you may notice that one of the files for your program appears to be missing from the folder where you thought it should be. Don't panic! When you initially saved the file, you probably saved it into the wrong folder. That means it's on your hard drive somewhere. To find a missing file, choose Find from the File menu at the desktop. A dialog box opens up. If you're Mac is running the Mac OS 8.5 or later operating system, the dialog box looks like the one shown below. Enter part or all of the file's name and then click the Find button:



Find items on local disks whose

name contains Animator

In my example, the Mac searches for any file names that contain the word *Animator.* Notice that I have the Find window's top pop-up menu set to "on local disks." That tells the Mac to search any and all drives for the file. After searching all the drives, a list of all matching files appears. I can then double-click any file name in the list and the Mac displays that file on the desktop.

Starting the CodeWarrior Project

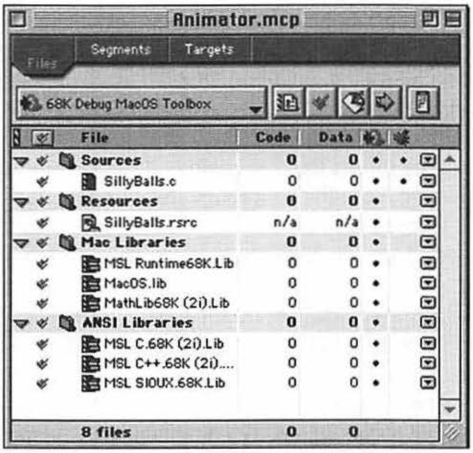
Every program starts as a project. So, of course, the Animator program does too.

If you don't have CodeWarrior running, double-click its icon to start it up. To create a new project, choose New Project from the File menu. If you're using the version of CodeWarrior that came from the CD-ROM included with this book, you have to be content with reading about what happens — CodeWarrior Lite's New Project menu item is disabled. If you own the full-featured CodeWarrior, choosing New Project from the File menu results in the display of a dialog box that allows you to choose a project stationery to be used with the new project (project stationery is described in Chapter 9).

Before clicking the OK button in this dialog box, choose a project stationery from the dialog box list. As I discuss in Chapter 9, CodeWarrior allows you to create different types of projects. For example, one type of project is used to create a program capable of running on both older and newer Macs. Another type of project is used to create a program that runs only on newer Power Macs. All the projects found on this book's CD-ROM use a project stationery that creates programs that run on both older and newer Macs. In other words, I played it safe. If you own the full-featured CodeWarrior and choose New Project from the File menu and then monkey around with the project stationery choices, you may end up with a project type that isn't right for what you're trying to do. If you dig yourself into that hole, don't click the OK button. Instead, click on the appropriate small arrow icons in the dialog box list to locate the stationery shown in the previous figure. If you've got a version of CodeWarrior that doesn't have this exact stationery type as an option, choose the option with the closest match — one with "68K" in its name.

The reason that "68K" is in the name of one of the project stationeries has to do with the processor found in older Macs — they each use a processor that is in the Motorola 68000 family. Chapter 3 has a little bit more to say about Macintosh processors. The reason I've decided to use a 68K stationery is because the program that results from it (that gets created from it) can run on older 68K Macs *and* on newer Power Macs. Conversely, a PPC stationery results in a program that can run *only* on newer Power Macs (which have a PowerPC processor).

With the appropriate stationery selected (clicking once on its name selects it), click the OK button. The dialog box disappears and a new dialog box appears. This one allows you to enter a name for the soon-to-be-created project. As expected, the dialog box list is empty — there won't be anything in the folder until you click the Save button to create a new project. First, though, I type in a name for the project. In Chapter 9, you discover that a typical project name is the name that will be given to the program, followed by a period and the letters *mcp*. I follow this naming convention by typing in **Animator.mcp** as my project's name. A click on the Save button dismisses this dialog box and brings on a new CodeWarrior project window:



File	Code	Data		
Sources	0	0	●	●
SillyBalls.c	0	0	●	●
Resources	0	0	●	
SillyBalls.rsrc	n/a	n/a	●	
Mac Libraries	0	0	●	
MSL Runtime68K.Lib	0	0	●	
MacOS.lib	0	0	●	
MathLib68K (2i).Lib	0	0	●	
ANSI Libraries	0	0	●	
MSL C.68K (2i).Lib	0	0	●	
MSL C++.68K (2i)....	0	0	●	
MSL SIOUX.68K.Lib	0	0	●	
8 files	0	0		

The file names SillyBalls.c and SillyBalls.rsrc serve as placeholders. The order in which you replace these files isn't important. Because I'm in the habit of creating my project's resource file before I create its source code file, I replace the SillyBalls.rsrc placeholder first.

Creating Animator's Resource File

I can create the resource file before my other files because I have a pretty good idea of the resources I need for the Animator program. Of course I do — I completed and tested the program long before I wrote this chapter! Even if I *really* was creating the program for the first time right here and now, I'd have a good idea of what resources I need. Why is that? Because when you set out to develop a program, you first plan out what that program is going to do. For the Animator program, I did that by making out a list of the things I wanted Animator to be able to do. Here's what I came up with:

1. Display a menu bar with a single menu in it.
2. Allow the user to beep the speakers of his or her Mac.
3. Allow the user to draw a growing square in a window.
4. Allow the user to draw a moving square in a window.
5. Allow the user to quit the program.

Looking over my list, I came to the conclusion that I need three resources. For the first point, I need an 'MBAR' resource. For points 2 through 5, I need a single 'MENU' resource — each of these points could be handled by a menu item within one menu. Additionally, points 3 and 4 require a window to be on the screen, so I also need a 'WIND' resource.

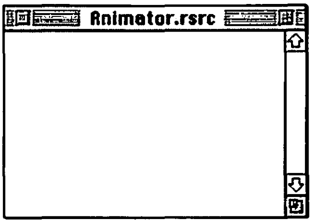
Creating the resource file

Begin by running ResEdit. If you're using CodeWarrior and your own project, feel free to double-click the SillyBalls.rsrc placeholder in the CodeWarrior project window to open ResEdit. If you're using CodeWarrior Lite and my own Animator.mcp project from the Chapter 19 folder of the CD-ROM, you can't use this trick. That's because I already created the resource file and added it to the project for you! If you're using CodeWarrior Lite but you still want the practice of creating a resource file, please do so. You can run ResEdit by double-clicking the icon of the ResEdit program or by double-clicking on the Animator.rsrc name in the project window.

In ResEdit, choose New from the File menu to create your new resource file. If you're using CodeWarrior Lite, consider this a practice file (CodeWarrior Lite won't let you add the file to the project). Give the file any old name you want. Because this is just practice, and you won't be adding the file to a project, it doesn't matter what you name the file or where it ends up on your hard drive! If you are just creating a practice resource file, go ahead and click the New button now.

If you're using the full-featured version of CodeWarrior rather than CodeWarrior Lite, you can consider this file the real McCoy — the resource file you'll eventually be adding to your own project. If that's the case, move to the folder that holds the project file (if you haven't created your project file yet, see the previous section). Use the pop-up menu that appears above the scrollable list to move about from folder to folder. Now, type in the resource file name. In the previous section, I created a CodeWarrior project and named it Animator.mcp. If I want to go with the naming convention used by most Mac programmers, I should name the resource file Animator.rsrc. If you're following along with me, type in the resource file name and then click the New button.

Whether you're a CodeWarrior Lite user who's going to work on a practice resource file or a CodeWarrior user who wants to create a resource file to add to your own project, you're now at the same point. After clicking the New button, the dialog box disappears and a new, empty type picker appears. The resource file is created, and it's ready for the addition of some resources.

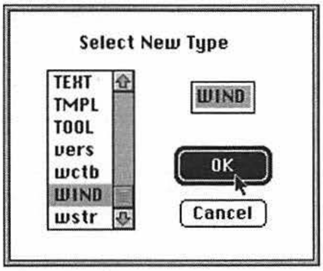


I went through that pretty fast. Just in case you missed something, here's a quick summary of the steps for creating a new resource file:

1. **Run ResEdit using one of the previously mentioned techniques (such as by double-clicking the SillyBalls.rsrc placeholder in the project window of a new project, or by double-clicking the Animator.rsrc name in the project window of the CD-ROM's Animator project, or by double-clicking the ResEdit icon from the Mac desktop).**
2. **Click the mouse to dismiss the ResEdit Jack-in-the-Mac introductory screen (if it appears).**
3. **Choose New from the File menu.**
4. **In the dialog box that appears, move to Animator's project folder.**
5. **Type in the resource file's name.**
6. **Click the New button.**

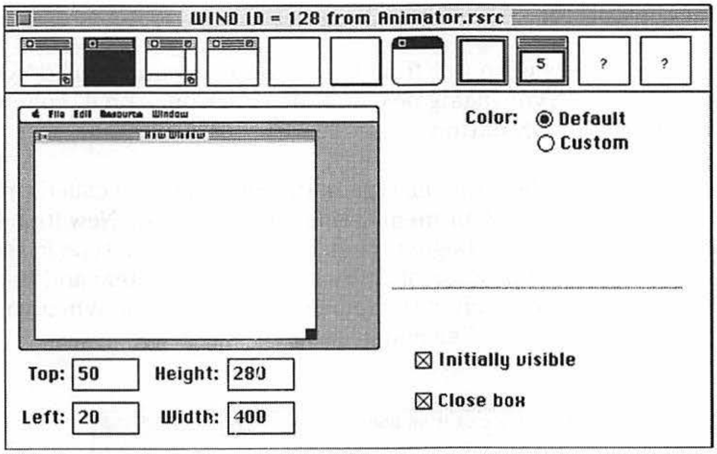
Adding the Window resource

The Animator program opens a window, so you know a 'WIND' resource is needed. Choose Create New Resource from ResEdit's Resource menu. The Select New Type dialog box opens. Scroll down to WIND and then click it once. Then click the OK button:



WIND

Dismissing the Select New Type dialog box opens a 'WIND' editor, as shown in the following figure. Because the 'WIND' editor initially displays a rather small window, I change the size of my 'WIND' by typing in the numbers shown here in the lower left of the screen:



Color: ● Default ○ Custom

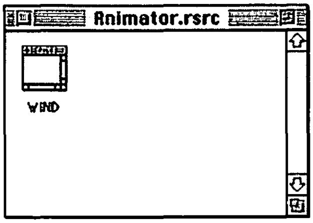
Top: 50 Height: 280

Left: 20 Width: 400

☒ Initially visible

☒ Close box

The only other change I make here is to the window type. I give the second icon from the left a click in the row of window icons at the top of the editor. That sets up my 'WIND' for a window that has a title bar and a close box, but no grow box in the lower-right corner. (If you aren't familiar with the term *grow box*, I don't want to hold you in suspense: A grow box is the small box that appears in the lower-right corner of any window that can be resized.) When I click the close box of the 'WIND' editor, here's what I see:

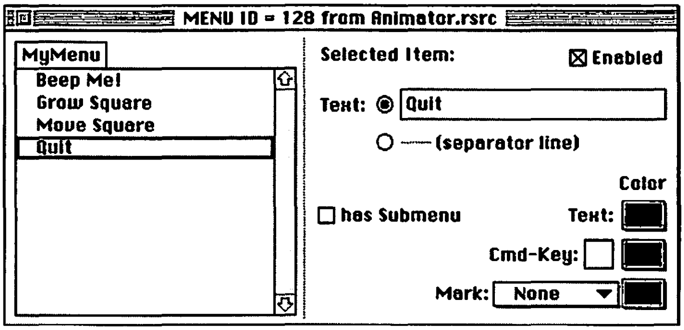


Adding the Menu resources

From my description of the Animator program at the beginning of this chapter, you should have a pretty good idea about what other resources are needed to create Animator. The Animator has one menu, and so it needs one 'MENU' resource. The menu appears, of course, in the menu bar, indicating that I also need an 'MBAR' resource for that. First comes the 'MENU' resource.

Choose Create New Resource from the Resource menu. Scroll down to MENU in the Select New Type dialog box, and then click once on it. Follow that click with a click on the OK button.

When the 'MENU' editor opens, type in the menu's title. I called the menu MyMenu. Next, add the four menu items. Choose Create New Item from the Resource menu and then begin typing. For the first item, type in the words **Beep Me!** Repeat the process of choosing Create New Item and then typing in a menu item name for each of the remaining menu items. When you're done, your 'MENU' should look like mine:



MENU ID = 128 from Animator.rsrc

MyMenu
Beep Me!
Grow Square
Move Square
Quit

Selected Item: ☒ Enabled

Text: ● Quit

○ —— (separator line)

□ has Submenu

Color

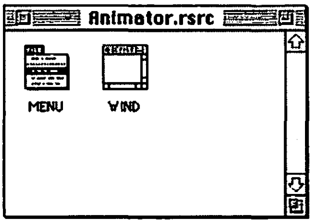
Text: ■

Cmd-Key: □ ■

Mark: None ▼ ■

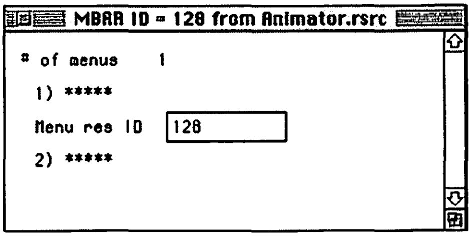
Close the 'MENU' editor. When you do that, you see another view of the 'MENU' resource. This view gives you a pretty good idea of how the menu looks once it's added to the Animator program.

After you're done admiring the new 'MENU' resource, click the window's close box to bring the type picker to the front. Now two different resource types appear in the type picker:



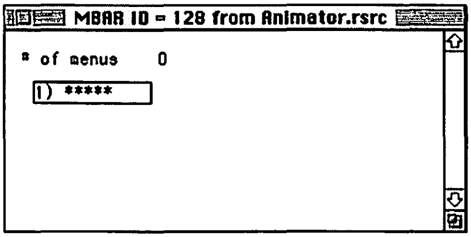
Now you have one last resource to create. Again, choose Create New Resource from the Resource menu. Scroll down to MBAR, click once on it, and then click the OK button.

Eventually, you want the Animator program to display a single menu, and so the resource file for Animator has a single 'MENU' resource. Now is the time to add it to the list of 'MENU' resources that the 'MBAR' keeps. I've added it in the following figure:



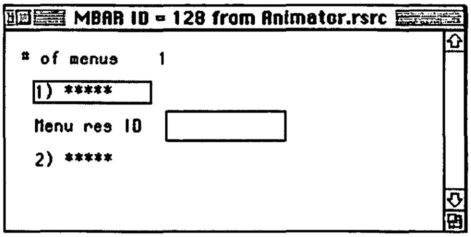
# of menus 1
1) *****
Menu res ID 128
2) *****

Do you know how to add a 'MENU' to the 'MBAR' resource? If not, a few tips on the subject may come in handy. To add a 'MENU' to the 'MBAR' resource, first click once on the row of five stars in the 'MBAR' editor. A rectangle appears around the stars, like this:



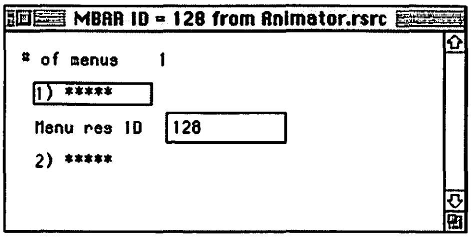
```
MBAR ID = 128 from Animator.rsrc

# of menus    0
1) *****
```

Next, choose Insert New Field(s) from the Resource menu. Here's what you see:



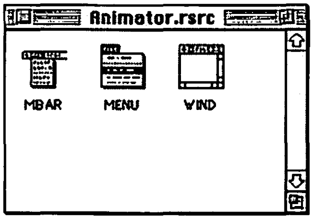
```
MBAR ID = 128 from Animator.rsrc

# of menus    1
1) *****
Menu res ID
2) *****
```

Type the resource ID of the 'MENU' resource into the edit box that's been added to the 'MBAR'. This resource file has only one 'MENU' resource in it (remember that resource numbers start at 128), and so the 'MBAR' is complete. Here's how it looks:



```
MBAR ID = 128 from Animator.rsrc

# of menus    1
1) *****
Menu res ID   128
2) *****
```

Click the close box of the 'MBAR' editor, and you see that the type picker now looks like this:



The resource file for the Animator program is now complete. Choose Save from the File menu and then choose Quit from the File menu to quit ResEdit.

Adding the resource file to the project

After you complete the Animator.rsrc resource file, it's time for you to add it to the Animator.mcp project. (Chapter 9 shows this to be a simple task.) If you're using CodeWarrior Lite, you can look at the Animator.mcp project on this book's CD-ROM to see that this has been done for you. If you're using the full-featured CodeWarrior, follow these steps to do it yourself:

1. **Click the Animator.mcp project window to make ResEdit inactive and CodeWarrior active.**
2. **Click once on the SillyBalls.rsrc name in the project window.**
3. **Choose Add Files from the Project menu.**
4. **If the upper list in the dialog box isn't displaying the contents of your project folder, use the pop-up menu at the top of the dialog box to move into that folder.**
5. **Click once on the Animator.rsrc file name in the top list.**
6. **Click the Add button.**
7. **Click the Done button.**

After performing these steps, the Animator.rsrc file name appears in the project window right beneath the SillyBalls.rsrc placeholder. Now it's time to get rid of that placeholder. To do that, first click once on its name to highlight it. Then choose Remove Files from the Project menu.

Your Animator.mcp project is shaping up quite nicely. Now it's on to the source code file.

Creating Animator's Source Code File

Animator requires a second file — the source code file. If you're following along in the Animator.mcp project that I supply on the CD-ROM that comes with this book, you can see that this file has been added to the project. If you're working with the full-featured version of CodeWarrior and working from your own project, then you know that it doesn't exist yet. To create it, choose New from the File menu of CodeWarrior. When you do, an empty window opens. Before working on the code, choose Save As from the File menu. When prompted to enter a name, type in **Animator.c**, but don't save the file just yet.

You want to make sure the source code file gets saved to the same folder that holds your Animator.mcp project. If the pop-up menu at the top of the dialog box isn't displaying the name of that folder, use that pop-up menu to move to the correct folder. Then and only then should you click the Save button to save the file.

Though you can give your source code file any name you want, it makes sense to give it a name that associates it with the program you're creating. I name my source code file Animator.c. Any name is all right, but make sure it ends with a period followed by the letter *c*.

You don't need to type in all the source code before adding the file to your project. In fact, it makes sense to add the file to the project right away so that you don't forget to do it. Here are the steps you take to add this file to your project:

1. **Click once on the SillyBalls.c name in the project window.**
2. **Click once on the new source code file window to make it active (to make it the frontmost window on your screen).**
3. **Choose Add Window from the Project menu.**

After performing these steps, the Animator.c file name appears in the project window. Now, remove the SillyBalls.c placeholder. First click once on its name to highlight it. Then choose Remove Files from the Project menu.

Is That It?

Now you're just about done. That is, except for that little part about writing the source code! I've saved that for the next chapter.

Chapter 20

Writing a Very Mac-Like Program — Part II

In This Chapter

- ▶ Viewing the complete source code for the Animator program
- ▶ Getting a description of all the major elements of the program
- ▶ Creating simple animated effects
- ▶ Compiling and running source code
- ▶ Naming your new application

In Chapter 19, I — and hopefully you — created a resource file, a project file, and a source code file for a program I call the Animator. In this chapter, you get a look at what the source code for Animator does. After hearing about what the program should do, you see for yourself by compiling and running the code. When you do that, CodeWarrior creates a standalone program for you. That is, CodeWarrior turns your code into a Mac application that you, I, or anyone else can run.

Introducing the Animator Source Code

For the next few pages, I present the complete source code listing for the Animator program. Just under 100 lines of code make up the listing. Personally, I don't consider that a lot of code. But then, I just read that the software that would run the Star Wars defense system proposed by former President Ronald Reagan would have required 100 *million* lines of code. That should make you feel better about the 100 lines you're about to view. If it doesn't, take a look at this breakdown of the 100 lines of code. It may take away any apprehensions you have.

- About 20 of those lines are just braces — that leaves about 80 lines.
- About 10 lines are variable declarations — that leaves about 70 lines.
- A little less than 10 lines make up the standard Toolbox initialization code that appears in every program — that leaves about 60 lines.
- You know that `switch` statements group code together and separate those groups with `case` labels and `break` statements. A few `switch` statements have a total of close to 10 `case` labels and 10 `break` statements — one `break` to end each `case`. That's over 20 lines of code, which leaves about 40 lines.
- In Chapter 18, you saw how to display a menu bar and how to figure out which menu was clicked. You also saw how to work with a window. There are about 15 lines of code devoted to these two tasks. That leaves about 25 lines.

Twenty-five lines of code? No problem! That's right — after discounting the lines of code that you've had plenty of experience with, there are only about 25 lines of code that can't be neatly pegged into a basic programming category. The Animator source code contains very little new code — it mostly demonstrates how to tie together all the topics I've already covered. Don't let the fact that there's not a lot of new stuff fool you into thinking that this chapter isn't important, though. Turning all the bits and pieces into a unified program will prove to be a mighty accomplishment!

Viewing the glory of the Animator code

I list all the source code for the Animator program in this section in a nice, easy-to-look-at, uninterrupted form. Then I devote the next section of this chapter to an in-depth description of what's going on in the code.

ON THE CD

If you're using the full-featured version of CodeWarrior, carry on with the project you started in Chapter 19. You can try your hand at typing in all of the code listed below, or you can take the easy way out and copy the code from the Animator.c file found in the C20 Animator folder on the CD-ROM and paste it in your own empty Animator.c file. If you're using CodeWarrior Lite, you can simply open the C20 Animator folder and double-click the Animator.mcp project. The Animator.c file that is a part of this project has all the code typed in for you.

```
void  main( void )
{                                  /* declare the variables */
  WindowPtr     theWindow;         /* the window */
  EventRecord   theEvent;          /* event record */
  short         allDone;           /* are we done yet? 0=no, 1=yes */
  WindowPtr     whichWindow;       /* for use by FindWindow */
  short         thePart;           /* part code for FindWindow */
  Handle        menuBarHandle;     /* used during menu bar set up */
  long          menuAndItem;       /* holds selected menu and item */
```

```
short         theMenu;        /* holds just the selected menu */
short         theMenuItem;    /* holds just the selected item */
Rect          theRect;        /* used to draw some squares */
short         count;          /* loop counter */
unsigned long theLong;        /* used when delaying drawing */

InitGraf( &qd.thePort );
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();

menuBarHandle = GetNewMBar( 128 );
SetMenuBar( menuBarHandle );
DrawMenuBar();

theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );

allDone = 0;
while ( allDone < 1 )
{
  WaitNextEvent( everyEvent, &theEvent, 7, nil );
  switch ( theEvent.what )
  {
    case mouseDown:
      thePart = FindWindow( theEvent.where, &whichWindow );
      switch ( thePart )
      {
        case inDrag:
          DragWindow( whichWindow, theEvent.where,
                      &qd.screenBits.bounds );
          break;

        case inGoAway:
          DisposeWindow( whichWindow );
          allDone = 1;
          break;

        case inMenuBar:
          menuAndItem = MenuSelect( theEvent.where );
          if ( menuAndItem > 0 )
          {
            theMenu = HiWord( menuAndItem );
            theMenuItem = LoWord( menuAndItem );
            switch ( theMenu )
            {
              case 128:
                switch ( theMenuItem )
                {
                  case 1:
                    SysBeep( 1 );
                    break;

                  case 2:
                    SetRect( &theRect, 0, 0, 400, 280 );
                    EraseRect( &theRect );
```

(continued)

(continued)

```
                      SetRect( &theRect, 200, 135, 200, 135 );
                      count = 0;
                      while ( count < 120 )
                      {
                        FillRect( &theRect, &qd.black );
                        InsetRect( &theRect, -1, -1 );
                        count++;
                        Delay( 2, &theLong );
                      }
                      break;

                    case 3:
                      SetRect( &theRect, 0, 0, 400, 280 );
                      EraseRect( &theRect );
                      SetRect( &theRect, 10, 10, 60, 60 );
                      count = 0;
                      while ( count < 140 )
                      {
                        FrameRect( &theRect );
                        OffsetRect( &theRect, 2, 2 );
                        count++;
                        Delay( 2, &theLong );
                      }
                      break;

                    case 4:
                      allDone = 1;
                      break;
                  }
                  break;
              }
              HiliteMenu( 0 );
            }
            break;
        }
        break;
    }
  }
}
```

Does much of this code look familiar? It should. You've seen some of it in example programs throughout this book — programs such as MyProgram, WindowWorks, and MenuDrop. You didn't *really* think that every time someone wrote a program they started completely from scratch, did you? Most programmers do like a good challenge, but they don't like repetitively typing in the same code.

TIP

When using CodeWarrior Professional, you don't have to start your source code completely from scratch for each new project. Instead, you can first create a new, empty source code file and add it to your new project — just as you've seen done in this book. Then you can choose the Open command from the CodeWarrior File menu and open an *existing* source code file — one that has at least some code similar to code you plan on writing. Copy any or all of the code from the older file and paste it into the newer, empty source code file of your new project — and then modify it. If you've typed in the example programs presented in this book, I hereby grant you permission to copy any of them and modify them. Heck, I didn't invent most of the code anyway. It's similar to Macintosh source code that thousands of other programmers are using!

Knowing What's Going On in the Code

The Animator program ties together all the individual concepts covered in this book — and that makes it worthy of a good long look. Ready? Here goes!

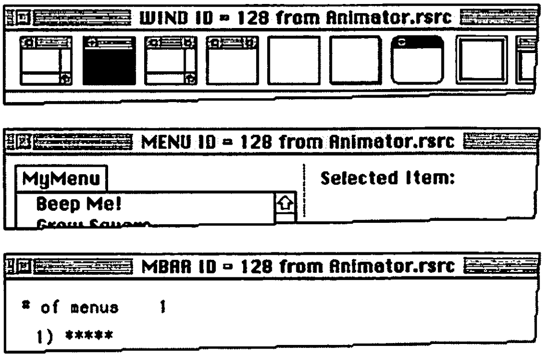
Finding out about 128

It would make the most sense if I describe the code starting at the first line and work my way to the end, don't you agree? That's what I'll do — after a very short diversion.

If you look over the source code for the Animator, you notice that it contains the number 128 in three separate locations. I plucked that code out of the listing, and here it is:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );

menuBarHandle = GetNewMBar( 128 );

switch ( theMenu )
{
   case 128:
```

Why the number 128? As a reminder of the importance of this number, I ran ResEdit and opened the editor for each of the program's three resources. In the following figure I show the title bar from each. Note the ID listed in each title bar.



WIND ID = 128 from Animator.rsrc

MENU ID = 128 from Animator.rsrc

MyMenu

Beep Me!

Selected Item:

MBAR ID = 128 from Animator.rsrc

# of menus 1

1) *****

Each resource has an ID of 128. Coincidence? Not at all. When ResEdit creates a new resource, it usually gives it an ID of 128. This isn't true of all resource types, but it is for 'WIND', 'MENU', and 'MBAR' resources. After the first resource of a type is created, the next resource of that same type is given the number 129. If I added a second 'MENU' to my program, ResEdit would give it an ID of 129.

Why the number 128? Why doesn't ResEdit start with number 1? Apple has reserved the numbers up to 127 for its own use. The Macintosh has a set of resources hidden from your view that it uses to display things like the trash can icon and the menu bar you see when you're at the desktop. It is possible to renumber a resource of yours to give it a number less than 128, but Apple strongly suggests that you don't.

Declaring variables

All Mac programs written in C begin with `void  main( void )` and an opening brace. They end with a closing brace. What does that have to do with variable declarations? Nothing. Because there's not a whole lot of explaining to accompany the fact that all programs have a `main` function, I didn't think that this concept needed its own section. Now, on to the declarations.

Animator uses 12 variables, which I briefly describe here. (I give more information about each variable as I encounter it later in the code.) Here are the declarations followed by the promised descriptions:

```
WindowPtr     theWindow;      /* the window */
EventRecord   theEvent;       /* event record */
short         allDone;        /* are we done yet? 0=no, 1=yes */
WindowPtr     whichWindow;    /* for use by FindWindow */
short         thePart;        /* part code for FindWindow */
Handle        menuBarHandle;  /* used during menu bar set up */
long          menuAndItem;    /* holds selected menu and item */
short         theMenu;        /* holds just the selected menu */
short         theMenuItem;    /* holds just the selected item */
Rect          theRect;        /* used to draw some squares */
short         count;          /* loop counter */
unsigned long theLong;        /* used when delaying drawing */
```

Animator opens a single window. To keep track of that single window, I declare a `WindowPtr` variable called `theWindow`. Like any good Mac program, Animator takes note of events and responds to them. The variable `theEvent` holds information about the most recent event. Animator is a great program, but users of it will probably want to quit at some point. The variable `allDone` helps out there.

A click of the mouse button constitutes an event. When that happens, the Animator wants more information about the event. The variable thePart holds the part of the screen at which the mouse click occurred. If the click occurred in a window, variable whichWindow holds a WindowPtr to that window — useful information later in the program.

The Animator has one menu. The variable menuBarHandle is used in the setup of the menu bar that holds this menu. When the user makes a menu selection, variable menuAndItem holds the key to which menu item was selected. As a combined value of menu and menu item, menuAndItem is of limited use. So variables theMenu and theMenuItem hold the individual information that is found in a combined form in menuAndItem.

What's the Mac without a little bit of drawing going on — right? The Animator uses a Rect variable called theRect to hold the screen coordinates of a rectangle. That same rectangle variable is used twice — once to make the rectangle grow and a second time to slide it across the window. Moving things in a window involves looping, and a loop requires a variable to keep track of when the loop should end. That's the purpose of the short variable called count. Because the Mac is no slouch of a computer, it runs through the code that makes up a loop pretty darn fast. Too fast, sometimes. The Toolbox provides an easy means of slowing down a loop. You see how that's done, and how the variable theLong plays a part in this delay, a little later in this chapter.

So much for declaring variables. Now it's on to initializing the Toolbox.

Initializing the Toolbox

You may have seen the Toolbox initialization code so many times that you never want to see it again. But for the sake of completeness, here it is:

```
InitGraf( &qd.thePort );     /* standard initializations */
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();
```

I said it before, and I say it again: To avoid serious problems in your program, include these eight lines of code right after your variable declarations in every program you write. You, and the Toolbox, will be glad that you did.

Displaying menus and windows

Working with menus and windows is fun — but first you have to get them on the screen. These five lines display a menu bar and one window:

```
menuBarHandle = GetNewMBar( 128 );
SetMenuBar( menuBarHandle );
DrawMenuBar();
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );
```

The first three lines make the Mac aware of a new menu bar and then display it. The fourth and fifth lines open a new window and tell the Mac that subsequent drawings should take place in it.

Establishing the event loop

The remainder of the program is the event loop. The variable allDone is set to 0, and then the while statement that begins the loop appears. allDone is obviously less than 1, so the code under the while statement runs. A call to WaitNextEvent asks the Toolbox to place information about the most recent event into theEvent. Next, a switch statement watches to see whether the event is worthy of a response. Animator only cares about one type of event — a click of the mouse button. By excluding all other event types from the switch statement, the program effectively filters them out. That is, other events, such as a press of a key, are simply ignored. Here's the event loop code — without the numerous lines that fall under the case mouseDown. I cover those lines in separate sections of this chapter.

```
allDone = 0;
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   switch ( theEvent.what )
   {
      case mouseDown:
         /* a bunch of mouse click code here! */
         break;
   }
}
```

Handling a mouseDown event

If the user clicks the mouse button, WaitNextEvent places a value of mouseDown in theEvent.what. The very next line, the switch statement, then matches the value of theEvent.what to the case mouseDown label, and the code under the case runs.

FindWindow determines just where the mouse click occurred. It places a value in variable thePart — a value that represents the part of the screen at which the click took place. A switch statement then examines variable thePart and determines which set of code should run to handle the mouse click.

A click in the drag bar of the window results in a call to DragWindow. This Toolbox function takes control and handles window movement for you.

A click in the close box of the window brings about a call to DisposeWindow. This function disposes of the window — that is, it removes it from the screen. Variable allDone is then set to a value of 1. When the program returns to the top of the event loop, this value of allDone signals the program to end.

Most programs don't end when the user closes a window. But for the simple Animator program, it wouldn't make much sense to continue if the user closes the window. After all, where would drawing then take place?

A click in the menu bar causes the code under case inMenuBar to run. I omit that code here because it represents a significant portion of the program. As such, it gets several pages devoted to it later in the chapter.

```
switch ( theEvent.what )
{
   case mouseDown:
      thePart = FindWindow( theEvent.where, &whichWindow );
      switch ( thePart )
      {
         case inDrag:
            DragWindow( whichWindow, theEvent.where,
                        &qd.screenBits.bounds );
            break;

         case inGoAway:
            DisposeWindow( whichWindow );
            allDone = 1;
            break;

         case inMenuBar:
            /* a bunch of menu-handling code here! */
            break;
      }
      break;
}
```

Handling a click in the menu bar

A mouse click in the menu bar is what causes the real action to take place. When that all-important mouse click happens, the Toolbox function MenuSelect is called to handle the display of the program's one menu. If a menu selection is made, MenuSelect places a value in variable menuAndItem — a value that represents the menu number and the menu item number. A menu selection always gives menuAndItem a value greater than 0. That nonzero value is what sends the code into the if branch that follows the call to MenuSelect.

The Toolbox functions HiWord and LoWord do the grunt work of separating both the menu number and the menu item number from variable menuAndItem. That first bit of information — the menu number — is then used in a switch statement to determine which menu to work with. Animator only has one menu — the 'MENU' resource with ID 128 — but many programs have more than one.

Knowing the menu that houses the selected menu item isn't enough information to determine how to respond to the selection. The program also needs to know which menu item within that menu is selected. The switch statement that uses variable theMenuItem handles that. In the next several pages, I describe how each menu item is handled. In the next batch of code, I replace the code for the menu items with comments in order to give you the overall feel for what's going on.

```
case inMenuBar:
   menuAndItem = MenuSelect( theEvent.where );
   if ( menuAndItem > 0 )
   {
      theMenu = HiWord( menuAndItem );
      theMenuItem = LoWord( menuAndItem );
      switch ( theMenu )
      {
         case 128:
            switch ( theMenuItem )
            {
               case 1:
                  /* handle Beep Me! item */
                  break;
               case 2:
                  /* handle Grow Square item */
                  break;
               case 3:
                  /* handle Move Square item */
                  break;
               case 4:
                  /* handle Quit item */
                  break;
            }
            break;
      }
      HiliteMenu( 0 );
   }
   break;
```

After the menu selection is handled, HiliteMenu is called to return the menu name to its original state. (The menu name is highlighted — and left that way — when a menu item is selected.)

Adding the Beep Me! item

I devote a separate section of this chapter to the code that handles each of the four menu items — no matter how few lines of code are necessary to do it! The first menu item, Beep Me!, simply uses the Toolbox function SysBeep to play the system alert sound.

```
switch ( theMenuItem )
{
   case 1:
      SysBeep( 1 );
      break;
   /* other case sections go here */
}
```

Nurturing the Grow Square item

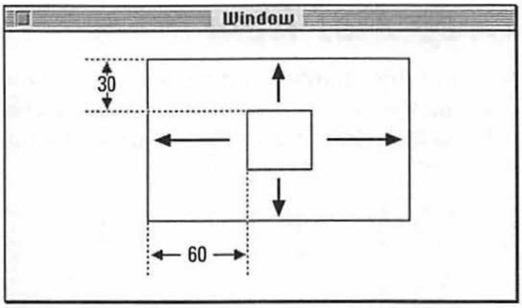
You know about a couple of the Toolbox functions that work with rectangles. In this chapter, you're about to see a few more.

The second menu item, Grow Square, uses a while loop to repeatedly draw a square. That wouldn't have the effect of growing the square unless I first enlarged the square a little bit each time the loop ran. That's exactly what the Toolbox function InsetRect does. You tell InsetRect the rectangle you want to shrink or expand, and by how much, and InsetRect does the work. If you pass InsetRect positive numbers, the rectangle gets smaller. Negative numbers make the rectangle get larger. Consider this example, which *isn't* from the Animator program:

```
SetRect( &theRect, 150, 50, 200, 100 );
FrameRect( &theRect );
InsetRect( &theRect, -60, -30 );
FrameRect( &theRect );
```

While I describe what the preceding code does, you should follow along in the next figure. First, a rectangle 50 pixels wide and 50 pixels high is set and then framed. That's the smaller rectangle in the center of the window in the figure. Next, a call to InsetRect is made. The value of –60 means the rectangle should become 60 pixels *larger* in each of the horizontal directions. The value of –30 means the rectangle should become 30 pixels larger in each of the vertical directions. InsetRect changes the size of a rectangle, but, like SetRect, it doesn't draw it. So I follow the call to InsetRect with another call to FrameRect. The result is the larger of the two rectangles in the figure.

If it seems odd that negative numbers make the rectangle bigger rather than smaller, think of the name of the Toolbox function — InsetRect. InsetRect is normally used to inset a rectangle — to make a rectangle smaller so that it fits inside another. The parameters to InsetRect tell the Toolbox how much smaller to make the rectangle. Because I want the opposite effect, I use negative numbers.



Now, back to the `while` loop. Actually, let me back up a little further still. The first thing that happens under the `case` is a call to `SetRect` and `EraseRect`. `EraseRect` is a Toolbox function that does pretty much what its name implies. Pass `EraseRect` a rectangle variable and this function clears everything that's drawn anywhere in that rectangle.

Now, take note of the size of the rectangle I pass to `EraseRect`. If you compare the size of this rectangle (400 pixels wide by 280 pixels high) to the window size as defined in the 'WIND' resource, you see that the two just happen to match. Mere coincidence? Of course not! By making the rectangle the same size as the window, I'm using `EraseRect` to white out, or erase, anything that was in the window. Why take this step at the start of the code for handling this menu selection? The user may have already made this menu selection, or another drawing selection, beforehand. I'm getting rid of any leftover drawing that may still be in the window.

Make a note of this technique for clearing a window:

```
SetRect( &theRect, 0, 0, 400, 280 );  /* Size of the window */
EraseRect( &theRect );                /* White it out       */
```

Now for the drawing. After whiting out the window, I again call `SetRect`. This time I create a rectangle that is almost centered in the window. But notice its size:

```
SetRect( &theRect, 200, 135, 200, 135 );   /* tiny!     */
```

The left and right sides have the same value (200). And so do the top and bottom sides (135). That means that the rectangle won't even show up when I draw it, but that will soon change. Next comes the `while` loop. I've set variable `count` to a value of 0, and `while` to `count < 120`, so my loop runs 120 times. Each time through the loop, the rectangle is filled in with black, then made larger by one pixel in each direction. The result? A solid black rectangle appears to grow from nothing to a size that almost fills the window. Here's the code that makes it all happen:

```
switch ( theMenuItem )
{
   /* other case section goes here */
   case 2:
      SetRect( &theRect, 0, 0, 400, 280 );
      EraseRect( &theRect, &white );
      SetRect( &theRect, 200, 135, 200, 135 );
      count = 0;
      while ( count < 120 )
      {
         FillRect( &theRect, &qd.black );
         InsetRect( &theRect, -1, -1 );
         count++;
         Delay( 2, &theLong );
      }
      break;
   /* other case sections go here */
}
```

The preceding code snippet contains one line that hasn't been discussed — the one that includes the call to a function named `Delay`. `Delay` is a Toolbox function that you can use to add a delay, or pause, of just about any length to your program. The first of the two parameters is the length of the delay. Your first guess may be that this number is the number of seconds to hold things up, but this isn't the case. Sometimes you want to delay things for a time much less than a second, so the `Delay` function gives you that option. It turns out that the first parameter in `Delay` is the number of sixtieths of a second to delay things. That means it requires a value of 60 here to delay things a full second. The following code displays several examples of the use of delay, along with comments that describe the delay that results from each:

```
Delay( 1, &theLong );     /* pause 1/60th of a second */
Delay( 2, &theLong );     /* pause 2/60th of a second */
Delay( 30, &theLong );    /* pause one half of a second */
Delay( 60, &theLong );    /* pause one second */
Delay( 300, &theLong );   /* pause five seconds */
```

From the preceding code, you can see that in Animator, I'm using `Delay` to generate a delay that is 2/60 of a second — at each pass through the `while` loop. That's not a very long time, but it does visibly slow down the growing of the rectangle.

Now, what about that second parameter? At the start of the Animator source code listing, I declared a variable named `theLong` that was of type `long`:

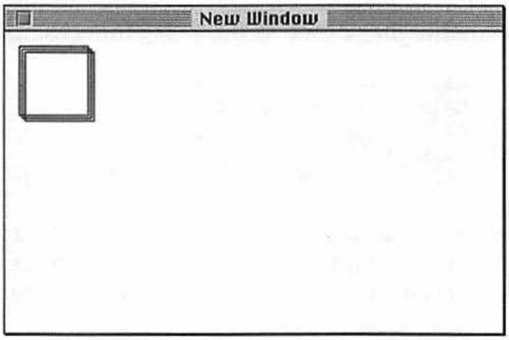
```
long   theLong;
```

Here's where that variable gets used. The second parameter to `Delay` is an `unsigned long` variable, preceded by an ampersand. I won't go into the ugly details of what this parameter is used for, but I will say this: Its value is unimportant. You don't have to understand just what an `unsigned long` is (in essence it's a data type used to hold very large numbers), and you don't have to assign theLong any value — just use it as I've done in the preceding examples. The Toolbox knows what to do in these situations.

Dodging the Move Square item

The third menu item, Move Square, works in a manner very similar to the growing square. First the window is cleared — just in case the big black rectangle from a selection of the Grow Square menu item is sitting in it. Next, a 50-x-50 pixel square is set up in the upper-left corner of the window. Then a while loop runs 140 times. And for the grand finale, take a look at what the while loop does.

In each pass through the loop the rectangle is drawn and then offset. Careful — *offset* is different than *inset*. The Toolbox function OffsetRect doesn't change the size of a rectangle; it moves it. The amount the rectangle gets moved, or offset from its current position, is specified in the last two parameters. A positive value for the second parameter moves the rectangle to the right; a negative value moves it to the left. A positive value for the third parameter moves the rectangle down, a negative value moves it up. My call to OffsetRect moves the rectangle right 2 pixels and down 2 pixels. The resulting window looks like this after three passes through the loop:



You can see from the preceding figure that the square continues to move to the right and down until it goes off the bottom of the window. The following code moves the square:

```
switch ( theMenuItem )
{
   /* other case sections go here */
   case 3:
      SetRect( &theRect, 0, 0, 400, 280 );
      EraseRect( &theRect );
      SetRect( &theRect, 10, 10, 60, 60 );
      count = 0;
      while ( count < 140 )
      {
         FrameRect( &theRect );
         OffsetRect( &theRect, 2, 2 );
         count++;
         Delay( 2, &theLong );
      }
      break;
   /* other case sections go here */
}
```

Finishing up with the Quit item

Here's another simple menu item — the Quit item. Choosing this fourth menu item simply sets variable `allDone` to a value of 1. When the program reaches the top of the event loop, the `while` test fails and the program ends.

```
switch ( theMenuItem )
{
   /* other case sections go here */
   case 4:
      allDone = 1;
      break;
}
```

Compiling and Running the Animator Program

If you've opted to type in all the Animator source code yourself, make sure to choose Save from the File menu. Then it's time to compile the code. Choose Compile from the Project menu.

Remember, if the Compile option appears dim, open the source code file — Animator.c — by double-clicking on its name in the project window.

If the Errors & Warnings window opens and reports an error (or errors), make the necessary corrections and try again. If you can't figure out what an error message means, try referring to Appendix C, where I list possible solutions for problems that you may encounter while compiling a program.

When your source code compiles successfully, it's time to give it a test run. Choose Run from the Project menu.

Remember, you can skip the Compile menu item altogether by simply choosing Run from the Project menu. The Run menu item compiles the code if it needs to be compiled.

Running the code without the resource file or with an incomplete resource file either causes the program to start and then quickly quit, or, worse yet, crash the Mac. If your Mac crashes, you then have to restart your computer and restart CodeWarrior. So don't forget to make the resource file and add it to the project. Without it, and the 'WIND', 'MENU', and 'MBAR' resources it holds, the program can't run.

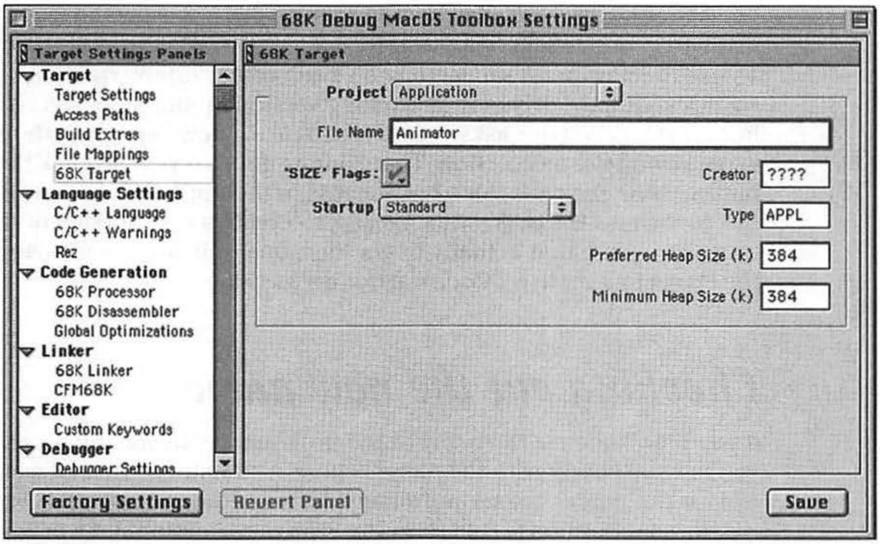
If everything is going according to plan, Animator should be off and running. Try each of the first three menu items a few times to verify that everything's working. When you're satisfied that the program works, choose the fourth menu item, Quit.

Naming the Application

Choosing Run from the Project menu allows you to test out the Animator program. It also tells CodeWarrior to turn the source code into an application. When it does that, what name does CodeWarrior assign to your program? Here's a bit of good news — you're in control of that option.

Stating your preference

CodeWarrior is *configurable*. That is, CodeWarrior provides you with a host of settings that you can adjust to match your programming preferences. If you click on the Edit menu in CodeWarrior, you see a menu item named Preferences. That sounds like it might be the choice to make, but for setting the program name it's not. Instead, choose the item directly beneath the Preferences item — it should have a name something like 68K Debug MacOS Toolbox Settings. When you do that, you see a dialog box like this one:


68K Debug MacOS Toolbox Settings

68K Target

Project: Application

File Name: Animator

'SIZE' Flags:

Startup: Standard

Creator: ????

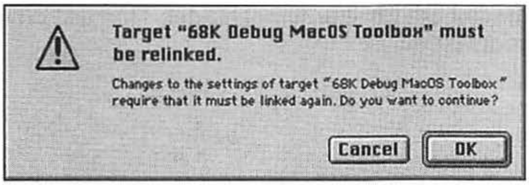
Type: APPL

Preferred Heap Size (k): 384

Minimum Heap Size (k): 384

Don't be alarmed if you make this menu selection and the dialog box you're faced with doesn't look at all like the one I've shown. Metrowerks wanted to pack as much information as they could in a single dialog box, so they came up with something they call *panels*. See the list on the left side of the dialog box? Clicking on any one of the items in this list changes the contents of the dialog box. Each list entry on the left side of the dialog box displays a different panel of information on the right side. To set the name of a program, you want to click the words *68K Target* — as I've done in the preceding figure.

After you click the 68K Target entry in the list, you can type in any name you want in the File Name edit box that appears in the dialog box panel. I chose the name Animator, but you won't hurt my feelings if you give your version of the program a different name. After typing the name, click the Save button. If you change the name of the program, then you may see this alert:


Target "68K Debug MacOS Toolbox" must be relinked.

Changes to the settings of target "68K Debug MacOS Toolbox" require that it must be linked again. Do you want to continue?

What CodeWarrior is telling you here is that in order for the change you made to go into effect, you need to again choose Run from the Project menu. To use the new program name you've selected, CodeWarrior isn't going to search your hard drive for an existing version of Animator and rename it. Instead, CodeWarrior asks you to let it make a new version of the program, while saving the old version. That sounds good to you, so click the OK button. Now click the close box located in the upper-left corner of the dialog box to dismiss the dialog box. Finally, to create a new version of the Animator program — one that actually bears the name Animator — choose Run from the Project menu to let CodeWarrior do its thing.

Checking out the new name

If you didn't change the name of the program, go ahead and do so now. While you may be happy with the name I chose (Animator), you still want to get a feel for the process of setting a name for your own programs. After setting the new name, choose Run from the Project menu to build a new version of the program.

CodeWarrior leaves the original version of the program — the one named Animator — alone. It then makes a new version of the program — the one with the new name — and places it in the same folder as my Animator.mcp project.

Congratulations!?

At this point, congratulations are certainly in order. You just created a Mac program that has a movable window, a menu bar, a functioning menu, and animation!

While Mac programs are capable of doing much more than this, you should certainly feel a sense of accomplishment. Being the clever, curious sort that you are, though, you may already be wondering how you go about reaching the next rung of the Macintosh programming ladder. The next chapter provides you with that information.

Chapter 21

Where Do You Go from Here?

In This Chapter

- Programming after this *...For Dummies* book
- Experimenting with existing source code
- Adding the Apple menu to a program
- Peeking at the source code for a more advanced Mac program
- Getting some information about the full-featured version of CodeWarrior

What's next? That depends . . . did you enjoy programming the Mac? If you did, make sure to read the rest of this book. The next part, The Part of Tens, lists several tips of note to all Mac owners who may want to follow in future programming endeavors. The appendixes also are full of programming code and tips that make your programs more interesting. If you own one of Apple's extremely popular iMac computers, make sure to look over Appendix E to read about some special programming considerations for iMac owners.

You've seen that programming is fraught with challenges, and some of them are quite aggravating. But Mac programming isn't without its rewards. The satisfaction of overcoming what at first appears to be an insurmountable problem can be a reward in itself.

Now that you've gone to the effort of finding out how to program the Mac, what can you do with what you've discovered? You could create a simple game for kids — one that displays faces by drawing shapes. Or you could write a program that draws graphs and charts. Most important, you can use the knowledge gained in this book to move on to bigger and better programming concepts that help you create bigger and better programs.

You are no longer a bewildered novice programmer — you've got a solid foundation in some of the most basic Macintosh programming principles. From here on in, your programs can only get more interesting. In this chapter, you get a couple of final tips on improving your programs. You also take a look at the source code for a program that's a little more sophisticated than the ones you've been working on. (That should prove to be an incentive to buy a Mac programming book aimed at intermediate-level programmers.) I close this chapter with a sneak preview of CodeWarrior Professional — the programming environment you want to consider getting if you're going to continue with Mac programming.

Experimenting

The best way to learn about programming is by experimenting with source code. Now that you have the source code listing for Animator — a real program with a menu and window — start experimenting! You can try changing any of the source code you want, but I want to make a couple of suggestions to get you started.

In your experimenting, you may end up in a predicament that you're unable to extract yourself from. For example, an attempt to compile Animator.c may result in an error message that doesn't make any sense to you. Or the Errors & Warnings window may display just too darn many error messages. Don't fret if you get to this point. Instead, simply drag the whole C20 Animator folder from your hard drive to the Trash can. Then throw this book's CD-ROM in your CD-ROM drive, go to the *...For Dummies* Examples folder on the CD-ROM, and locate the C20 Animator folder. Now drag this folder to your hard drive so that you can start over with a fresh copy of the Animator.mcp project.

Changing the timing of an animation

The Grow Square and Move Square menu items both use the Toolbox function `Delay` to set the speed at which graphics change in the Animator window. To change the speed at which drawing takes place, try changing the value of the `Delay` function's first parameter. Here's how the function call made in the Move Square code looks now:

```
Delay( 2, &theLong );
```

What would happen if you changed it to match the following?

```
Delay( 10, &theLong );
```

Take a guess before you run the code with this change. After you witness what happens, decide whether the number 500 would be a good choice! (Hint: Get ready to take a snack break if you change the code to such a value and then choose Run from the Project menu!)

Changing the loop

Both the Grow Square and Move Square menu items use a loop to perform their animated effects. Changing what goes on within the body of one of these loops can produce all sorts of interesting results. As it stands now, the Grow Square menu item uses a loop with a counter that stops at 120. You can make changes in the body of this loop that affect only certain *iterations,* or times through, the loop. The following shows one such change. Using the following code, the first half of the rectangle is drawn in light gray, and after that, the entire rectangle is drawn in black:

```
if ( count < 60 )
   FillRect( &theRect, &qd.ltGray );
else
   FillRect( &theRect, &qd.black );
```

The following snippet shows where the preceding code should go in the Animator source code listing:

```
case 2:
   SetRect( &theRect, 0, 0, 400, 280 );
   EraseRect( &theRect );
   SetRect( &theRect, 200, 135, 200, 135 );
   count = 0;
   while ( count < 120 )
   {
      if ( count < 60 )
         FillRect( &theRect, &qd.ltGray );
      else
         FillRect( &theRect, &qd.black );
   InsetRect( &theRect, -1, -1 );
   count++;
   Delay( 2, &theLong );   /* slow down drawing */
   }
   break;
```

More ideas, please!

Okay, I've got one more. Read the next section to see how to add the Apple menu to the menu bar of the Animator — or any other program you may be working on.

Adding the Apple Menu

Hundreds of Macintosh programming topics go beyond the scope of this entry-level *...For Dummies* book. Later in this chapter, I introduce a couple of them. First, however, take a look at a final topic that I feel you're quite capable of mastering — one that is a little trickier than what you're used to, but not so difficult that it can't be covered (and understood) in just a few pages. From the title of this section you know that the subject is, of course, adding the Apple menu to the menu bar of an application.

Because you understand the Animator program — a program that includes a menu bar with a functioning menu — it makes sense to modify this program rather than start from scratch. This approach allows me to simply add a small amount of new source code to the source code you're already familiar with. It also allows me to skate by without having to think up an idea for a new program! I did, however, have to come up with a new name for this new program. After several seconds of brainstorming, I came up with AnimatorApple. Grant you, that the name isn't too catchy, but it is descriptive, and it does differentiate it from the original Animator program.

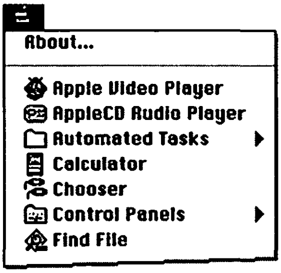
If you look in the ...*For Dummies* Examples folder, you see a folder titled C21 AnimatorApple. That folder holds the files for the project I describe over the next few pages. Whether you're using CodeWarrior Professional or CodeWarrior Lite, open the AnimatorApple.mcp project so that you can follow along.

Understanding why you want to add the Apple to your menu

Why do you want to add the Apple to your menu? Because, quite frankly, Macintosh users expect and demand that it be there. If you are sitting in front of your Mac right now, take a look at the screen. No matter what program you have running, I'll bet that there's a small, multicolored Apple sitting at the far left of the menu bar. That's a pretty safe bet — all but the most trivial programs implement this menu.

So why didn't any of the examples in this book include the Apple menu? Yes, you guessed it — they were all very trivial programs! Don't take that as a criticism of the programs, however. The best way to learn just about any subject is by working with simple examples. So small programs like MyProgram, WindowWorks, MenuDrop, and Animator serve very useful purposes. Now, however, it's time to move on to bigger and better things.

When a user clicks the Apple menu, a menu like the one shown next appears. Because there could be dozens of items in this menu, I cut the menu off after only showing a few items, just to keep it simple. But I think you can still recognize the menu as one you've worked with often.



If you drop down the Apple menu on your own Mac, you may very well see all of the items that appear in the preceding figure. You may also see some, none, or other items. That's because each Mac user has the ability to change the contents of this menu. Any items that a user places in the Apple Menu Items folder in his or her System Folder show up in the Apple menu — no matter what program the user runs. It is this fact that makes adding the Apple menu to your program a little harder than adding other menus. The extra effort that's necessary to add the Apple does, however, pay off. When users of your program see the Apple in the menu bar, they appreciate that they're using a *real* Mac application.

Understanding Apple menu resources

The Apple menu starts its life as a 'MENU' resource — just as any other menu does. As you soon see, you do a couple of things differently as you create this resource. After you create the 'MENU' resource, you'll be back on familiar ground: You add the new 'MENU' resource to the existing 'MBAR' resource — just as you would any other 'MENU' resource.

TIP

As a shortcut in my development of the AnimatorApple program, I made a copy of the Animator.rsrc resource file found in the C20 Animator folder. I then renamed the copied file AnimatorApple.rsrc. I knew in advance that this new program was going to be similar to the program I developed in Chapters 19 and 20, so I thought I'd save myself a little extra work. Always look for effort-saving tricks like this. As a programmer, you've got enough work to do without repeating your prior efforts!

Changing the ID of a resource

I'm working on a copy of the Animator resource file, so the file already holds a 'MENU' resource — the 'MENU' used for the Animator's MyMenu menu. This 'MENU' has a resource ID of 128. Before creating the 'MENU' resource that I want to use for the Apple menu, I want to renumber the ID of this existing 'MENU' resource from 128 to 129. Changing the ID isn't a requirement, but I have my reasons for doing this.

The Apple menu appears in the program's menu bar before (to the left of) the MyMenu menu, so I'd like it to have an ID smaller than 129. For an orderly guy like me, it feels good to have numbers increase in value from left to right — like they did on the number line in school. Now, you may be wondering why I don't just leave the MyMenu ID set to 128 and give the new Apple menu an ID of 127. If you think back to the reason why ResEdit generally starts numbering resources with the number 128, you have the answer. Give up? Remember, Apple reserves the numbers up to 127 for its own use. So it's best to stay away from numbers less than 128.

Enough of the reasoning for making a resource ID change. You just need to do it. The process for changing the ID of any resource is the same. Start at the main window of the resource file, which is the type picker window. That's the window that holds the icons of all the different types of resources in the file. Then follow these steps:

1. **Double-click the icon of interest.**

   A list of resources of that type appears.

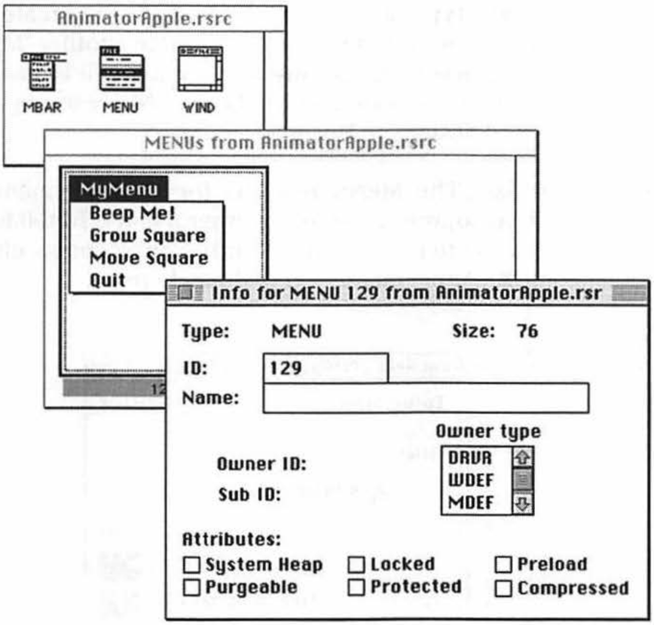
2. **Click once on the resource whose ID you want to change.**
3. **Choose Get Resource Info from the Resource menu.**

   A window with resource information appears.

4. **Type in the new resource ID in the ID edit box.**

In this next figure, you can see that I've followed the preceding steps for the MyMenu 'MENU' resource. Its ID has now been changed from 128 to 129.

A project's 'MENU' resources don't *have* to be numbered consecutively. For example, I could give the Apple 'MENU' an ID of 5000 and the MyMenu 'MENU' an ID of 253. This has no effect on the order in which each 'MENU' will later appear in the menu bar of the program. It is the order in which 'MENU' resources are listed in the 'MBAR' resource that establishes the placement of each 'MENU' in a menu bar.



Type: MENU Size: 76

ID: 129

Name:

Owner type

Owner ID:

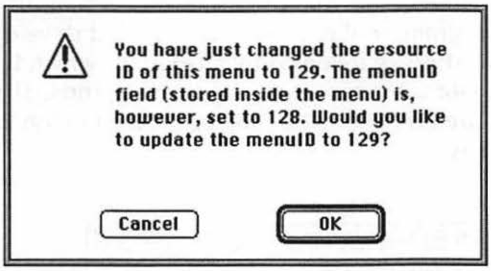
Sub ID:

Attributes:

□ System Heap □ Locked □ Preload

□ Purgeable □ Protected □ Compressed

After typing the new resource ID in the Get Resource Info window, click the window's close box. When you do that, you see this alert:

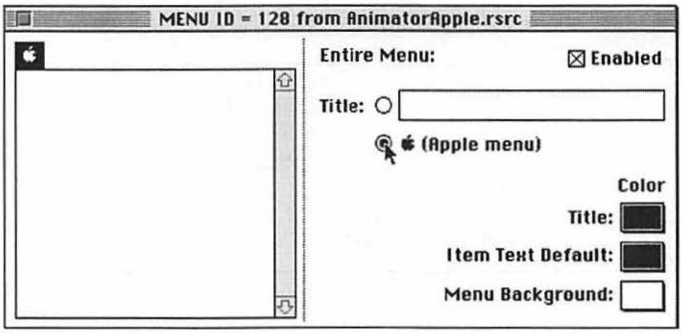

You have just changed the resource ID of this menu to 129. The menuID field (stored inside the menu) is, however, set to 128. Would you like to update the menuID to 129?

Because ResEdit is polite enough to ask my permission before it monkeys around with this mysterious internal menuID field, I reciprocate the kindness by giving ResEdit the green light to do so. You should do the same by clicking the OK button.

Creating the Apple 'MENU' resource

Now it's on to the creation of the new 'MENU' resource. You've performed the first step before — choose Create New Resource from the Resource menu. If you do this with the list of 'MENU' resources open on your screen, ResEdit won't prompt you for the type of new resource you want to create. Instead, it quite appropriately assumes that you want to create another 'MENU' resource, and it opens a new menu-editing window. ResEdit is smart enough to notice that the resource file no longer holds a 'MENU' resource with an ID of 128, so that's the ID it assigns to this new 'MENU'.

Now comes the new part. The 'MENU' resource for the Apple menu is a little different than 'MENU' resources used for all other menus. To tell ResEdit to mark this resource as one that will be used for the Apple menu, click the radio button labeled (Apple menu) as I'm doing here:



MENU ID = 128 from AnimatorApple.rsrc

Entire Menu: ☒ Enabled

Title: ○

● (Apple menu)

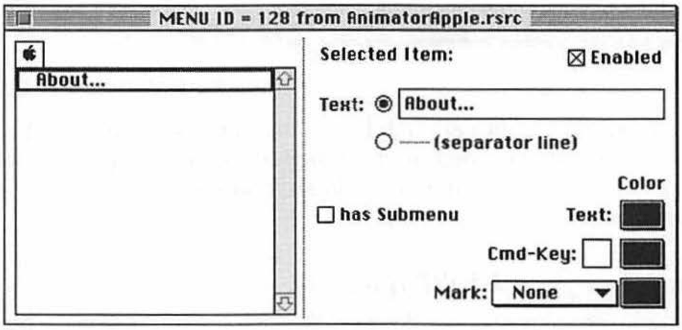
Color

Title: ■

Item Text Default: ■

Menu Background: □

Next, add the items that will appear in the Apple menu. Choose Create New Item from the Resource menu to add the first item. Programs typically make the first item in the Apple menu one that opens a window that displays information about the program or about the company that developed the program. I type in the word *About* followed by an ellipsis, which is pretty much what you see done for most programs. By the way, those three dots that make up an ellipsis are created by holding down the Option key and then pressing the semicolon key.



MENU ID = 128 from AnimatorApple.rsrc

Selected Item: ☒ Enabled

Text: ● About...

○ —— (separator line)

Color

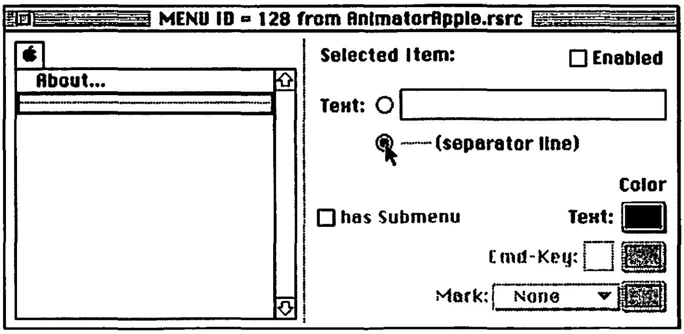
□ has Submenu

Text: ■

Cmd-Key: □ ■

Mark: None ▼ ■

The Apple menu generally has a gray separator, or divider, line as its second item. This line doesn't do anything except provide a visual indicator of where the user's own Apple menu items start. You can confirm this by clicking the Apple menu on your own Mac and taking a look. ResEdit provides you with an easy way to turn a menu item into a separator line. Again, choose Create New Item from the Resource menu. Then click the radio button labeled (separator line):



That's it for the 'MENU' resource that will be used for the Apple menu. What's that, you say, seems like something's missing? You're right there — you haven't added any of the many items that appear in the Apple menu below the separator line. As you see ahead, that's a job handled by source code, not by resources.

Adding the Apple 'MENU' to the 'MBAR'

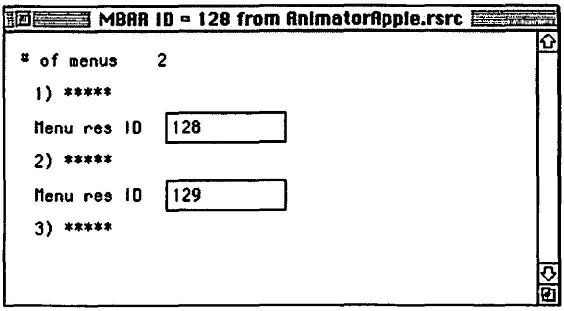
You can create as many 'MENU' resources as you want, but your program won't do anything with them unless they're listed in an 'MBAR' resource. At this point, you have an Apple 'MENU' with an ID of 128 and a MyMenu 'MENU' with an ID of 129. I'm working with a copy of the Animator.rsrc resource file, so I've already got an 'MBAR' resource in this AnimatorApple.rsrc file. It, however, just lists the ID of the one 'MENU' used in the original Animator program. I add a second 'MENU' to it by following these steps:

1. **Double-click the MBAR icon in the type picker window.**

   A list of 'MBAR' resources appears.

2. **Double-click the 'MBAR' resource in the list (you typically only have a single 'MBAR' resource).**
3. **Click once on the last row of asterisks.**
4. **Choose Insert New Field(s) from the Resource menu.**
5. **Click in the new Menu Res ID edit box.**
6. **Type in the ID of the 'MENU' resource to be added to the 'MBAR'.**

After following these steps, the 'MBAR' now looks like this:



```
# of menus     2
  1) *****
  Menu res ID  128
  2) *****
  Menu res ID  129
  3) *****
```

That completes the work for adding the Apple menu — from a resource standpoint anyway. Now it's on to the source code.

Dealing with Apple menu source code

In your quest to add the Apple menu to the Animator program, modifying the resources in the Animator.rsrc resource file represents only half the battle. Now you've got to do a little work on the Animator.c source code. In this section, you see what needs to be added, and why.

ON THE CD

If you're sitting in front of your Mac as you read this, double-click the AnimatorApple.mcp project located in the C21 AnimatorApple folder. Then open the AnimatorApple.c source code file by double-clicking its name in the project window. As you read about the changes to Animator.c, check the AnimatorApple.c file to see exactly where these changes fit in. To make that easy to do, I added plenty of comments to the AnimatorApple.c source code file.

Declaring new variables

The source code for the Animator program declares a dozen variables. AnimatorApple uses all those same variables, plus two more:

```
MenuHandle   appleMenu;
Str255       itemName;
```

The first variable, `appleMenu`, lets AnimatorApple get in touch with the new Apple 'MENU' resource — the program needs to access this resource so that the various items located in the user's Apple Menu Items folder can be added. The second variable, `itemName`, comes into play when the user of AnimatorApple chooses an item from the Apple menu.

New menu bar setup code

The Animator source code uses three lines of code to set up the menu bar:

```
menuBarHandle = GetNewMBar( 128 );
SetMenuBar( menuBarHandle );
DrawMenuBar();
```

The first line gives the program access to the information stored in the 'MBAR' resource. The second line takes that information and does something with it — it sets up, or prepares, the program for the display of the menu bar. It takes the third line to actually draw the menu bar to the top of the screen.

To get a menu bar to recognize and set up the Apple menu, you need some more code — but only a couple of lines. I insert the two new lines of code in the preceding snippet and come up with the following:

```
menuBarHandle = GetNewMBar( 128 );
SetMenuBar( menuBarHandle );
appleMenu = GetMenuHandle( 128 );
AppendResMenu( appleMenu, 'DRVR' );
DrawMenuBar();
```

The first new line calls GetMenuHandle. The parameter to this Toolbox function is the ID of a 'MENU' resource. I give it a value of 128, which in AnimatorApple turns out to be the ID of the new Apple 'MENU' resource. In exchange for that information, the GetMenuHandle function returns something called a MenuHandle to the program. The program tucks this MenuHandle into the appleMenu variable, which is the first of the two variables new to this program. A MenuHandle allows your program to access — to get a handle on, so to speak — the information in a 'MENU' resource.

The second new line of code uses the MenuHandle variable appleMenu to actually do something with the information in the Apple 'MENU' resource. This line of code calls the Toolbox function AppendResMenu to append the names of all of the items in the user's Apple Menu Items folder onto the current contents of the Apple menu. Recall that the current contents of this menu is the About menu item and the dashed line, or separator, menu item. It's this significant line of code that allows the contents of the Apple menu to vary from Macintosh to Macintosh.

Recapping the handling of a selection from a menu

In Chapter 18, you see what happens when the Toolbox function WaitNextEvent comes across an event that happens to involve a click of the mouse button. The Toolbox function FindWindow is called to determine in which part of the screen the mouse click took place. If FindWindow reports that the event occurred while the cursor was over the menu bar, your program handles things by first determining which menu was clicked. Here's an overview of the code section that would appear in a hypothetical program that displayed three menus based on 'MENU' resources with IDs of 128, 129, and 130:

```
case inMenuBar:
   switch ( theMenu )
   {
      case 128:
         /* handle selection from an item in 'MENU' 128 */
      case 129:
         /* handle selection from an item in 'MENU' 129 */
      case 130:
         /* handle selection from an item in 'MENU' 130 */
   }
```

Next, the particular menu item that was selected is determined. That information is used to run the code that exists for just that one menu item. Something like this:

```
case 128:
   switch ( theMenuItem )
   {
      case 1:
         /* handle 1st menu item from 'MENU' 128 */
      case 2:
         /* handle 2nd menu item from 'MENU' 128 */
      case 3:
         /* handle 3rd menu item from 'MENU' 128 */
case 129:
   /* use same approach as shown for 'MENU' 128 */
case 130:
   /* use same approach as shown for 'MENU' 128 */
```

Handling a selection from the Apple menu

The original Animator program has a single menu named MyMenu. This menu is represented by a 'MENU' resource with an ID of 128. In AnimatorApple, I change the ID of this 'MENU' to 129 so that I can give my new Apple 'MENU' the ID of 128. In the AnimatorApple.c source code, that necessitates changing the `case 128` to `case 129`. Because AnimatorApple handles the MyMenu items just as they are handled in Animator, everything beneath this `case` label remains the same. Here's a snippet that provides some context for this change. Again, note that only the first line of this snippet is different:

```
case 129:           /* This was 128 */
   switch ( theMenuItem )
   {
      case 1:
         SysBeep( 1 );
         break;

      case 2:
         SetRect( &theRect, 0, 0, 400, 280 );
      EraseRect( &theRect );
      SetRect( &theRect, 200, 135, 200, 135 );
      count = 0;
      while ( count < 120 )
      {
         FillRect( &theRect, &qd.black );;
```

To handle a menu selection from the new Apple menu, you need a new `case` section. Because the Apple 'MENU' resource has an ID of 128, that's the label you need to give to this new `case` label. Here, in its entirety, is the new code that's needed to handle a selection of any menu item in the Apple menu. Note that even though the items in the Apple menu vary from user to user, the following code can — without modification — always be used:

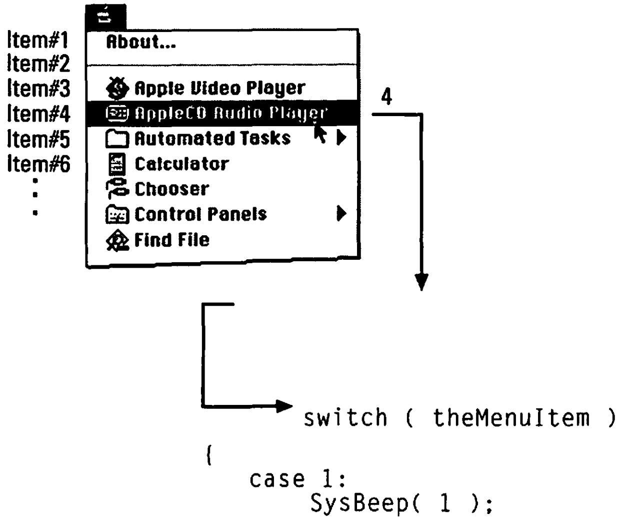
```
case 128:
   switch ( theMenuItem )
   {
      case 1:
         SysBeep( 1 );
         break;
      default :
         GetMenuItemText( appleMenu, theMenuItem, itemName );
         OpenDeskAcc( itemName );
         break;
   }
   break;
```

Most programs that include an About menu item in the Apple menu display a dialog box or an alert in response to the user choosing this item. All I do is call the Toolbox routine `SysBeep` to beep the user's speaker. That's a bit of a cop-out, but my hands are tied here — I didn't cover dialog boxes or alerts anywhere in this book. If you want to find out about those topics, you'll find references to a couple of other, more advanced-level books later in this chapter.

The code under the `case 1` label handles a selection of the first Apple menu item. A selection of *any* other menu item in the Apple menu is covered by the code under the default statement. This may be new to you, so let me take a moment to explain.

A program runs a `switch` statement by comparing the value of the variable in the `switch` statement with each of the `case` labels. When it encounters a match, the code under the matching `case` label is executed. Simple enough. But what if there is no match? Normally, if there isn't a match, the program skips all the code under the `switch` statement and goes merrily on its way. If there's no match, and a section labeled `default` is present, however, the program executes the code under the `default` statement.

Take a look at how the switch and the default work in this particular example. Say the user selects the second item found in his or her Apple Menu Items folder. For this particular user, that item is the AppleCD Audio Player. As shown in the following figure, this item is the fourth item in the Apple menu. That means variable theMenuItem ends up a value of 4. It is this number that is used in the switch statement. Because no case has a label of 4, the switch statement runs the code under the default statement.



```
              switch ( theMenuItem )
 {
     case 1:
         SysBeep( 1 );
```

Only two lines of code are necessary to take care of a menu item selection that involves an item from the Apple Menu Items folder. The first line is a call to GetMenuItemText. This Toolbox function reports the text of a menu item to your program. GetMenuItemText requires three parameters: the menu that holds the item in question, the number of the item in question, and a variable of the data type Str255.

The first parameter is one your program already has. If you go back a few pages to the section titled "New menu bar setup code," you see the code that assigns the MenuHandle variable appleMenu a value. Because the program already has a variable that can be used to reference the correct menu — the Apple menu — use this variable again here.

The second parameter that GetMenuItemText needs is the item number of the selected menu item. Again, the program already has this information. The variable theMenuItem holds this value. So that's the variable you use here.

The last parameter is a string variable of type Str255. You've worked with strings throughout this book. For example, you drew the string *Hello, World!* to a window using the Toolbox function DrawString. A string, like a number, can be stored in a variable. Generally that's done by using a variable of the type Str255. Recall that itemName, one of the two new variables declared by AnimatorApple, is of just such a type. Here's another look at those two variable declarations:

```
MenuHandle   appleMenu;
Str255       itemName;
```

I won't go into all the details of what a Str255 is or how you use one, but I will say this. The *Str* part of this data type stands for *string*. The 255 part refers to the number of characters that a variable of this type can hold, which is 255.

You don't have to assign a string to the itemName variable — the Toolbox does that for you. In fact, that's the sole purpose of the GetMenuItemText function. For example, if the user chooses the AppleCD Audio Player item from the Apple menu, the GetMenuItemText function assigns a value of AppleCD Audio Player to the variable itemName.

After the AnimatorApple program has the name of the selected item, it can use that name to go ahead and open that item. I'm using the word *open* a little loosely here. You can stick anything you want in the Apple Menu Items folder in your System Folder and its name appears in the Apple menu. When you choose an item from this menu, your Mac does what's appropriate for that item. If the item is a folder, the folder opens. If the item is an application, the application runs. The line of code that carries out this magic is the one that calls the Toolbox routine OpenDeskAcc. I show this line of code in the context in which it's used:

```
default :
   GetMenuItemText( appleMenu, theMenuItem, itemName );
   OpenDeskAcc( itemName );
   break;
```

When you pass the name of a file, application, or folder to OpenDeskAcc, the Toolbox finds that item in the Apple Menu Items folder and does its thing on the item. In the previous snippet, you can see that the last parameter to GetMenuItemText and the one parameter to OpenDeskAcc are the same. GetMenuItemText stores the name of the selected menu item in the itemName string variable, and OpenDeskAcc uses this string to determine what item is to be worked with.

Years back, only small applications called *desk accessories* could be stored in the Apple menu. Thus the name of the Toolbox function — `OpenDeskAcc`. Now, anything goes. But while the Mac is capable of storing and opening just about anything in the Apple menu, the old Toolbox routine has kept its same name.

What happens if the user selects the separator line? This is the second item in the menu, yet there's no `case 2` label. And surely the code under the default statement — the code used to handle menu selections of Apple Menu Item folder items — doesn't apply. This isn't an oversight — there's just no need to handle this scenario. Separator lines — no matter what menu they appear in — are *inactive* menu items. Selecting a separator line never produces any results, so you never have to write source code to handle this type of menu item.

Handling update events

A Mac program can recognize several different types of events. In Chapter 17, you experience two such event types, the `keyDown` and `mouseDown` event types. Another event type is the `updateEvt` — the update event. An update event occurs when a window gets moved partially offscreen and then back on screen, or when a window that was partially or fully obscured by a different window is brought to the forefront. When one of these situations occurs, the program has to redraw the contents of the window. That is, things need to be updated.

In AnimatorApple, you aren't too interested in update events. However, in order for the Apple menu to properly function, AnimatorApple must include a `case` section that handles update events. Although a user's selection of an item from the Apple menu may not seem to have much to do with updating anything, it apparently does — if you don't include the following code in the program, the Apple menu won't work as expected:

```
case updateEvt:
   BeginUpdate( theWindow );
   EndUpdate( theWindow );
   break;
```

The preceding code goes right near the `case mouseDown` section — it can go either before or after it. The code under the `case` label includes calls to two Toolbox functions — `BeginUpdate` and `EndUpdate`. Knowing exactly what these routines do isn't too important. Just pass each a `WindowPtr` variable — any `WindowPtr` variable your program declares — and everything will be fine. AnimatorApple declares a `WindowPtr` variable named `theWindow`, so that's what I use as the parameter.

After adding the case `updateEvt` section, AnimatorApple is all set to handle selections from the Apple menu!

Viewing the AnimatorApple source code listing

The AnimatorApple.c source code listing is so similar to the Animator.c source code listing found in Chapter 20 that I don't want to waste paper showing it here. Instead, take a look at the listing on your screen. To do that, open the AnimatorApple.mcp project found in the C21 AnimatorApple folder and then open the AnimatorApple.c file. I've added comments to any new code, so you should be able to quickly spot the changes.

Introducing a More Advanced Program: SightAndSound

I've written this book with several goals in mind, including:

- Getting you acquainted with the C language
- Familiarizing you with source code and resources
- Eliminating your apprehensions about compilers
- Helping you create a Mac program from start to finish

You may notice that absent from the preceding list is a point about creating a sophisticated, exciting application that shows off the graphics and sound capabilities of the Macintosh. In this book, you haven't explored the really fun part of Mac programming. Sorry, but that's just the nature of learning something new — you have to have a good grasp of the basics before moving on to the esoteric.

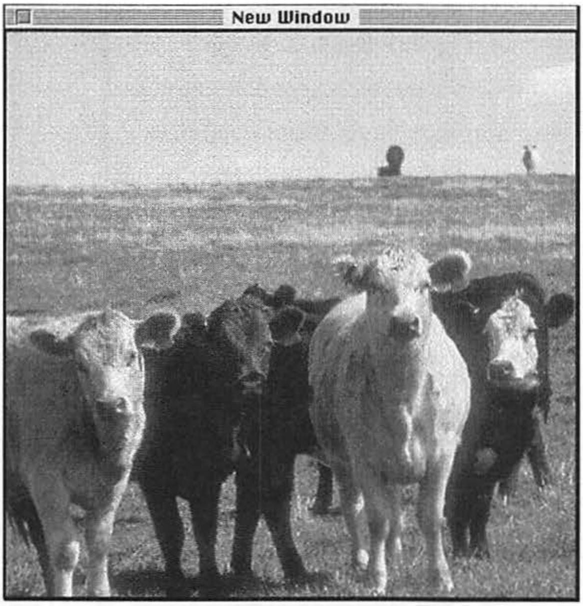
Cool programming topics like displaying color graphics and pictures in windows, playing sounds, running QuickTime movies, and sending the contents of a window to the printer are all covered in other Mac programming books. I list a couple of those books later in this chapter. But before you invest in a second book, ask yourself a few pertinent questions:

- Do I know the basics of programming?
- Do I enjoy the challenge of programming?
- Will I be able to learn more advanced programming topics?

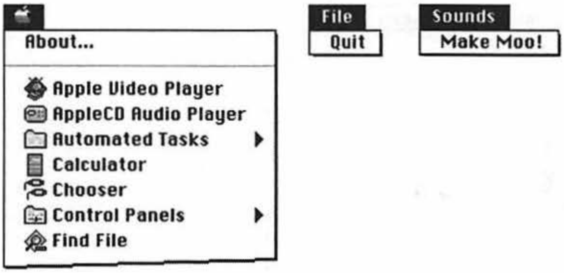
By now you should have the answer to the first two questions. To determine the answer to the third question, it looks like you have to drop the bucks on another book. But wait — don't do that just yet! Instead, read on. Over the next few pages I present the SightAndSound program — an application that demonstrates a couple of the more interesting features a Mac program can have. Unlike the other programs described in the rest of this book, I won't describe SightAndSound line by line. Instead, I just provide some general references to what certain sections of the source code do. You won't be expected to understand exactly what every line of code is doing. But if you find that you have a general feel for what's going on, you can be confident that you can follow the much more thorough descriptions provided in other books.

Understanding what SightAndSound does

When you run SightAndSound, you're presented with a window that displays a picture. While I can't show it in this book, the picture is in living color. You see this amazing color when you run the program on a Mac that has a color monitor.



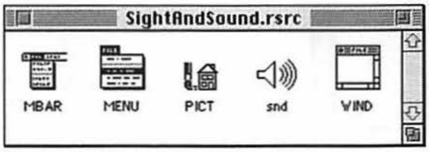
At the top of the screen is a menu bar that holds the following three menus:



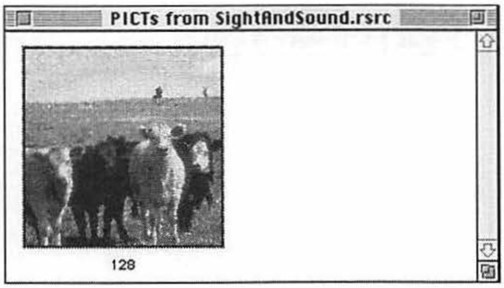
Earlier in this chapter, you see how to implement the first of these menus — the Apple menu. The second menu is the File menu. It holds a single item — Quit. Nothing new so far. The third menu, the one titled Sounds, also has a single item. When you choose Make Moo! from this menu, the melodic sound of cows mooing emits from your Mac's speakers. Why mooing? To match the picture of the cows, of course. Why cows? Because I'm here in Wisconsin, also known as the Dairy State. I could have taken a picture of a brick of cheese and used that instead, but what sound could I have then placed in the Sounds menu?

Pictures and sounds are resources

Throughout this book, you see the important role resources play in Macintosh programs. Using ResEdit to view the contents of the resource file used in the SightAndSound project further illustrates this. Here you can see two types of resources new to you: the 'PICT' and the 'snd' resource types:

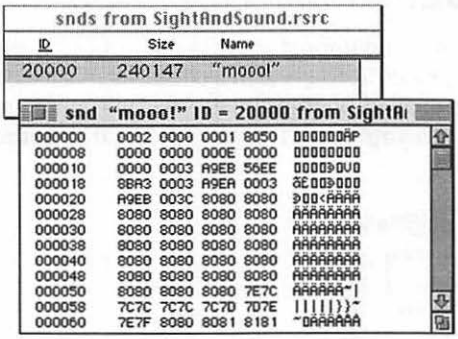


Double-clicking the 'PICT' icon in the type picker window displays the pictures in the resource file. In this example, only one 'PICT' resource appears — but a resource file can hold any number of 'PICT' resources.



ResEdit doesn't display large pictures in their true size. To see a picture in actual size, you double-click the small version of the picture. That opens a new window and displays an actual-size version of the picture in it.

Like pictures, sounds can be stored as resources in a resource file. ResEdit calls such resources 'snd' resources. When you double-click the 'snd' icon in the type picker window, ResEdit displays a list of the sounds in the resource file. The SightAndSound program uses only one sound, so only one 'snd' resource appears in the SightAndSound resource file. Double-clicking on a sound in the list opens a window that displays that sound:


snds from SightAndSound.rsrc

ID	Size	Name
20000	240147	"mooo!"

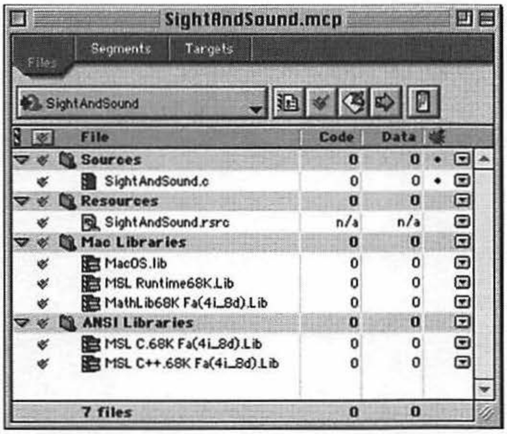
snd "mooo!" ID = 20000 from SightA

```
000000    0002 0000 0001 8050   □□□□□□ÄP
000008    0000 0000 000E 0000   □□□□□□□□
000010    0000 0003 A9EB 56EE   □□□□∋□V□
000018    8BA3 0003 A9EA 0003   ã£□□∋□□□
000020    A9EB 003C 8080 8080   ∋□□<ÄÄÄÄ
000028    8080 8080 8080 8080   ÄÄÄÄÄÄÄÄ
000030    8080 8080 8080 8080   ÄÄÄÄÄÄÄÄ
000038    8080 8080 8080 8080   ÄÄÄÄÄÄÄÄ
000040    8080 8080 8080 8080   ÄÄÄÄÄÄÄÄ
000048    8080 8080 8080 8080   ÄÄÄÄÄÄÄÄ
000050    8080 8080 8080 7E7C   ÄÄÄÄÄÄ~|
000058    7C7C 7C7C 7C7D 7D7E   |||||}}~
000060    7E7F 8080 8081 8181   ~□ÄÄÄÅÅÅ
```

Because no easy way exists to display a sound in a graphical format, ResEdit simply displays the numbers that make up the sound. That's right — the Mac is smart enough to store a sound in a file as a series of numbers. When it comes time to play the sound, the Mac converts these numbers back into a sound. I show you this 'snd' resource to satisfy your curiosity. Although you may have cause to store sounds in a resource file, you normally won't have much of a need to view such a resource.

Workin' with the same ol' kind of project

A program that is more complex than the ones presented in this book may still have a CodeWarrior project that looks very similar to the projects used to create simpler programs. Here's the project window for the SightAndSound program:



File	Code	Data
Sources	0	0
SightAndSound.c	0	0
Resources	0	0
SightAndSound.rsrc	n/a	n/a
Mac Libraries	0	0
MacOS.lib	0	0
MSL Runtime68K.Lib	0	0
MathLib68K Fa(4i_8d).Lib	0	0
ANSI Libraries	0	0
MSL C.68K Fa(4i_8d).Lib	0	0
MSL C++.68K Fa(4i_8d).Lib	0	0
7 files	0	0

From the preceding figure you can see that the files listed in the SightAndSound.mcp project are the same as those named in projects you've already seen — a source code file, a resource file, and a bunch of library files (the names of the library files may differ depending on whether you're using the project from the CD-ROM or a project you created yourself using CodeWarrior Professional). So where does the added complexity come in? A program that does more than another program has more source code and more resources than that program. But it can still keep all its code in a single source code file and all its resources in a single resource file.

A CodeWarrior project *can* hold more than one source code file or more than one resource file. When a project uses a lot of resources or a lot of source code, programmers often do break things up into separate files. This is done simply for organizational purposes — the resulting program remains the same no matter how a programmer distributes code and resources among files.

Glancing at the SightAndSound source code listing

Here, for your reading enjoyment, is the entire source code listing for the SightAndSound program. As I stated a few pages back, you won't be expected to understand everything you see here. Instead, browse through the listing, watching for code that looks familiar and code that doesn't. When you encounter something that looks new, take a moment to try to guess what may be going on. After the listing, I'll point out what some of the new code does.

```
void  OpenWindow( void );
void  HandleAppleMenu( short theItem );
void  HandleFileMenu( short theItem );
void  HandleAnimalMenu( short theItem );

short          allDone;
MenuHandle     appleMenu;
WindowPtr      theWindow;
PicHandle      mooPicture;
Rect           mooRect;

main()
{
  EventRecord   theEvent;
  WindowPtr     whichWindow;
  short         thePart;
  Handle        menuBarHandle;
  long          menuAndItem;
  short         theMenu;
  short         theMenuItem;

  InitGraf( &qd.thePort );
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs( nil );
  FlushEvents( everyEvent, 0 );
  InitCursor();

  menuBarHandle = GetNewMBar( 128 );
  SetMenuBar( menuBarHandle );
  appleMenu = GetMenuHandle( 128 );
  AppendResMenu( appleMenu, 'DRVR' );
  DrawMenuBar();

  OpenWindow();

  allDone = 0;
  while ( allDone < 1 )
  {
    WaitNextEvent( everyEvent, &theEvent, 7, nil );
    switch ( theEvent.what )
    {
      case updateEvt:
        BeginUpdate( theWindow );
        DrawPicture( mooPicture, &mooRect );
        EndUpdate( theWindow );
        break;

      case mouseDown:
           thePart = FindWindow( theEvent.where,
             &whichWindow );
        switch ( thePart )
        {
          case inDrag:
            DragWindow( whichWindow,theEvent.where,
            &qd.screenBits.bounds );
            break;

          case inGoAway:
            DisposeWindow( whichWindow );
            allDone = 1;
            break;

          case inMenuBar:
            menuAndItem = MenuSelect(theEvent.where );
            if ( menuAndItem > 0 )
            {
              theMenu = HiWord( menuAndItem);
              theMenuItem = LoWord(menuAndItem );
              switch ( theMenu )
              {
                case 128:
                  HandleAppleMenu( menuAndItem );
                  break;

                case 129:
                  HandleFileMenu( menuAndItem );
                  break;

                case 130:
                  HandleAnimalMenu( menuAndItem );
                  break;
              }
              HiliteMenu( 0 );
            }
            break;
        }
        break;
    }
  }
}


void  OpenWindow()
{
  short   width;
  short   height;

  mooPicture = GetPicture( 128 );

  mooRect = (**mooPicture).picFrame;
  OffsetRect( &mooRect, - mooRect.left, - mooRect.top );

  width = mooRect.right - mooRect.left;
  height = mooRect.bottom - mooRect.top;

  theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
  SizeWindow( theWindow, width, height, true );
  ShowWindow( theWindow );
```

(continued)

(continued)

```
  SetPort( theWindow );
}


void  HandleAppleMenu( short theItem )
{
  Str255   itemName;
  short    reference;

  switch ( theItem )
  {
    case 1:
      SysBeep( 1 );
      break;

    default :
      GetMenuItemText( appleMenu, theItem, itemName );
      reference = OpenDeskAcc( itemName );
      break;
  }
}


void  HandleFileMenu( short theItem )
{
  switch ( theItem )
  {
    case 1:
      allDone = 1;
      break;
  }
}


void  HandleAnimalMenu( short theItem )
{
  Handle   mooSound;

  switch ( theItem )
  {
    case 1:
      mooSound = GetResource( 'snd ', 20000 );
      SndPlay( nil, (SndListHandle)mooSound, false );
      break;
  }
}
```

Looking at function prototypes

Right after the Sound.h file in the listing come the names of four functions. In Chapter 4 you see that a function is a (usually) small amount of source code written to perform a specific task. Up until now you've worked mostly with Toolbox functions — functions written by Apple and available for use in your own programs. I say you've worked mostly with Toolbox functions because in each program in this book you also see one function not written by Apple — the `main` function.

I wrote the main function in each example program. But I didn't have to stop there. I could have written other functions and included them in a program along with the main function. Why do that? Again, recall from Chapter 4 that one purpose of a function is to isolate a section of code to clarify what's going on in a program. That's what I've done here in the SightAndSound source code. Besides main, I wrote four other functions. I've written the name of each near the top of the source code listing:

```
void  OpenWindow( void );
void  HandleAppleMenu( short theItem );
void  HandleFileMenu( short theItem );
void  HandleAnimalMenu( short theItem );
```

These four lines of code aren't the functions themselves — those appear later in the listing. Instead, these lines are called *function prototypes*. They exist to make it clear to the compiler that this program defines its own functions, and to tell the compiler what it should expect the parameters of those functions to be.

Getting another dose of functions

While I'm on the subject of functions, you should take a look at one. Look back at the source code listing and find the Toolbox initialization code — that's something you're very familiar with. After that code comes the code to set up and display the menu bar. Again, that's code that should look familiar. Just past that, however, is this line:

```
OpenWindow();
```

Because the word OpenWindow is followed by a pair of parentheses, you know it's a call to a function. But OpenWindow isn't a Toolbox function — it's one I defined myself. Look back at the source code listing again and find the end of main. In the past, the end of main was the end of the source code listing. In SightAndSound, however, more code comes after main. The first line of code that follows main looks like this:

```
void  OpenWindow()
```

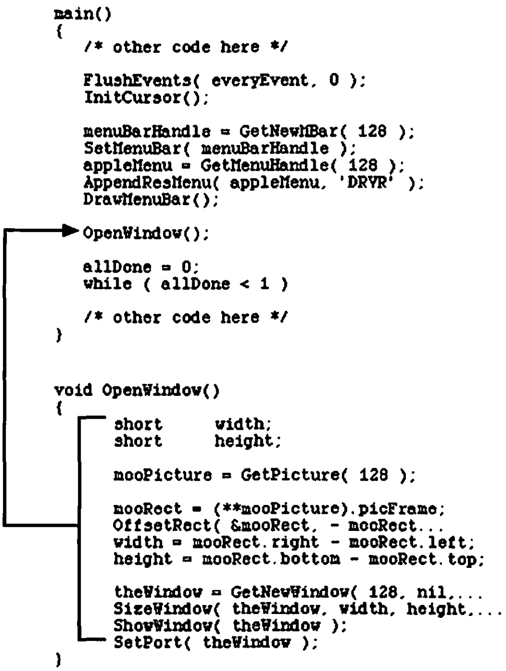
This is the start of the definition of my OpenWindow function. Just as the definition — the body — of the main function appears between braces, so too does the body of the OpenWindow function:

```
void  OpenWindow()
{
  /* function body here */
}
```

What does OpenWindow do? Anything I want it to — 'cause I wrote it! Specifically, this function does the following:

- Gets the picture information from the 'PICT' resource
- Sets up the picture so that its upper-left corner is displayed in the upper-left corner of a window
- Calls `GetNewWindow` to open a new window
- Changes the size of the window so that it matches the size of the picture
- Displays the window on the screen
- Makes the window's port the active port — ready for any other drawing that may take place

I didn't have to write a function that does all of the above. I could have just included all of the above code inside of `main`. I chose to group all this window-related code together in the `OpenWindow` function, though. Now, when the call to `OpenWindow` is made, it's just as if this code did appear in `main` anyway:



```
main()
{
   /* other code here */

   FlushEvents( everyEvent, 0 );
   InitCursor();

   menuBarHandle = GetNewMBar( 128 );
   SetMenuBar( menuBarHandle );
   appleMenu = GetMenuHandle( 128 );
   AppendResMenu( appleMenu, 'DRVR' );
   DrawMenuBar();

   OpenWindow();

   allDone = 0;
   while ( allDone < 1 )

   /* other code here */
}


void OpenWindow()
{
      short     width;
      short     height;

      mooPicture = GetPicture( 128 );

      mooRect = (**mooPicture).picFrame;
      OffsetRect( &mooRect, - mooRect...
      width = mooRect.right - mooRect.left;
      height = mooRect.bottom - mooRect.top;

      theWindow = GetNewWindow( 128, nil,...
      SizeWindow( theWindow, width, height,...
      ShowWindow( theWindow );
      SetPort( theWindow );
}
```

SightAndSound includes three other functions — one to handle each of the three menus. I'll leave it as an exercise for you to find the definitions for these three functions and the calls to each. If you are in any way interested in finding out about these other three functions, then you have some solid proof that you're ready to move on to more advanced Mac programming techniques.

That Wasn't Too Bad; How Do I Learn More?

The SightAndSound source code listing shows you that you already have a good understanding of much of what goes into a typical Mac program. If you'd like to fill in the gaps, though, consider picking up a more advanced Macintosh programming book.

And Now a Few Words about CodeWarrior Professional

If you're interested in continuing with your programming endeavors, you'll want to get a full-featured integrated development environment, or *IDE*. The Lite version of CodeWarrior that's included with this book is great for learning about programming, but it's not enough for more complicated development. In particular, the Lite version of CodeWarrior doesn't allow you to create your own new projects — and that's something you'll need to be able to do if you're to develop your own programs. If you want to really program, you should consider purchasing CodeWarrior Professional.

CodeWarrior Professional is a three-CD set that includes two versions of the CodeWarrior IDE — one that runs on a Mac and one that runs on a PC with Windows 95, Windows 98, or Windows NT. Each version of CodeWarrior includes several compilers. If you're a real glutton for punishment, you can learn all sorts of computer languages and program in any of them without buying separate compilers. With CodeWarrior Professional, you get separate compilers for all of these languages: C, C++, Java, and Pascal.

CodeWarrior Professional also comes with a wealth of electronic documentation. Read it on screen or print it out. Whatever your preference, you'll get plenty of information on using the CodeWarrior environment, programming in C and C++, and more.

CodeWarrior Professional comes with a complete set of tools to debug your code, analyze memory usage, and to profile code at run time with microsecond accuracy. If those sound like advanced topics — you're right. You may not be ready for these programming tools now, but if the time comes when you're a Macintosh programming wizard, you'll be all set.

If you're beginning to feel comfortable with programming, it may be a good time to take a look inside the CodeWarrior Goodies folder on this book's CD-ROM. It holds a *few* of the things you'll find on the CodeWarrior Professional CD-ROMs.

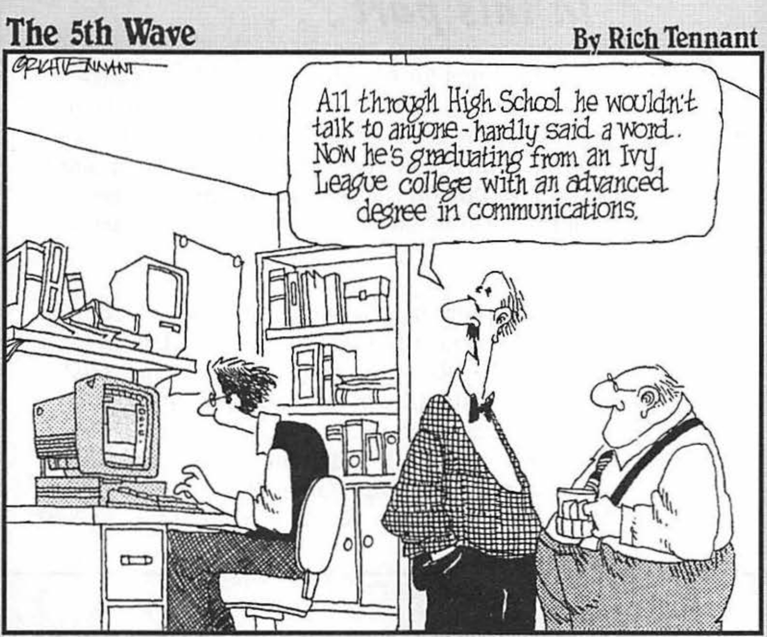
I'll finish up by providing you with a list of much of what's included in CodeWarrior Professional. Don't be intimidated if you don't understand many of the items in this list. Metrowerks takes the the-more-the-merrier approach. They provide everything any Mac programmer could possibly need, and then they let each programmer pick what he or she needs. You don't *have* to use anything except the C compiler — though the more you program, the more you'll certainly want to try out.

- CodeWarrior IDE with project manager, editor, and browser
- Source-level debugger
- Compilers and linkers for several languages
- Support for Mac OS, Windows 95, Windows 98, Windows NT, and Java virtual machine
- Profiling and memory tracking tools
- PowerPlant application framework
- Constructor visual interface builder
- Microsoft Foundation Classes application framework
- Java API framework
- Metrowerks Standard Libraries
- SIOUX input-output console library (for command-line programs)
- Macintosh Toolbox libraries
- Thousands of pages of helpful documentation
- Electronic versions of a few programming books
- Megabytes of tutorial and example code
- Helpful source code and libraries
- Demos of various programmer tools
- One free update when you register

If you're interested in purchasing CodeWarrior Professional, look on this book's CD-ROM. There you find a text file with all the latest scoop on Metrowerks products and how you can go about ordering them. For still more information, visit the Metrowerks Web site at `www.metrowerks.com`.

Part VI

The Part of Tens


The 5th Wave
By Rich Tennant
©RICHTENNANT
All through High School he wouldn't talk to anyone - hardly said a word. Now he's graduating from an Ivy League college with an advanced degree in communications.

In this part . . .

Just what are the steps for creating a CodeWarrior project? After a project is created, which Toolbox functions should you use in the source code? And, finally, what are the common programming mistakes that can be — but often aren't — avoided? You'll find the answers to these three questions — in the form of lists — right here in this part of the book.

Each of the three chapters in this part contain short, concise summaries of what to do (or not do) to create a Mac program. Each chapter has about ten steps, or items, in its list. Why not exactly ten as the title of this part indicates? Sorry, things just don't always work out so evenly. So maybe I took a few liberties when I named the part. But really, now, would you rather it had a more accurate name, like *The Part of Sometimes More, Sometimes Less, But Always Very Close to Ten*?

Chapter 22

Ten Steps to Creating a Mac Program

In This Chapter

- Creating a CodeWarrior project file
- Creating a resource file
- Adding the resource file to your project
- Removing the resource file placeholder from your project
- Creating a new source code file
- Saving the source code file
- Adding the source code file to your project
- Removing the source code file placeholder from your project
- Writing source code
- Compiling and running the code

Creating a new Mac program using CodeWarrior *always* involves the same steps. That's good, because it allows me to create a step-by-step plan that you can follow for every Mac program you write.

Using the Warning icon may be coming on a little too strong, but I did want to catch your eye. The Lite version of CodeWarrior that comes on this book's CD-ROM doesn't allow you to create new projects from scratch — it was designed to let you experiment, compile, and run the example projects in this book. To make your own, brand-new programs, you need the full-featured version of CodeWarrior — Metrowerks CodeWarrior Professional.

Creating a CodeWarrior Project File

The project file is CodeWarrior's way of keeping track of all the information about the program you're developing. You need a separate project file for each program you write.

After launching (starting up) CodeWarrior, choose New Project from the File menu. In the dialog box that appears, you choose a *project stationery*. A project stationery is simply a template that tells CodeWarrior which files need to be included in your new project. Beginners do best by going with the MacOS Toolbox 68K stationery. To get to that stationery, click on MacOS, and then click on C_C++, and finally click on MacOS Toolbox. Sorry about all that clicking, but it's easier than it sounds! Now click once on MacOS Toolbox 68K and then click on the OK button.

Now, a second dialog box appears. Here you get to name the project. Type a name that is at least somewhat descriptive of the program you'll be developing and end the name with a period and the letters mcp (for Metrowerks CodeWarrior Professional), as in KidsGame.mcp.

TIP

The previous dialog box (the one that let you choose a project stationery) included a Create Folder check box. Now, after clicking the Save button here in the Name New Project As dialog box, CodeWarrior will create a new folder and place the new project in that folder. Before clicking the Save button, make sure that the pop-up menu at the top of the dialog box shows the name of the folder where you want the newly created folder to end up. The exact location for this new folder isn't critical, but it should end up somewhere in the main CodeWarrior folder.

Here's a summary of the steps for creating a new project:

1. **Start up CodeWarrior.**
2. **Choose New Project from the File menu.**
3. **Select a project stationery.**
4. **Click the OK button.**
5. **Type a name for the new project.**
6. **Click the Save button.**

Creating a Resource File

You use a resource editor, such as ResEdit, to create a resource file for the program. To switch from CodeWarrior to ResEdit, double-click on the name of the resource file in the project window. If you're working from a new project window, then CodeWarrior will have placed a resource file named SillyBalls.rsrc in the project window — go ahead and double-click on that name. If ResEdit isn't running when you do this, it now runs. If ResEdit is already running when you do this, that's fine. In either case, if you double-clicked on the SillyBalls.rsrc name, then an empty resource file appears on your screen. You want to create a new resource file to be used by your new project. Choose New from the File menu to do that.

Check the pop-up menu at the top of the dialog box that appears. Make sure it shows the name of the folder that holds your CodeWarrior project. If it doesn't, use this menu to navigate to that folder — you want the resource file to be in that same folder so that CodeWarrior can find it.

Next, type a name for the resource file. While not required, it's good practice to give the resource file the same name as the project file (without the *.mcp* part of the project name). Then append *.rsrc* to the end of the file name to make it clear that this is a resource file. Save the file by clicking the New button.

Create a resource for each of the graphical elements your program will use. You'll be making great use of the Create New Resource menu item from the Resource menu to do this. When finished, save the file and quit ResEdit. Remember, ResEdit is a resource *editor,* which means that if you forget to include a resource, you can always use ResEdit to open the resource file and add more resources.

Here are the steps for creating a new resource file:

1. **Create a new project.**
2. **Double-click the SillyBalls.rsrc name in the project window.**
3. **Choose New from ResEdit's File menu.**
4. **Use the pop-up menu to move to the folder that holds the project.**
5. **Type in a name for the new resource file.**
6. **Click the New button.**

Adding the Resource File to Your Project

Creating a resource file isn't enough. You also have to make CodeWarrior aware of the fact that you want this particular file to be a part of your CodeWarrior project. To do that, click once on the SillyBalls.rsrc name in the CodeWarrior project window. Then choose Add Files from the Project menu. Use the pop-up menu at the top of the dialog box that opens to work your way into the folder that holds the resource file you want to add to the project. Click once on this file's name in the top list of the dialog box and then click the Add button. The file moves to the bottom list. Click the Done button to finalize your addition and to dismiss the dialog box.

These are the steps for adding your resource file to your project:

1. **Make sure that you're working with CodeWarrior and not ResEdit (click the project window or choose CodeWarrior from the menu at the far-right end of the menu bar).**
2. **Click the SillyBalls.rsrc name in the project window.**
3. **Choose Add Files from the Project menu.**
4. **Use the pop-up menu to move to the folder that holds the resource file.**
5. **Click the resource file name in the list.**
6. **Click the Add button.**
7. **Click the Done button.**

Removing the Resource File Placeholder

A new CodeWarrior project window holds a resource file placeholder named SillyBalls.rsrc. This file exists primarily as a placeholder of sorts — it is nothing more than a reminder for you to add your own resource file to the project. After adding the resource file, remove the placeholder. First, click once on the name SillyBalls.rsrc in the project window. Then choose Remove from the Project menu. That's it — the placeholder file is gone!

To remove the resource file placeholder:

1. **Click the SillyBalls.rsrc name once in the project window.**
2. **Choose Remove from the Project menu.**

Creating a New Source Code File

You type your source code into a text file that's created in CodeWarrior. To create this new text file, choose New from the CodeWarrior File menu to open an untitled, empty window.

Saving the Source Code File

You can save your source code file at any time, but I suggest that you do so before you even type a line of code, just so you don't forget. Choose Save As from the File menu. Type in a name that is appropriate for the program you're writing. This can be any name you want; but for a C language source code file, it must end with *.c* (a period and the letter *c*). Before clicking the Save button, use the pop-up menu at the top of the dialog box to work your way into the folder that holds your resource file and project file.

To save a new source code file:

1. **Choose Save As from the File menu.**
2. **Type in a name for the new source code file (end the name with *.c*)**
3. **Use the pop-up menu to move to the folder that holds the project file.**
4. **Click the Save button.**

Adding the Source Code File to the Project

Creating a new source code file isn't enough to make CodeWarrior aware of its existence. You've got to add the new file to the project. Don't be concerned with the fact that you don't have any code typed in the new file. Begin by clicking once on the source code file placeholder in the project window — that would be the SillyBalls.c name. Then choose Add Window from the Project menu.

Again, here's a list of the steps you perform to add a source code file to a project:

1. **Click the SillyBalls.c name in the project window.**
2. **Choose Add Window from the Project menu.**

Removing the Source Code File Placeholder

A new CodeWarrior project window holds a source code file placeholder named SillyBalls.c. Like the resource file placeholder, the purpose of this entry in the project window is to serve as a reminder — a reminder that you not only need to create a source code file, but that you must add the file to the project. After adding the source code file, remove the placeholder. First, click once on its name in the project window. Then choose Remove from the Project menu.

To remove the source code file placeholder:

1. **Click the SillyBalls.c name once in the project window.**
2. **Choose Remove from the Project menu.**

Writing the Source Code

Type away! Or do what most programmers do. Choose Open from the File menu and open an existing C source code file — one you created for a different program. Copy part or all of it and then paste it into your new file. Now use the old code as the basis for your new program. Make the necessary changes to turn the old code into new code. In any event, choose Save from the File menu occasionally to save your work.

Compiling the Source Code

When you feel satisfied that your source code looks complete, choose Compile from the Source menu. Hopefully, things will be uneventful. If that's the case, your code was successfully compiled. If a window opens up with an error message, things didn't go quite so smoothly. Take note of the error message and make the necessary corrections to your source code. If you can't figure out the message, try referring to Appendix C. There I list several errors and possible cures that get rid of the pesky messages. After your corrections are made, try compiling again. Keep trying until you get it right!

Running the Code

CodeWarrior has a powerful menu item named Run, which you find in the Project menu. When you choose this one item, CodeWarrior compiles your source code file, builds a standalone application, saves that program to your hard drive, and then gives the program a test run. As the program runs, test out whatever it is your program is supposed to be able to do. Click menus and move the window — try out all the features you added to the program. When through, quit the program. If you aren't entirely satisfied with the results, or one or more features don't work as you expected, it's time to look over your source code to figure out what went wrong. Make any changes that you want and then run the program again.

Chapter 23

Ten Toolbox Functions You Can't Live Without

In This Chapter

- Initializing functions
- Displaying a window
- Activating a window
- Displaying a menu bar
- Getting event information
- Determining the location of a mouse click
- Understanding window functions
- Understanding menu functions
- Understanding graphics functions

The Toolbox is Apple's name for the collection of the thousands of functions that Mac programmers can use to get their programs to do all sorts of wonderful things. Two dozen of these functions are scattered throughout this book and even more are listed in Appendix B. While you may find many of the Toolbox functions interesting, many are indispensable, and those are the ones I list here.

Using the Toolbox Initialization Functions

The eight functions that initialize the Toolbox are vital to any Mac program. Call them at the start of your program, right after you declare your variables. And call them in the order I list here:

```
InitGraf( &qd.thePort );
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();
```

Eight functions? But that leaves just two more for my list of ten functions you can't live without. Hey, that's no fun! I didn't intend to kill the whole list on boring initialization stuff. So I hope you don't mind if I group all eight of these functions together and only count them as one.

Displaying a Window

Everything that's drawn in a Mac program is drawn in a window. So the function that displays a window is one of the most important of all Toolbox functions. Here's a typical call to the function that performs this task, along with the one variable declaration you need to make:

```
WindowPtr  theWindow;

theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

If your program calls GetNewWindow, make sure that the resource file for the project has a 'WIND' resource in it with an ID that matches the first parameter in the call to GetNewWindow.

GetNewWindow returns a WindowPtr to your program. You can use that WindowPtr in other Toolbox calls, including the one I'm about to cover.

Preparing a Window For Drawing

After creating and displaying a window, you want to draw to it — text, graphics, or both. But before you start whipping out those fancy pictures, make sure to tell the Mac to draw to the new window with a call to SetPort:

```
SetPort( theWindow );
```

SetPort tells the Mac which window it should draw in, which is always important, especially so if more than one window is open. To help SetPort do its thing, you pass it the WindowPtr of the window. Use the WindowPtr variable that was returned by the call to GetNewWindow.

Displaying a Menu Bar

You need to make one variable declaration and three Toolbox calls in order to display a menu bar:

```
Handle  menuBarHandle;

menuBarHandle = GetNewMBar( 128 );
SetMenuBar( menuBarHandle );
DrawMenuBar();
```

Pass GetNewMBar the ID of the 'MBAR' resource that's in the project's resource file. In return, GetNewMBar gives your program a Handle. That's something that the SetMenuBar function needs in order to gather up all the information about the menu bar and the menus in it. Finally, a call to DrawMenuBar displays the menu bar on the screen.

Oops, there I've gone and done it again! I'm counting a few functions as one so that I can cram as much important stuff as possible into this Part of Tens chapter. As a matter of fact, don't be surprised if I do this a few more times.

Capturing Events

Capturing events — sounds exciting, doesn't it? Well, it isn't really too thrilling, but it sure is easy, thanks to WaitNextEvent. This function looks for an event and, when it notices one, stores all the information about the event in an EventRecord variable:

```
EventRecord  theEvent;

WaitNextEvent( everyEvent, &theEvent, 7, nil );
```

You use the information housed in the EventRecord at several points in your program. One of those occasions is when the user clicks the mouse button.

Locating a Mouse Click

The user can click the mouse anywhere on the screen. It's up to your program to figure out where the cursor was at the time of the mouse click. Your program needs that information to determine how to respond to the click. To get that information, call FindWindow:

```
EventRecord   theEvent;
WindowPtr     whichWindow;
short         thePart

thePart = FindWindow( theEvent.where, &whichWindow );
```

FindWindow uses information from the EventRecord variable, so you call the function after WaitNextEvent. In particular, the where part of the EventRecord is examined. (Refer to Chapter 17 for more information on how a single EventRecord variable holds more than one piece of information about an event.) When completed, FindWindow gives your program the screen or window location where the mouse click took place. It stores this information in variable thePart. If the mouse click occurs in a window, FindWindow also uses the variable whichWindow to provide your program with a WindowPtr to the window.

Working with Windows

If FindWindow tells your program that a mouse click occurred in the drag bar of a window, you should respond by calling DragWindow. The DragWindow function follows the cursor as the user moves the mouse and moves the window accordingly.

```
WindowPtr     whichWindow;
EventRecord   theEvent;

DragWindow( whichWindow, theEvent.where, &qd.screenBits.bounds );
```

The first parameter of DragWindow tells the Toolbox which window to drag. This variable gets its value from the call to FindWindow that was made earlier. The second parameter is the location where the mouse is first clicked. The last parameter tells DragWindow to allow the window to be dragged anywhere on the screen.

If FindWindow tells your program that a click of the mouse was made in the close box of a window, you should call DisposeWindow to remove the window. FindWindow notifies your program which window needs closing by assigning a value to the WindowPtr variable whichWindow.

```
WindowPtr   whichWindow;

DisposeWindow( whichWindow );
```

Managing Menus

Displaying a menu is one thing, making it usable is another. When the user clicks the mouse in the menu bar, call MenuSelect to take care of the work of dropping menus as the user moves the mouse over them. When the user makes a menu selection, MenuSelect is smart enough to know which menu item is selected and from which menu. It stores this information in variable menuAndItem.

```
long   menuAndItem;

menuAndItem = MenuSelect( theEvent.where );
```

Call the Toolbox functions HiWord and LoWord to extract the number of the menu and the number of the menu item from menuAndItem.

```
short   theMenu;
short   theMenuItem;

theMenu = HiWord( menuAndItem );
theMenuItem = LoWord( menuAndItem );
```

The call you make to MenuSelect highlights a menu name in the menu bar. After you take care of business, call HiliteMenu to return the menu name to its original state, which is typically black text on a white background:

```
HiliteMenu(0);
```

Drawing Text

On a Macintosh, text is said to be drawn, not written. Use the Toolbox function DrawString to draw a word or a sentence to a window. Enclose the text in double quotes and precede the first word of text with the backslash character (\) and the letter *p*.

```
DrawString( "\pDrawing text is easy!" );
```

Where does the text get drawn? I know, wiseguy — in the window. More specifically then, where in the window? That depends on the parameters you pass to the function MoveTo. Tell MoveTo how many pixels to move in from the left edge of a window and how many pixels to move down from the top. Then call DrawString:

```
MoveTo( 20, 50 );
DrawString( "\pDrawing text is easy!" );
```

Drawing Shapes

You can use the Toolbox function FrameRect to frame a rectangle, or you can use FillRect to fill in a rectangle with a pattern. But first tell the Toolbox where the rectangle should go and how big it should be. Use a call to SetRect for this purpose:

```
Rect  theRect;

SetRect( &theRect, 0, 0, 400, 280 );
```

Don't mix up the parameters of `SetRect`. Try thinking of them like this:

```
SetRect( &theRect, left, top, right, bottom );
```

After setting up the rectangle, call `FrameRect` to draw a black frame around it:

```
FrameRect( &theRect );
```

If you'd like to fill the rectangle, call `FillRect`. The second parameter tells the Toolbox what pattern to fill the rectangle with. Here's a call that fills the rectangle with a light gray pattern:

```
FillRect( &theRect, &qd.ltGray );
```

You can also use black, white, gray, or dkGray, preceded by `&qd.`, to achieve other fill effects.

Chapter 24

The Ten Most Common Mac Programming Mistakes

In This Chapter

- Problems with the resource file
- Mismatched braces
- Too many semicolons
- Using the wrong capitalization
- Problems drawing a string of text
- Function parameters and the & character
- Loop counters and infinite loops
- Giving a variable its starting value
- Problems with the `switch` statement

In programming, making mistakes is commonplace. Everyone who writes programs makes them. While there are easily more than ten different mistakes a programmer can make, there are about ten or so slip-ups that just about everyone new to programming the Mac makes. I outline them in this chapter so that you can be on the lookout to avoid as many of them as possible.

Having Trouble with the Resource File

Don't forget to make a resource file, and don't forget to add it to your project! In your excitement to start writing code, you may jump right in and forget all about resources. Some Toolbox function calls look for a resource. The `GetNewWindow` function is an example:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

What does the number 128 represent? The ID of a 'WIND' resource. What happens if you run a program with this GetNewWindow line in it and that resource can't be found? You don't get a friendly little reminder to the fact that you forgot to make a resource. Instead, your program will simply quit, or the screen of your Mac freezes up and an alert appears. This dreaded alert may say something about a bad F-Line instruction. I'm not exactly sure what the heck a bad F-Line instruction is, but judging from the fact that a little bomb appears in the corner of the alert and the Mac freezes up, I can only assume that it's not good. What's particularly troubling, though, is that the error message gives you no clue that the problem is resource-related. A few things can give you an alert like the one mentioned above:

- ✓ No resource file added to the project.
- ✓ A resource file is in the project, but a resource is missing from that file.
- ✓ A resource file is in the project, and the resources are all present in the file, but you've used an incorrect resource number in your code.

If you forgot to add your resource file to the project, use the Add Files menu item from the CodeWarrior Project menu to do that. Chapter 9 introduces this menu item. If you've added the resource file, but forgot to add one or more resources to that file, use ResEdit to take care of that task. Chapters 7 and 8 provide more information on adding resources to a resource file. If you've referenced a non-existing resource (say, you've used 129 as the first parameter in a call to GetNewWindow when in fact the 'WIND' resource in your resource file has an ID of 128), make the correction in your source code. Chapters 7 and 8 discuss resource ID numbers. See Chapter 15.

Not Pairing Braces

Every opening brace must have a corresponding closing brace. For example, the code under a loop begins with a brace, so it must end with one as well:

```
while ( count < 10 )
{
   DrawString("\pTest");
   count++;
}
```

What happens if you forget a brace? The CodeWarrior compiler responds with an error message about an *expression syntax error.*

Adding an Extra Semicolon

In C, a semicolon ends just about every line. So when you're typing code, it's easy to start sticking semicolons everywhere. That can lead to problems because the first line of a branch or loop *doesn't* end with a semicolon.

This is correct:

```
while ( count < 100 )
```

This isn't:

```
while ( count < 100 );
```

Look closely at the preceding two lines of code. The second `while` statement ends with a semicolon, which is wrong. In C, declarations, assignments, and Toolbox calls all end with a semicolon:

```
short   dogs;           /* declaration */
dogs = 5;               /* assignment */
MoveTo( 30, 50 );       /* Toolbox call */
```

But loops don't:

```
while ( count < 100 )  /* no semicolon */
```

And branches don't either:

```
switch ( thePart )      /* no semicolon  */
```

Adding a semicolon to the end of the first line of a loop or branch may or may not give you an error message. Even if the code does compile successfully, the extra semicolon can lead to real confusion. Your code runs, but the results won't be as expected. Adding a semicolon to the first line of a loop causes the lines below the loop to run only once regardless of how many times you hoped they would run. For a branch that inadvertently ends with a semicolon, the lines beneath always run — even if you don't want them to.

Using Incorrect Case

What's wrong with the following declaration?

```
windowPtr   theWindow;
```

Here's a hint: C is case-sensitive, meaning that the proper use of uppercase and lowercase letters is very important. The C data type for a pointer to a window is a *`WindowPtr`*, not a *`windowPtr`*. That first letter makes a big difference. If you make a mistake of this type, the CodeWarrior compiler displays an error message that says an *undefined identifier* error occurred. To a compiler, an identifier is essentially a variable. Because the compiler doesn't recognize *`windowPtr`* as a data type, it assumes it's an identifier, a variable. And because a variable named *`windowPtr`* wasn't defined by the program, CodeWarrior determines that something is wrong.

Forgetting the \p in DrawString

The *DrawString* function is a very handy and easy-to-use Toolbox function, but it does take a little getting used to. Several mistakes can occur when using it, and most result in an error message that says something like *Cannot convert 'char *' to 'const unsigned char *'*. I want to show a correct usage of *DrawString*:

```
DrawString( "\pHello, World!" ); /* CORRECT! */
```

The number one mistake in using *DrawString* comes when a programmer forgets to include *\p* before the text, like this:

```
DrawString( "Hello, World!" );   /* WRONG! No \p */
```

Another common mistake is to forget the quotations before and after the text:

```
DrawString( \pHello, World! );   /* WRONG! No quotes */
```

And while you may remember to use quotes, you could mistakenly use the wrong type of quote marks. *DrawString* requires double quotes, not single:

```
DrawString( '\pHello, World!' );  /* WRONG! Wrong quotes */
```

Finally, be careful which slash you use. The backslash (\) is the correct one. Here's a look at the incorrect slash being used:

```
DrawString( "/pHello, World!" );  /* WRONG! Wrong slash */
```

Forgetting the & with a Parameter

You may notice that in many Toolbox calls one of the parameters requires that you include the & character before it. I never liked the awkward name of this character — the ampersand — and after working with C I know why. The ampersand can be the cause of a lot of programming headaches. Here's a typical Toolbox call that requires its use:

```
SetRect( &theRect, 0, 0, 400, 280 );
```

If you try to call *SetRect* without using the ampersand before *theRect*, you get an error message. This message is much like the one you get when you incorrectly use *DrawString*, which is something about not being able to convert one thing to another. Because many Toolbox functions have parameters that don't require the ampersand, forgetting to include it when required is an easy mistake to make.

I can't give you any set rule as to when a parameter requires the ampersand. You have to make sure that you match your source code with that in this book. The good news is that you don't have to page through the entire book to find an example of a call to each function. Before you use a function, you can refer to Appendix B. There I list all of the functions used in this book and a few others. I also give an example call to each so that you can see what the parameters should be.

Forgetting to Increment a Loop Counter

How many times will the following loop run? Three times? Four times?

```
count = 0;

while ( count < 4 )
{
   DrawString( "\pHello!" );
}
```

How many times? That depends on when you turn your computer off — cause that's the only thing that can stop this loop from running! The variable *count* is not incremented anywhere in the loop. That means that *count* always has a value of 0, and 0 is always less than 4. That's called an *infinite loop*. Let me rewrite the loop with *count* properly incremented:

```
count = 0;
while ( count < 4 )
{
   DrawString( "\pHello!" );
   count++;
}
```

Forgetting to Give a Variable an Initial Value

When you declare a variable, it doesn't have a value. Actually, it may have a value, but you can't be sure of what that value is:

```
short  allDone;  /* variable allDone = ?? */
```

After you use a variable in an assignment statement, you know its value:

```
allDone = 0;     /* variable allDone has a value of zero */
```

Using a variable in a branch statement or loop statement before giving it a value can lead to unexpected results. Here's some of the code from my Animator program. This code appears right at the start of the program's event loop:

```
allDone = 0;
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
   /* rest of program here */
```

What would happen if I omitted the line that assigns *allDone* a value of 0? Variable *allDone* could then easily have a value other than 0. If that's the case, what happens when the program reaches the next line of code — the *while* statement? *allDone* may not be less than 1, the loop won't run, and the Animator program ends. And that's not good.

Forgetting a Break in a Switch Statement

A *switch* statement places code in groups and lets your program select which group to run. Each group of code starts with a *case* label and ends with a *break* statement. Here's an example:

```
switch ( booksWritten )
{
   case 1:
      DrawString("\pYour first book!");
      break;
   case 2:
      DrawString("\pYour second book!");
      break;
}
```

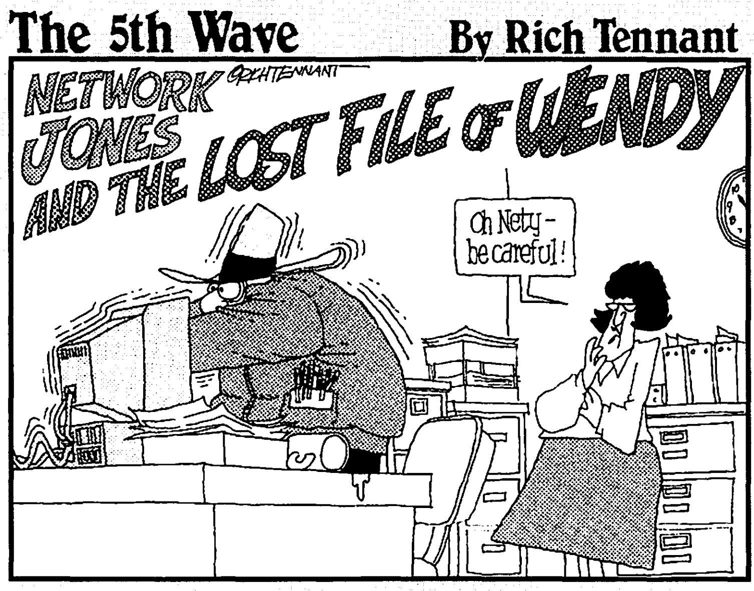
If *booksWritten* has a value of 1, then the code under the first *case* label runs. If *booksWritten* has a value of 2, then the code under the second *case* label runs. Now, look at the same example without a *break* statement after the code under the first *case* label:

```
switch ( booksWritten )
{
   case 1:
      DrawString("\pYour first book!"); /* forgot a break */
   case 2:
      DrawString("\pYour second book!");
      break;
}
```

Now what happens if *booksWritten* has a value of 1? The code under both *case* labels runs! The *break* statement is the signal to the *switch* to stop running. Without a *break*, the determined *switch* just plods on to the next line of code, the *case 2* label, and then on to the code under it.

Part VII

Glossary and Appendixes



In this part . . .

Just when you think that you've got this C stuff under control, you realize that you don't remember how to write an if-else branch. Or you forget what goes between the parentheses that follow a call to the Toolbox function `GetNewMBar`. And don't you hate it when you get one of those error messages and you can't figure out how to make it go away? It's almost as infuriating as coming across a Mac programming term and not remembering what it means.

Well, don't get frustrated. In this part, you can find help and answers to get you through all of these situations and others. And a special bonus: You'll also find a description of the content's of the book's CD-ROM in this part, and you'll discover how to access CodeWarrior Lite and other files that come up in the course of this book. For you iMac owners, there's even an appendix that supplies you with a few special programming tips.

Appendix A

C Language Reference

Entire books have been written on the C language, so how did I manage to cover it all in just a few pages of an appendix? By not covering it all, of course. But I do summarize all the features of C that appear in this book, plus a few additional choice tidbits of C not covered in this book.

Variables

A variable holds a value. What kind of value? That depends on the type of the variable. See also "Data Types."

Declaring a variable

A variable must be declared so that the compiler becomes aware of it. The format of a variable declaration is always the same regardless of the type of variable. First list the data type, and then the variable name. Follow that with a semicolon. Here's an example that declares a variable named `tickets` to be of the `short` data type:

```
short   tickets;
```

Giving a variable a name

Your variables should have names that describe the type of information they hold. When you name your variables, keep a couple of restrictions in mind:

- Use only letters, digits, and the underscore character in a variable name.
- The first character of the name must be a letter or underscore — not a digit.

Assigning a variable a value

When you declare a variable, it has no value. Well, it may have some random value, but probably not the value you want it to have. To give a variable a value, first specify the variable you're working with. Follow that name with the equal sign and then the value you want to assign. End it all with a semicolon. In the following example, I declare a `short` variable named `Total` and then give it a value of 25:

```
short   Total;

Total = 25;
```

There's a second way to assign a variable a value. When you declare the variable, you can give it a value on the spot. Here's an example that has the same outcome as the previous two lines of code:

```
short   Total = 25;
```

Data Types

When you declare a variable, you state the type of data that variable holds, and you also state the name of the variable.

Number types

If you want a variable to hold a whole number, you have three options. The `int` data type holds numbers as high in value as 32,767. So does the `short` data type. Though the `int` and the `short` data types are just about one and the same, I recommend you use the `short` type. In the Macintosh world, the `short` data type is becoming much more popular than the `int` data type. I'm not sure why the `short` is so trendy, but I want to be hip, so I use it.

For numbers larger than 32,767, use the `long` data type — it holds whole numbers as large as 2 billion. If the number you want to use is greater than 2 billion, or it contains a decimal point, use the `float` data type.

Window types

Variables don't always hold numbers. Sometimes data types hold other information. The `WindowPtr` data type is an example. If you call a Toolbox function that works with a window, you have to let the Toolbox know which window you're referring to. When a window is created, your program gets something called a pointer to the window — a `WindowPtr`. Think of this `WindowPtr` as a reference to one particular window.

Here's the declaration of a window pointer variable and its use in two Toolbox functions:

```
WindowPtr   theWindow;

theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );
```

Menu types

When you display a menu bar in your program, you have to help the Toolbox out by providing it with a `Handle` to the menu bar. The `Handle` has many purposes, but in this book, you only see it used with the menu bar. The `Handle` helps the Toolbox get a handle on (so to speak) the information that describes the menu bar. Here's how I use the `Handle` data type in this book:

```
Handle  menuBarHandle;

menuBarHandle = GetNewMBar( 128 );
```

Operators

In C, the symbols that perform different mathematical and comparative operations are called *operators.*

Math operators

C has a symbol, or operator, for each of the four major mathematical operations: addition, subtraction, multiplication, and division. Intuitively enough, the plus sign (+) symbolizes addition, and the minus sign (-) represents subtraction. For multiplication, place an asterisk (*) between two variables or numbers. For division, use the slash symbol (/). Here are several examples of math operators:

```
short   score1;
short   score2;
short   total;

score1 = 30;
score2 = 10;

total = score1 + score2;   /* total = 30 plus 10       =  40  */
total = score1 - score2;   /* total = 30 minus 10      =  20  */
total = score1 * score2;   /* total = 30 times 10      = 300  */
total = score1 / score2;   /* total = 30 divided by 10 =   3  */
```

You can use operators on any combination of variables and/or numbers, as shown in these examples:

```
short   score1;
short   score2;
short   total;

score1 = 25;
score2 = 5;

total = score1 + score2 + 10;   /* total = 25 + 5 + 10   =  40  */
total = score1 - score2 + 10;   /* total = 25 - 5 + 10   =  30  */
total = score1 / 5;             /* total = 25 / 5        =   5  */
total = 50 + (score1 * score2);   /* total = 50 + (25 * 5) = 175  */
```

I put a new twist into that last example — a pair of parentheses. If you have more than one operation on one line of code, you can tell the computer which operation to perform first by setting it off in parentheses. In the last example above, variable `score1` is multiplied by variable `score2` first. Then 50 is added to that result.

In programming, you often want to increment the value of a variable by one. The increment operator, which is two plus signs in a row (++) does that. Here a variable named `index` is first given a value of 5 and then incremented to 6:

```
short   index;

index = 5;   /* index now equals 5 */
index++; /* index now equals 6 */
```

Adding one to the value of a variable is called *incrementing* the variable. Subtracting one from a variable is called *decrementing* the variable. The C language provides a means to easily do that. Use the decrement operator, which is two minus signs in a row (--). Here's an example:

```
short   index;

index = 5;   /* index now equals 5 */
index--;     /* index now equals 4 */
```

Comparative operators

Programmers often test the value of a variable by comparing it to the value of another variable or to a number. Programmers perform these tests to determine whether a loop should run another time. To perform this test, use one of the comparative operators. The less-than operator (<) checks to see whether the value to the left of it is less than the value to the right, as in this example:

```
while ( count < 5 )    /* is count less than 5 ?     */
```

The greater-than operator (>) determines whether the value to the left of the operator is greater than the value to the right:

```
while ( count > 0 )    /* is count greater than 0 ?  */
```

Assignment operators

Every time you assign a variable a value, you're using an operator — the assignment operator. That's what C calls it, but you and I would simply refer to it as the equal sign (=). In this example, variable `numberOfWins` receives a value of 7 through the use of the assignment operator:

```
short   numberOfWins;

numberOfWins = 7;
```

Looping Statements

By setting up a loop, you can make any section of source code run more than once. And by changing a single number in the loop, you can have that section of code run two, ten, or ten thousand times. Now that's power!

The while statement

A `while` loop, or `while` statement, begins with the word `while` followed by a test in parentheses. The test determines how many times the code under the `while` runs. The test compares a variable with a value (which could be another variable). The test is performed after each running of the code beneath the `while`. If the test condition is not met, the test is said to have failed, and the code under the `while` no longer runs. The following `while` loop writes the word *Warning!* three times:

```
short   count;

count = 0;

while ( count < 3 )
{
   DrawString( "\pWarning!" );
   count++;
}
```

Make sure you don't place a semicolon at the end of the line of code that contains the word `while` unless you want the loop to go forever.

Branching Statements

A loop runs the same code more than once. A branch runs code only once, but it allows the program to select the code from more than one grouping.

The switch statement

A `switch` branching statement begins with the word *switch* followed by a variable between parentheses. Groups of code appear underneath the `switch`. Each group begins with the word `case` and ends with the word `break`. The `switch` statement works by making a comparison between the value of the variable between the parentheses on the `switch` line and each of the values associated with the `case` labels. The following example gives a response based on the number of computers the user has:

```
short   numberOfComputers;

switch ( numberOfComputers )
{
   case 0:
      DrawString( "\pUsing someone else's, eh?" );
      break;
   case 1:
      DrawString( "\pSometimes one is one too many!" );
      break;
   case 2:
      DrawString( "\pLove chaos, huh?" );
      break;
}
```

Don't inadvertently add a semicolon to the end of the line of code that contains the word `switch`.

In many situations, you may be interested in just a few of many possible numbers, but you still want to acknowledge the other possibilities. C provides a catch-all provision that you can add to the `switch` branch — the `default` statement. After you add all the `case` labels you need, add the word *default.* Then add the code you want to run in the instances when none of the `case` conditions are met. In the following example, my code lets the user know that he or she wins if he or she gets the number 7 or 11. If the user gets any number other than those two numbers, the program writes the message, *Sorry, try again!*

```
short   diceTotal;

switch ( diceTotal )
{
   case 7:
      DrawString( "\p7 is a winner!" );
      break;
   case 11:
      DrawString( "\p11 is a winner!" );
      break;
   default:
      DrawString( "\pSorry, try again!" );
      break;
}
```

The *if* statement

You use the `switch` branch when you want your program to be able to choose from two or more possibilities. For situations where you only want your program to run a section of code or not run it, use the `if` branch instead. Here's an example:

```
short   score;

if ( score < 70 )
{
   MoveTo( 30, 50 );
   DrawString( "\pSorry, you need at least 70% to pass." );
}
```

The `if` branch uses a test condition to determine whether the code under the `if` should run or not run. In the preceding example, the `if` examines the value of the variable `score` to see whether it is less than 70. If it is, the two lines of code between the braces run. If `score` isn't less than 70, the program skips the lines and doesn't write any message.

The `if` branch can also include a second part — an `else` section. If the test condition fails, the code under the `else` runs. In this example, if the score is less than 70, the first block of code, the code directly under the `if` test, runs. If the score is greater than or equal to 70 (that is, should the `if` test fail), then the second block of code, the code directly under the `else`, runs instead of the first block:

```
short   score;

if ( score < 70 )
{
   MoveTo( 30, 50 );
   DrawString( "\pSorry, you need at least 70% to pass." );
}
else
{
   MoveTo( 30, 50 );
   DrawString( "\pCongratulations, you passed the test!" );
}
```

Notice that no semicolon appears at the end of the line that includes the word `if`. Putting one there is a no-no; a semicolon would cause the code that follows the `if` test to always execute, regardless of whether the test passes or fails.

Toolbox Functions

See Appendix B!

Appendix B

Toolbox Reference

After typing in all the programs listed in this book, you'll be proficient at using Toolbox functions. But there will still be times when you can't remember the exact spelling of one, or how many parameters get passed to it. Look here when these situations arise.

Initialization

Initialization means to set up, or give initial values, to certain variables or data. Initialization can pertain to many things, but in Mac programming, it generally means initializing things in the Toolbox.

When a program runs, the Toolbox needs a little information about it. This is called *Toolbox initialization.* Fortunately, the Toolbox is capable of initializing itself. For example, the Toolbox function `InitGraf` initializes graphics.

Every program you write should include eight Toolbox initialization calls. If you call the following eight functions, in the order shown, the Toolbox will be properly initialized. Call the functions just after declaring your program's variables. The parameters for each call are listed below:

```
InitGraf( &qd.thePort );
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();
```

Events

An action such as a mouse click is called an *event.* The Macintosh stores information about events as they occur. The Toolbox can help you get information about events so that your program can respond appropriately.

Your program can get information for the most recent event by calling WaitNextEvent. The first parameter to WaitNextEvent tells your program to keep a watch for every type of event. Always use everyEvent. The second parameter is an EventRecord variable that holds the information about the event. This parameter must be preceded by an ampersand character. The third and fourth parameters are used when other programs are running concurrently with yours and for special cursor-handling work. For simplicity, set the third parameter to 7 and the fourth parameter to nil.

```
EventRecord   theEvent;

WaitNextEvent( everyEvent, &theEvent, 7, nil );
```

Windows

Macintosh means windows. Not much happens on the screen without at least one window present. The Toolbox does much of the work of displaying, moving, and closing windows.

Opening and displaying a window

To open and display a new window, call GetNewWindow. The traits of the window — its type, size, and initial screen location — are found in a 'WIND' resource. The first parameter to GetNewWindow is the resource ID of this 'WIND' resource. The second and third parameters are used for window memory storage and positioning. Always use the two values shown in the next example.

After opening the window, GetNewWindow returns a WindowPtr to your program. This WindowPtr variable can then be used in future Toolbox calls such as SetPort and DisposeWindow.

```
WindowPtr   theWindow;

theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
```

Closing a window

To remove a window from the screen, call DisposeWindow. To let the Toolbox know which window to close, pass the WindowPtr variable that was returned to your program by GetNewWindow.

```
WindowPtr   theWindow;

theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
DisposeWindow(theWindow);
```

Moving a window

Call DragWindow when your program receives a mouseDown event in a window's drag bar. DragWindow does all of the work of moving the window around the screen as the user drags the mouse. The first parameter to DragWindow is a WindowPtr to the window to drag. Use a call to FindWindow to get this value. The second parameter is the screen coordinates where the mouse click took place. The third parameter tells the Toolbox what part of the screen it can use to drag the window. Passing a value of &qd.screenBits.bounds tells the Toolbox to use the entire screen. (Don't forget to precede this third parameter with an ampersand.)

```
EventRecord   theEvent;

WindowPtr    whichWindow;
DragWindow(whichWindow,theEvent.where,&qd.screenBits.bounds);
```

Responding to the Mouse Button

When the user clicks the mouse button, your program wants to know where the click occurs. Did the user have the cursor positioned over a window? Over the menu bar? This information is vital if you want your program to respond in an appropriate and logical manner. The Toolbox gives you this information in one easy-to-use function call.

When a mouseDown event happens, respond by calling FindWindow to determine in what part of the screen or window the mouse click occurs. The first parameter to FindWindow holds the screen coordinates where the mouse button is pressed. The second parameter is a pointer to the window that was clicked in — if one was. The second parameter is *not* the pointer that is returned by the Toolbox in a call to GetNewWindow. Instead, it is a different WindowPtr variable created just for this purpose. It gets its value from the Toolbox — the Toolbox places a value in this variable when it is finished running FindWindow. This second parameter must be preceded by an ampersand character.

After the call to FindWindow is complete, the variable thePart holds a value that represents the part of the screen or window that was clicked on.

```
EventRecord  theEvent;
WindowPtr    whichWindow;
short        thePart;

thePart = FindWindow( theEvent.where, &whichWindow );
```

Menus

What's a Mac program without at least one menu? The Toolbox helps you set up a menu bar and handle a user's menu selection.

Displaying menus and the menu bar

To let your program know which 'MBAR' resource it should use as the basis for a menu bar, call GetNewMBar. Pass this function the resource ID of the 'MBAR' that's in your program's resource file. In return, GetNewMBar provides your program with a Handle that you use in a call to SetMenuBar.

```
Handle  menuBarHandle;

menuBarHandle = GetNewMBar( 128 );
```

To help your program set up the menu bar with the appropriate menus, call SetMenuBar. Pass the Handle obtained from the preceding call to GetNewMBar as the only parameter.

```
Handle  menuBarHandle;

SetMenuBar( menuBarHandle );
```

Neither GetNewMBar or SetMenuBar actually display the menu bar on the screen. To do this, call DrawMenuBar. This function requires no parameters, but you still need to include a pair of parentheses after the function name.

```
DrawMenuBar();
```

Responding to a mouse click in the menu bar

In response to a click in the menu bar, call MenuSelect. This function does the work of tracking the mouse as the user moves it about the menu bar. MenuSelect displays and hides menus as the user moves over their names in the menu bar. The only parameter MenuSelect requires is the screen coordinates where the mouse button is first clicked in the menu bar. When the user makes a menu selection, MenuSelect returns information about the selected item in the long variable menuAndItem. Use the Toolbox functions HiWord and LoWord to extract information from this variable.

```
EventRecord   theEvent;
long          menuAndItem;

menuAndItem = MenuSelect( theEvent.where );
```

When the user makes a menu selection, MenuSelect highlights the name of the menu in the menu bar. When your program has finished handling the menu item, call HiliteMenu to return the menu name to its original state of black text on a white background.

```
HiliteMenu( 0 );
```

Determining which menu item is selected

MenuSelect returns a number that represents both the menu and the menu item. This information is stored in the long variable menuAndItem. To extract just the number of the menu from this variable, call HiWord. Pass menuAndItem as the only parameter. When the function is complete, HiWord returns the resource ID of the 'MENU' resource from which the menu selection was made.

```
Long    menuAndItem;
short   theMenu;

theMenu = HiWord( menuAndItem );
```

As mentioned in the description of HiWord, MenuSelect returns a number that represents both the menu and the menu item. To extract just the number of the menu item from this variable, call LoWord. Pass menuAndItem as the sole parameter. When the function is complete, LoWord returns a number that tells your program which menu item was selected. The first menu item is number 1, the second item is number 2, and so forth.

```
long    menuAndItem;
short   theMenuItem;

theMenuItem = LoWord( menuAndItem );
```

QuickDraw

Programming is like life — you have many chores that have to be taken care of, but you should also have time for fun. Drawing is the fun part of programming. The Toolbox provides you with some help to make even this fun part of programming easier. QuickDraw is the name of the set of Toolbox functions that performs text, line, and shape drawing in the windows of your programs.

Setting up ports

Every window has a port, and all drawing takes place in the port. After using the Toolbox function GetNewWindow to create and display a window, make its port the current, or active, port. Make a call to SetPort to do this. Pass SetPort one parameter, and make it the WindowPtr of the window to draw to.

```
WindowPtr   theWindow;

theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );
```

Moving to a location

The MoveTo function moves, without drawing, to a location in a window. The first parameter is the pixel distance in from the left of the window, the second parameter is the pixel distance down from the top of the window.

```
MoveTo( 20, 50 );  /* move 20 pixels in, 50 pixels down */
DrawString( "\pText" ); /* draw some text  */
```

Drawing a line

Unsurprisingly, the Line function draws a line. The first parameter tells how many pixels to the right the line should go, and the second parameter tells how many pixels down the line should go. The starting point of the line is determined by a call to MoveTo.

```
MoveTo( 20, 50 );  /* move 20 pixels in, 50 pixels down */
Line( 200, 10 ); /* draw a line 200 pixels across, 10 down */
```

Drawing a shape

Before the program can draw a rectangle, its boundaries must be established. SetRect sets up the coordinates of the Rect variable named as the first parameter. SetRect does not draw a rectangle. You use FrameRect or FillRect to do that. The first parameter to SetRect must be preceded by an ampersand character. The last four parameters are the pixel coordinates of the rectangle. The ordering of the last parameters is the left, top, right, and bottom of the rectangle.

```
Rect   theRect;

SetRect( &theRect, 20, 40, 300, 150 );
FrameRect( &theRect );
MoveTo( 20, 50 );   /* move 20 pixels in, 50 pixels down */
```

After setting the boundaries of a rectangle using SetRect, that same rectangle can now be framed using a call to FrameRect. Pass FrameRect a Rect variable as the only parameter. The parameter *must* be preceded by an ampersand character.

```
Rect   theRect;

SetRect( &theRect, 10, 10, 100, 100 );
FrameRect( &theRect );
```

After setting the boundaries of a rectangle using SetRect, that same rectangle can now be filled with a pattern using a call to FillRect. Pass FillRect a Rect variable as the first parameter. The first parameter must be preceded by an ampersand character. The second parameter to FillRect is the pattern that the Toolbox uses to fill in the rectangle. The available Toolbox-defined patterns are:

```
white, ltGray, gray, dkGray, black
```

Patterns must use capitalization as shown above. As you can see in the following examples, a pattern must also be preceded by an ampersand (&) plus the letters *qd* and a period.

```
Rect   theRect;

SetRect( &theRect, 10, 10, 100, 100 );
FillRect( &theRect, &qd.white );    /* white rectangle      */
FillRect( &theRect, &qd.ltGray );   /* light gray rectangle */
FillRect( &theRect, &qd.gray );     /* gray rectangle       */
FillRect( &theRect, &qd.dkGray );   /* dark gray rectangle  */
FillRect( &theRect, &qd.black );    /* black rectangle      */
```

You had to have known that rectangles aren't the only shape you can draw!

To draw an oval, first set up a rectangle using `SetRect`. Follow the call to `SetRect` with a call to `FrameOval`. This function draws the frame of an oval, neatly inscribed just within the boundaries established by `SetRect`. The one and only parameter to `FrameOval` is a `Rect` variable that *must* be preceded by an ampersand.

```
Rect   theRect;

SetRect( &theRect, 20, 20, 80, 100 );
FrameOval( &theRect );
```

`FillOval` works just like `FillRect`, except that it (of course) fills in an oval. After setting the boundaries of a rectangle using `SetRect`, call `FillOval` to fill an oval that fits into the rectangle with a pattern. Pass `FillOval` a `Rect` variable as the first parameter. The first parameter must be preceded by an ampersand character. The second parameter is the pattern that the Toolbox uses to fill in the oval. `FillOval` offers the same predefined patterns used by `FillRect`:

```
white, ltGray, gray, dkGray, black
```

Use the capitalization shown above for all patterns.

```
Rect   theRect;

SetRect( &theRect, 100, 100, 150, 150 );
FillOval( &theRect, &qd.ltGray );
```

Drawing text

On a Macintosh, text is drawn — not written. To write a letter, word, or sentence, use `DrawString`. The only parameter `DrawString` needs is the text to draw. Always enclose the text in double quotes, and precede the text with `\p`. Where the text appears depends on the previous call to `MoveTo`.

```
MoveTo( 35, 60 );/* here's where text will start  */
DrawString( "\pTest text" );   /* draw some text */
```

Appendix C

If Something Should Go Wrong . . .

With the crystal-clear explanations and examples presented in this book, how could anything possibly go wrong in your programming endeavors? I'm sure it won't, but you may still want to browse through this appendix just for fun!

Errors While Trying to Compile Your Code

Compilers are sticklers for detail and are quick to complain when they find something out of place. The CodeWarrior compiler is no exception. When you choose Compile, Make, or Run from the Project menu, you may end up seeing an error message listed in a window titled Errors & Warnings. If that happens, look through the headings of this section to find the error message — and the means to correct the mistake.

TIP

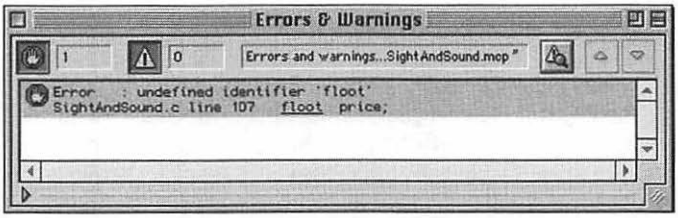
The Errors & Warnings window may give you bad news, but it does its best to help you out. You can double-click anywhere on an error message in this window and CodeWarrior brings the window holding your source code file to the front and scrolls to the line of code that holds the error.

The Compile menu item is dim

You can't compile your source code if the Compile menu item can't be selected! CodeWarrior doesn't know what to compile if your source code isn't open, or if the source code file isn't highlighted in the project window. If your source code file has been added to the project file, that's great. But now you have to either open it by double-clicking the file's name in the project window or highlight it by clicking once on its name in the project window. After you open or highlight the source code file, you can then check the Project menu to see that the Compile option is now enabled.

Declaration syntax error

When CodeWarrior encounters a word it doesn't recognize, it calls it a *declaration syntax error.* When that happens, you see a message like this in the Errors & Warnings window:



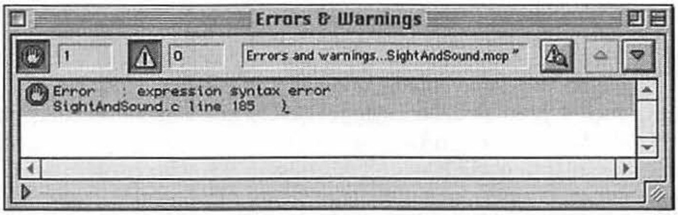
```
Error   : undefined identifier 'floot'
SightAndSound.c line 107   floot  price;
```

A few things can cause the CodeWarrior compiler to issue an error of this type:

- A C word that is misspelled.
- A C word that uses incorrect uppercase or lowercase.
- A word that is not part of the C language.

Expression syntax error

Always remember the age-old words of the wise old computer man: For every opening brace there must be a closing brace. If you get an error message like the one pictured here, then your braces don't match up.



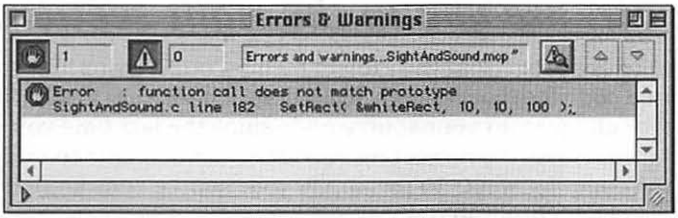
```
Error   : expression syntax error
SightAndSound.c line 185   }
```

By omitting a brace, you create a syntax error in a branching or looping section of your code. Double-click the error message to move to the location near the error in your source code file. Once there, hunt for a missing brace — from my experience, I can tell you that it's most likely a closing one. Here's an example of code that's missing a brace:

```
while ( count < 10 )
{
   DrawString( "\pTest " );
   count++;
MoveTo( 20, 40 ); /* should be a brace before this line */
DrawString( "\pEnd test" );
```

Function call does not match prototype error

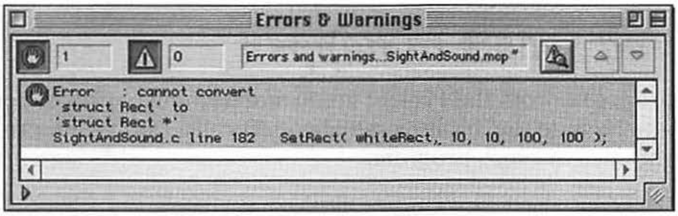
Many Toolbox functions require one or more parameters to be passed to them. CodeWarrior is aware of just how many parameters should be passed to each Toolbox function. CodeWarrior also knows which data type each of these parameters should be. Talk about smart! But just as a very smart *person* can be annoyingly smart, so too can a *compiler* be obnoxiously smart — CodeWarrior won't ever let you forget just how knowledgeable it is! If you make a mistake in naming the parameters that go with a Toolbox function call, CodeWarrior rubs it in by displaying an error message similar to this one:



```
Error    : function call does not match prototype
SightAndSound.c line 182   SetRect( &whiteRect, 10, 10, 100 );
```

Cannot convert error

Certain mistakes you make in listing the parameters to a Toolbox function result in an error message that includes words about not being able to convert one thing to another — like this:



```
Error    : cannot convert
'struct Rect' to
'struct Rect *'
SightAndSound.c line 182   SetRect( whiteRect, 10, 10, 100, 100 );
```

This error results from a situation similar to the one mentioned in the previous section, "Function call does not match prototype error." If you haven't read that section, do so now.

One of the primary candidates for bringing on this error message is forgetting to include the & symbol before a parameter that requires it. In particular, Toolbox routines that require that a rectangle variable be passed in further require that this variable should be preceded by the ampersand symbol. A second mistake that can cause this error is messing up the parameter to the `DrawString` Toolbox function.

Errors While Trying to Run Your Code

After successfully compiling your source code, you want to test it by choosing Run from the Project menu. That runs your code and builds an application. That is, if all goes well. If it doesn't, check out the problems and solutions listed in this section.

First off, are you in the right section?

When you choose Run from the Project menu, CodeWarrior first compiles your source code file if necessary. What does *if necessary* mean? If you made a change, or changes, to your source code since the last time you chose Compile or Run from the Project menu, CodeWarrior wants to update things. That is, the compiler wants to recompile your source code so that your changes can be incorporated.

If recompiling your code opens the Errors & Warnings window, then your mistake involves something the compiler didn't like. Check under the first section in this appendix — not here. This section assumes that your code gets compiled successfully but something goes wrong as CodeWarrior tries to run the code. In those instances, you won't see the Errors & Warnings window.

Nothing seems to happen

If you choose Run from the Project menu and nothing happens, you may have forgotten to include the Toolbox initialization calls. Don't forget to include these eight lines in every program:

```
InitGraf( &qd.thePort );
InitFonts();
InitWindows();
InitMenus();
TEInit();
InitDialogs( nil );
FlushEvents( everyEvent, 0 );
InitCursor();
```

If you left one or more of these eight lines out of your code, then go back and insert any that you forgot.

A flickering alert and a frozen Mac

Look at the previous section — the one titled "Nothing seems to happen." If you forget the initialization calls and nothing happens, consider yourself lucky! It's more likely that you'll instead see an alert flickering on your Mac's screen.

You may encounter a few variations of this alert — it may have a small picture of a bomb in the upper-left corner, and it may have some words written in it. In any case, you need to restart your Mac to get out of the jam. If pressing the reset button on your Mac doesn't do anything, you may even need to unplug your Mac from the wall outlet. While this is all a bit inconvenient, you don't have to worry too much — you didn't break anything.

Forgetting to include the initialization calls near the start of your source code is one way to bring about such an alert. A second way to end up in this predicament is by making a mistake involving resources.

The numero uno resource-related mistake is failing to include a particular resource in your project's resource file. For instance, the Mac gets hopelessly confused if your program attempts to display a menu bar and there is no 'MBAR' resource in the resource file. If your project doesn't run correctly, open the resource file by double-clicking its name in the project window. Then verify that the file has the 'WIND', 'MENU', and 'MBAR' resources it needs.

The program runs and then quits immediately

What went wrong if you choose Run from the Project menu and you see your program's menu bar for a moment, and then your program ends? You probably forgot to give variable `allDone` a value, or you gave it the wrong value. That would mean your event loop got skipped and your program assumed it was time to pack up and leave. Here's the part of your source code to examine:

```
allDone = 0;   /* Must give allDone a value of 0 here */
while ( allDone < 1 )
{
   WaitNextEvent( everyEvent, &theEvent, 7, nil );
...
...
```

Link failed error

When you run your code, two things may happen. First, if the source code has been changed since the last time you compiled, CodeWarrior compiles it again. Second, CodeWarrior links the code. That just means that your source code is merged with your project's resources and with whatever code is in the various libraries that CodeWarrior placed in the project. If all goes well, your program runs on the screen. If all *doesn't* go well, you may see the Errors & Warnings window. In this window could be any number of different error messages, but if they start with the words *Link Error,* then you know the problem probably isn't with your source code.

A link failed error means that CodeWarrior couldn't find some code it needs to complete the program. The most common cause for this is that you inadvertently removed a library from the project. When you create a new project, CodeWarrior adds several libraries to it. These libraries hold code that CodeWarrior uses in conjunction with your own source code. If you click one of these library names in the project window and then select Remove from the Project menu, that library is deleted from the project. That's something you don't want to do. Don't worry, though, the library is still around on your hard drive — it just isn't a part of your project.

Because the preceding scenario would occur unintentionally, it's pretty unlikely that you know which library you removed. The solution? Close the project and create a new one. That's not quite as bad as it sounds. You don't have to create a new source code file or resource file. Just add these same files to the new project, and you're all set.

Errors While Running Your Code

So your program is compiled and is running — that's great news! But wait a minute. You say it's not doing something that it should be doing? Check this section for answers to problems of this nature.

Things aren't getting drawn in the window

If your program is supposed to be drawing text or graphics to a window, but it's not, you may have omitted the call to SetPort. Make sure that the call appears after the call to GetNewWindow, like this:

```
theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );
SetPort( theWindow );
```

A rectangle that should be there just ain't there

When drawing a rectangle, don't forget to first set up the coordinates for the rectangle. Without a call to SetRect, a call to FrameRect or FillRect won't work.

So you insist that you did call SetRect, but the rectangle still hasn't appeared? Double-check the order of the coordinates you passed to SetRect. Here's how SetRect views the four numbers you give it:

```
SetRect( &theRect, left, top, right, bottom );
```

What do you suppose would be the result of this code:

```
SetRect( &theRect, 10, 100, 50, 20 );
FrameRect( &theRect );
```

Absolutely nothing! Why? Because the fourth number, 20, is less than the second number, 100. You're essentially asking the bottom of the rectangle to appear *above* the top of it, and that's something SetRect can't do.

Errors Not Addressed in This Appendix

If you come across a problem not addressed on these pages, what should you do? Here are a few ideas:

- ✓ Leaf through the index, looking for a word that at least partially describes your predicament — I may have addressed the problem somewhere in this book.

- ✓ Post your question in an Internet newsgroup — `comp.sys.mac.programmer.codewarrior` and `comp.sys.mac.programmer.help` are excellent sources of help for new programmers. Make sure you read the newsgroup's FAQ first to check that the question hasn't already been asked and answered.
- ✓ If you have another Mac programming book — even one you find too advanced — look through its index and table of contents. Even if much of the book isn't for you, perhaps it contains a page or two that may help.

Appendix D

Glossary

argument: The same as a *parameter* — a variable or value passed to a function.

branch: Source code that allows a program to follow just one of two or more paths. The `switch` and the `if` statements are examples of C branches.

Central Processing Unit (CPU): The computer chip that serves as the brains of a computer.

comparative operator: A symbol that compares the value on the operator's left side with the value on its right side. The less-than operator (<) is an example. These operators are also referred to as *relational operators.*

compiler: The software program that turns source code into code that a computer can understand.

coordinate system: The means of identifying every pixel on a monitor.

counter: A variable used to control the number of times a loop runs.

decrement: To decrease the value of a variable by one. The decrement operator is represented by two minus signs (--).

desktop: The area of the screen that holds icons such as folders and the trash can.

dialog box: A special type of window that contains items — such as buttons — that allow a program user to communicate with the program.

event: Each action a program user takes is an event. A click of the mouse or a press of a key are each events. The Toolbox data type `EventRecord` holds information about an event.

event loop: A section of source code that repeatedly watches and responds to events as they occur. The event loop runs until the program ends.

floating-point number: A number that has a decimal point. The numbers 5.24, 9200.0, and -72.3 are examples. The C data type for a floating-point number is the `float` type.

function: A group of source code that serves a specific task. That is, a group of source code that performs a single function. Functions can also be referred to as *routines.*

graphical user interface (GUI): The user interface of a computer helps the user communicate with the computer. When the user interface contains graphical elements such as icons, menus, and windows, it is said to be a graphical user interface.

icon: A small image that represents something. The trash can on the desktop is an example.

increment: To increase the value of a variable by one. The increment operator is represented by two plus symbols (++).

initialization: To give something a value for the first time. The group of functions that give the Toolbox some initial values at the start of a program are called Toolbox initialization functions.

integer: A whole number — a number with no decimal point. The numbers 7, 8214, and –42 are examples. The C data type for an integer is the `int` type.

loop: A group of source code that repeats itself. The `while` statement is an example of a C loop. The C language provides programmers with other types of loop statements as well, including the `for` statement. (This book only describes the `while` statement.)

menu: A part of the graphical user interface that allows the computer user to make selections from a list of choices.

menu bar: A collection of menus in a program. The menu bar is located at the top of the screen.

'MBAR': A type of resource that represents a menu bar.

'MENU': A type of resource that represents a single menu.

operating system: Software that controls the very basic activities of a computer, such as copying files.

operator: A symbol that is used to perform an operation on variables or values. An example is the addition operator (+), which adds two variables or values together.

parameter: A variable or value passed to a function. Also referred to as an *argument.*

pixel: The smallest dot on the screen. Because there are approximately 72 pixels in a one-inch line, you may not be able to see a pixel. But the Mac knows all about them — the Mac builds, or defines, images by adjusting the color of a number of pixels.

pop-up menu: A menu that doesn't appear in the menu bar. Usually found in a dialog box.

port: The means of identifying which window should be drawn to. A port is also referred to as a *GrafPort* or a *graphics port.*

project file: A CodeWarrior file that holds information about the contents of a project. The project file holds the name of the source code files, resource file, and libraries used to create a single program.

real number: *See* floating-point number.

rectangle: A commonly and easily drawn graphics shape. The Toolbox data type that holds information about a rectangle is the `Rect` type.

relational operator: *See* comparative operator.

resource: Information that defines one part of the graphical user interface, such as a menu or window. Resources are used in conjunction with source code to form programs.

source code: A series of statements, or instructions, written in a computer language such as C.

string: A letter, word, or group of words. In C, a string to be used by the Toolbox is preceded by the *\p* characters and enclosed in quotes, as shown here: `\pExample string`.

Toolbox function: A collection of functions written by Apple and stored in the ROM chips and System file in the Macintosh.

type: Every variable has a type — a category that defines what kind of data the variable can hold.

variable: A piece of code used to store a value.

'WIND': A type of resource that represents a window.

window: The object that is used to display text and graphics. A window can usually be opened, moved, and closed by the user.

Appendix E

iMac Programming and Movie Playing

In This Chapter

- Understanding how to play QuickTime movies
- Discovering the Movie Toolbox functions
- Looking at a movie-playing example program
- Adding movie-playing features to your own programs

At this writing, well over 800,000 iMacs have been sold — and that's just four months after its introduction. It's a safe bet that plenty of you — and plenty of the Mac users who may run your own program — have an iMac. In Chapter 3 you read that among the chief features of the iMac is its speed — the iMac is fast. Speed is the key element that makes one really cool multimedia technology a reality. That technology is QuickTime movie-playing. Here I provide you with an introduction of how you can make your own program a movie-playing one. Although such a program will run on most Mac models, it will especially shine when run on a fast machine such as Apple's iMac.

Playing Movies

By now you're familiar with the Macintosh Toolbox, so you know it to be the huge set of Apple-written functions that give programmers the ability to add all sorts of features to their programs. In this book the focus is on the Toolbox functions that bring the basic features to a Mac program — features such as windows and menus. But the Toolbox provides numerous functions that allow you to add all sorts of goodies to your Mac programs. In Chapter 21 you get a hint of this when you see the code for programs that display a picture and play a sound. Another forte of the Toolbox is movie-playing — the Toolbox includes several functions that can be used to give a Mac program the capability to play QuickTime movies.

QuickTime is Apple's software technology that allows programs to play movies. Just about any Macintosh can play QuickTime movies, but there's a big variance in the movie-playing quality from one Mac model to another. The chief factor in how well a Mac plays movies is processor speed. If a Mac has a fast processor (CPU), then it plays movies nice and smooth. If a Mac has a slow processor, then a movie may appear jerky — there'll be pauses between frames or even skipped frames. In the past, entry-level Macs (lower-priced models aimed at those new to computing) tended to have slower processors, so playing QuickTime movies wasn't always a thrilling experience for owners of such computers. That's changed forever with the arrival of the iMac. There's no overlooking this fact: the iMac is *fast*. If a user of your program has an iMac, you're safe in assuming that the user can successfully and enjoyably view QuickTime movies. With that in mind, here's a quick introduction on how to write a Mac program that plays a movie.

TIP

To get a hold of some movies, you can buy commercial CD-ROMs that include QuickTime movies. You can also make your own. To do that, you connect a camcorder or VCR to your Macintosh. You then run the Apple Video Player program located under the Apple menu (it's included with most Macs, including the iMac). You can also download a number of QuickTime movies from the Internet — for example, Apple has many of its television commercials available for download in QuickTime format.

The Movie Toolbox

You're familiar with the Toolbox, the huge set of Apple-written functions that you call from within your own source code. A part of the Toolbox that I haven't explored in this book is called the Movie Toolbox. As its name implies, the functions in this part of the Toolbox exist to assist you in providing your own Mac programs with QuickTime movie-playing capabilities.

Just as you initialize other parts of the Toolbox, you need to initialize the Movie Toolbox. You do that by calling the EnterMovies function, like this:

```
EnterMovies();
```

When the Movie Toolbox is initialized, your program can call any of the dozens of Movie Toolbox functions. A detailed look at the use of this part of the Toolbox is beyond the scope of this book, but I provide you with a brief overview of some of the most important functions in the next few sections.

OpenMovieFile

A QuickTime movie exists in a file on disk (on a hard drive, floppy disk, CD, and so forth). Before a program can play the movie, the program needs to open the movie file. The `OpenMovieFile` function does just that.

NewMovieFromFile

The just-mentioned `OpenMovieFile` function opens a file, meaning the program is able to access the contents of the file. Accessing the file means placing the file's data, or information (the movie), into memory. That's how a program typically works with a file's data — it places the data in RAM. `NewMovieFromFile` is the Movie Toolbox function that gets movie data into memory where the program can make use of it.

CloseMovieFile

Even before a program plays the movie in an open movie file, the program typically closes the file. Why? Because the information that's needed has been placed in memory by the call to `NewMovieFromFile`.

SetMovieGWorld

To display a movie, a regular 'ole window is opened by calling the same `GetNewWindow` function that you're used to using. Then a call to the Movie Toolbox function `SetMovieGWorld` is made to associate, or link, the movie data in memory with the newly opened window.

GetMovieBox and SetMovieBox

It's possible to display only a part of a movie in a window. That is, you could tell a program to crop the boundaries of a movie frame so that the user sees only a portion of each frame as the movie plays. The Movie Toolbox functions `GetMovieBox` and `SetMovieBox` are used to do this. More importantly for us, these same functions can be used to ensure that the entire frame of a movie is displayed in a window. After that task is taken care of, the window that is to display the movie needs to be resized. QuickTime movies come in all sorts of sizes — a movie may be very small or it may fill the screen. When a program opens a window in which to play the movie, though, the window size may very well not match the movie size. A call to the Toolbox function `SizeWindow` resizes a window to match the size of a movie.

GoToBeginningOfMovie

I'm in the habit of watching a VHS tape and then forgetting to rewind it. It annoys the heck out of the next person who wants to watch the tape, but hey, nobody's perfect. QuickTime movies can be subject to the same kind of irresponsible behavior, too. Before a program starts playing a movie, it should call the Movie Toolbox `GoToBeginningOfMovie` to ensure that the movie starts from the beginning.

StartMovie and MoviesTask

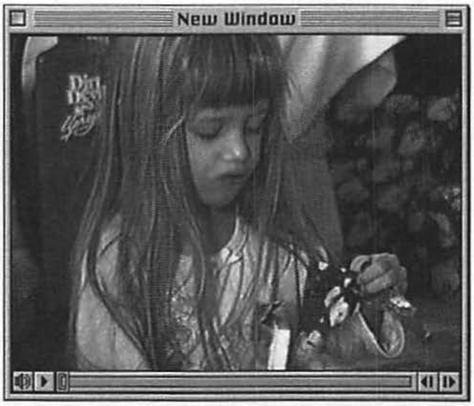
Finally — let's play the darn thing! The Movie Toolbox function `StartMovie` starts a QuickTime movie playing. But it doesn't keep it playing. The `StartMovie` function doesn't know how long the movie is (that is, it doesn't know how many frames the movie consists of), so it's necessary to repeatedly call another Movie Toolbox function, `MoviesTask`, to keep the movie going.

DisposeMovie

When a program is finished playing a movie it should free up, or release, the memory that holds the movie data. That allows the memory to be used for other purposes.

A QuickTime movie-playing example

As I mentioned earlier in this appendix, complete coverage of QuickTime movie playing requires a lot of pages. But I'd feel bad if I left you hanging without some kind of usable example. So of course I've provided such an example. The program, named QuickMovie, plays a QuickTime movie and then quits. In order to work properly, the movie *must* be in the same folder as the QuickMovie program, and the movie file *must* have the name XmasMovie. Double-clicking on the QuickMovie icon starts the program and opens a movie of a little girl opening a Christmas present:



Next comes the source code for the QuickMovie program. I haven't provided enough of an explanation to make all the code understandable, but it's still some very useful code. First, look it over and see how much you can understand. If you carry on with your Mac programming endeavors, pick up a more advanced Mac programming book and read up on QuickTime to get all the juicy movie-playing details. Before you do that, however, there's a way you can make the code work for you — even without understanding exactly how the code works. I'll tell you about that after you look over the code.

The QuickMovie program works by initializing the Toolbox, initializing the Movie Toolbox (by calling `EnterMovies`), and then by calling the function `PlayOneMovie`. This all happens from within the program's main function:

```
void  main( void )
{
   InitGraf( &qd.thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( nil );
   FlushEvents( everyEvent, 0 );
   InitCursor();

   EnterMovies();

   PlayOneMovie();
}
```

Now here's the `PlayOneMovie` function. You can get a general idea of how the function works by reading the Movie Toolbox function summary earlier in this appendix:

```
void  PlayOneMovie( void )
 {
   FSSpec          theFSSpec;
   Short          theFileRefNum;
   Movie          theMovie;
   Short          theMovieResID = 0;
   Str255          theMovieResName;
   Boolean        wasAltered;
   WindowPtr      theWindow;
   Rect           theMovieBox;
   unsigned long  theLong;

   FSMakeFSSpec( 0, 0, "\pXmasMovie", &theFSSpec );
   OpenMovieFile( &theFSSpec, &theFileRefNum, fsRdPerm );
   NewMovieFromFile( &theMovie, theFileRefNum,
                     &theMovieResID, theMovieResName,
                     newMovieActive, &wasAltered );
   CloseMovieFile( theFileRefNum );
```

(continued)

(continued)

```
    theWindow = GetNewWindow( 128, nil, (WindowPtr)-1L );

    SetMovieGWorld( theMovie, (CGrafPtr)theWindow, nil );

    GetMovieBox( theMovie, &theMovieBox );
    OffsetRect( &theMovieBox, -theMovieBox.left,
                -theMovieBox.top );
    SetMovieBox( theMovie, &ta"MovieBox );

    SizeWindow( theWindow, theMovieBox.right,
                theMovieBox.bottom, true );
    ShowWindow( theWindow);

    GoToBeginningOfMovie( theMovie );

    StartMovie( theMovie );

    do
    {
       MoviesTask(theMovie, 0);
    }
    while ( IsMovieDone( theMovie ) == false );

    Delay( 120, &theLong );

    DisposeMovie( theMovie );
    DisposeWindow( theWindow );
}
```

Even though I wrote the `PlayOneMovie` function, you can easily make use of it in your own program. Here's how. Begin by copying the entire function `PlayOneMovie` (its code exists in the QuickMovie.c file in the XE QuickMovie folder on this book's CD-ROM). Now paste it into your own program's source code file. At the top of your source code file include a function prototype — a line of code that tells the compiler a little bit about the function. That prototype looks like this:

```
void  PlayOneMovie( void );
```

In the program's main function, add a call to the Movie Toolbox initialization function `EnterMovies` — like this:

```
void  main( void )
{
    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( nil );
    FlushEvents( everyEvent, 0 );
    InitCursor();

    EnterMovies();

    [ other code goes here ]
}
```

Now include a call to the PlayOneMovie function. For instance, if I added the function prototype to the top of the Animator.c source code file in the Chapter 20 Animator example, I could call the PlayOneMovie function when the user selected, say, the Move Square menu item. Looking back at Chapter 20, you'll find the following code in the source code listing for the Animator program. This is the code that runs when the user chooses the third menu item (the Move Square item) from the program's one menu:

```
case 3:
   SetRect( &theRect, 0, 0, 400, 280 );
   EraseRect( &theRect );
   SetRect( &theRect, 10, 10, 60, 60 );
   count = 0;
   while ( count < 140 )
   {
      FrameRect( &theRect );
      OffsetRect( &theRect, 2, 2 );
      count++;
      Delay( 2, &theLong );
   }
   break;
```

Instead of moving a square across the window, this menu item could be made to play a QuickTime movie by replacing the preceding code with a call to PlayOneMovie:

```
case 3:
   PlayOneMovie();
   break;
```

If you look in the XE AnimatorWithMovie folder on this book's CD-ROM, you'll find a new version of the Chapter 20 Animator example. In this version I've added the previously mentioned changes so that the program now plays the Christmas movie when the user chooses the Move Square menu item.

What if you don't want your own program to play this particular movie of a little girl opening a Christmas present? What if you'd like your program to play a different movie? That's an easy one. Just change the name of the movie to play in the PlayOneMovie function to match the name of the QuickTime movie file to play (and make sure that movie file is in the same folder as your program. For example, if the movie to play was named MyVacation, then this line of code:

```
FSMakeFSSpec( 0, 0, "\pXmasMovie", &theFSSpec );
```

should be changed to this:

```
FSMakeFSSpec( 0, 0, "\pMyVacation", &theFSSpec );
```

Appendix F

What's on the CD-ROM?

In This Chapter

- Having CodeWarrior Lite on this book's CD-ROM
- Installing CodeWarrior Lite onto your hard drive
- Copying ResEdit and the *...For Dummies* Examples folder from the CD-ROM

You get both bad and good news when it comes to Mac programming. You want the bad news first? Okay, brace yourself. The bad news is that to create a Macintosh program, you need a compiler, which is the software that turns your source code into a living, breathing program. The good news is that I know you have one. I made sure that a copy of the CodeWarrior Lite compiler made it onto the CD-ROM that comes with this book.

You use a compiler quite a bit in Mac programming, and so I want to make absolutely sure that the compiler makes it from the CD-ROM to your hard drive. I also want to make sure that everything gets set up just right so that you and I are both confident that you can work with all the examples that appear in this book. In this chapter, I walk you through the process of getting the CodeWarrior Lite compiler and other files from the book's CD-ROM.

CodeWarrior Professional or CodeWarrior Lite?

CodeWarrior Lite is a trimmed-down version of the full-featured CodeWarrior Professional compiler. Although CodeWarrior Lite can't perform all the compiling tricks that CodeWarrior Professional can, it does allow you to try out each and every example on this book's CD-ROM.

If you haven't purchased CodeWarrior Professional, then this is really your lucky day. The CodeWarrior Lite compiler is just waiting for you on this book's CD-ROM. Make sure to read the next section, "Installing CodeWarrior Lite," to see how that's done. Then read the remainder of this appendix to make sure you copy a few other choice files from the CD-ROM to your hard drive.

For owners of the full-featured CodeWarrior

If you're fortunate enough to own CodeWarrior Professional and already have it installed on your machine, you may be tempted to skip the next section — or even the rest of this appendix. Please don't. Even though you own a copy of CodeWarrior, you may still want to install the Lite version on your hard drive. Here's why. All the examples for this book have been thoroughly tested using the current version of both CodeWarrior Professional and CodeWarrior Lite. If you own any other version of CodeWarrior, I can't guarantee everything will go perfectly smoothly. For experienced programmers, minor differences from one version of a compiler to another aren't too important. They just click a few buttons, or type a few extra words somewhere, and everything works out just fine. For beginners, it's best to guarantee complete compatibility, though.

Here's what I recommend you do. First, install this book's CD-ROM version of CodeWarrior Lite on your Mac's hard drive. Then try things out using CodeWarrior Lite. After that, when you have a little confidence, try things out again — this time with your full-featured version of CodeWarrior.

Installing CodeWarrior Lite

No, there's no way you can avoid using a compiler; in order to work with code and turn it into a program, you need a compiler. So if you haven't done so already, unpack the CD-ROM from the back of this book and, handling it carefully by its edges, set it in your Mac's CD-ROM drive. As with any CD-ROM disc, make sure the side with the writing on it goes in face-up.

Running the installers

After inserting the CD-ROM in the drive, a CD-ROM icon named CodeWarrior Lite appears on your Mac's desktop. Double-click this icon to open the window that lists the disc's contents. You see a window with the names of several files and folders listed in it. Right now, of most importance to you are the two *installers*. Between them, these two installers have over 1,000 files (yes, that's right, more than 1,000!) embedded within them. Running the two installers places copies of all these files on your hard drive. Here are the icons you need to look for:

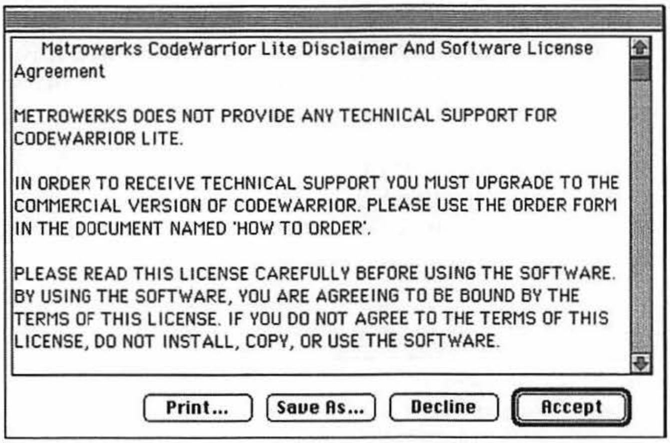
CW LITE C_CPP Installer

CW LITE IDE Installer

WARNING!

Metrowerks has set up the CodeWarrior Lite Installer programs such that they do all the work for you, and do it *correctly*. The installer creates one main CodeWarrior Lite folder on your hard drive, and then it keeps everything neat and tidy by placing everything it needs (including more folders and all those files) in this folder.

The file named CW LITE IDE Installer is the first of the two installers to run. Its purpose is to place a copy of the CodeWarrior Lite integrated development environment (IDE), along with a number of other files, on your hard drive. To run the installer, simply double-click the CW LITE IDE Installer icon. When you do that, a dialog box like this one appears:



Metrowerks CodeWarrior Lite Disclaimer And Software License Agreement

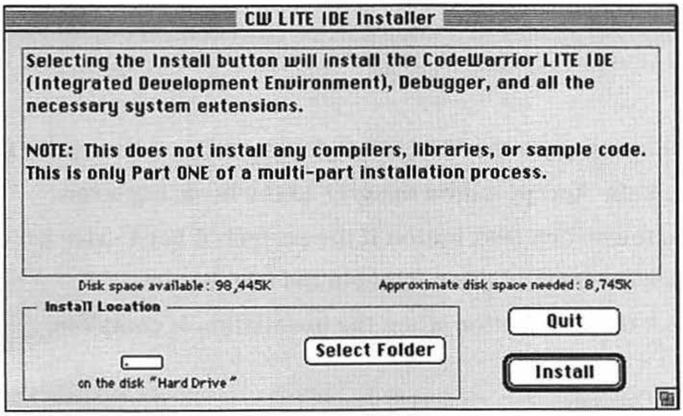
METROWERKS DOES NOT PROVIDE ANY TECHNICAL SUPPORT FOR CODEWARRIOR LITE.

IN ORDER TO RECEIVE TECHNICAL SUPPORT YOU MUST UPGRADE TO THE COMMERCIAL VERSION OF CODEWARRIOR. PLEASE USE THE ORDER FORM IN THE DOCUMENT NAMED 'HOW TO ORDER'.

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, DO NOT INSTALL, COPY, OR USE THE SOFTWARE.

Scroll through the dialog box text and then click the Accept button to agree to give up your firstborn. No, no, all you're really being asked to do is use Metrowerks' software in good faith. That is, don't attempt to sell it, distribute it yourself, or do anything else naughty with this program that Metrowerks is so kindly providing you with.

After clicking the Accept button, a dialog box with a few buttons appears:



Selecting the Install button will install the CodeWarrior LITE IDE (Integrated Development Environment), Debugger, and all the necessary system extensions.

NOTE: This does not install any compilers, libraries, or sample code. This is only Part ONE of a multi-part installation process.

Disk space available: 98,445K

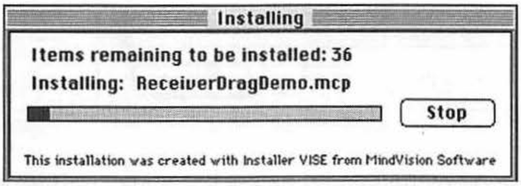
Approximate disk space needed: 8,745K

Install Location

on the disk "Hard Drive"

If you have more than one hard drive, the dialog box won't look just like the one in the previous figure. In place of the Select Folder button there will be a Switch Disk button and a pop-up menu. These items let you choose where to install CodeWarrior Lite. Typically, programmers keep their programming tools on their startup, or main, hard drive. Before installing, glance at the hard drive icon pictured in the lower left of the install dialog box. The name under it will most likely be the name of your startup hard drive. That's the drive I recommend you install CodeWarrior Lite on. If you insist on ignoring my advice, use the Switch Disk button or the pop-up menu to change the drive. Playing with either this button or this pop-up menu won't actually install any files, so don't worry if you accidentally end up with the wrong drive name under the hard drive icon. Just select a different drive. When you are finally content with the hard drive name under the hard drive icon, all you need to do is click the Install button to kick things off.

Now, here's the really great part of installing CodeWarrior Lite: You get to take a break from programming before you even begin to program! Installation takes a couple of minutes. You can gauge how much of a break you get by glancing at the progress indicator on your screen:



Items remaining to be installed: 36

Installing: ReceiverDragDemo.mcp

This installation was created with Installer VISE from MindVision Software

When installation is complete, a new dialog box appears. Click the Quit button to confirm that you're all finished with the installer program.

If you're a little intimidated by the thought of having to actually start programming, you're in luck. You get to delay that act just a bit longer! You've run one installer, but you still need to run the second one — the one named CW LITE C_CPP Installer. As it's name hints, this installer places on your hard drive a number of files that CodeWarrior Lite needs in order to work with C and C++ source code. Now that you've run the first installer, you know the drill:

1. **Double-click on the CW LITE C_CPP Installer icon to start the installer.**
2. **Click the Accept button to agree to the licensing terms.**
3. **Use the Switch Disk button if the preferred hard drive isn't named.**
4. **Click the Install button to begin the installation.**
5. **Click the Quit button when the installation is complete.**

When you run the *second* installer (CW LITE C_CPP Installer), you'll want to specify the same hard drive as you did when you ran the *first* installer (CW LITE IDE Installer). The running of the first installer created a new CodeWarrior Lite folder on the hard drive you specified, and you want all the files installed by the second installer to end up in this same CodeWarrior Lite folder. Don't worry about trying to tell the second installer anything about the exact location of the CodeWarrior Lite folder — just make sure the same hard drive is specified during the running of both installers.

Checking to see if the installation worked

After the installers run, you'll want to convince yourself that the installation was successful. Look on the hard drive you specified as the destination for CodeWarrior Lite and see whether you find a folder named CodeWarrior Lite. When you find it, double-click it to satisfy yourself that there's a bunch of files and folders inside! In particular, you'll want to peek in the Metrowerks CodeWarrior folder to see if there's a file named CodeWarrior Lite 3.0 in it. This file is the CodeWarrior Lite IDE — double-clicking on it starts up CodeWarrior Lite.

Installing Other Files from the CD-ROM

Metrowerks set up the CodeWarrior Installer program so that you'd have an easy way to copy all those CodeWarrior-related files and folders to your hard drive. The installer does that task quite well. But it doesn't copy everything from the CD-ROM to your hard drive. It leaves a few goodies on the disc. It's up to you to decide if you want any of this other stuff on your hard drive. For working along with this book, you should definitely copy two other things:

- ResEdit, the resource editor program.
- *...For Dummies* Examples folder, which holds all of the examples from this book.

Copying ResEdit to your hard drive

Just about every Mac program uses resources. You use a resource editor, such as Apple's ResEdit, to create and edit these resources (see Chapter 7 for details on ResEdit). Where can you get this vital piece of programming equipment? Call the neighbors and wake up the dog! You've got a fully functional version of ResEdit right on this book's CD-ROM. And if you've already installed CodeWarrior Lite, or if you have a folder devoted to programming on your hard drive, you can copy ResEdit right into either one of those folders. If for some reason you already have ResEdit on your hard drive, now is the time to drag it into your main CodeWarrior folder. For users of CodeWarrior Lite, that would be your CodeWarrior Lite folder.

If you're pretty sure that you have a copy of ResEdit tucked away in a folder on your hard drive already, but you don't remember where you put it, don't worry. Go ahead and copy the version from this book's CD-ROM. The version on the CD-ROM is the most recent as of this writing, so you'll know you're not using a dated version.

If you want ResEdit, copy the entire ResEdit 2.1.3 folder from the CD-ROM to your hard drive. While it can go anywhere on your hard drive, I suggest you copy the folder into your main CodeWarrior folder by dragging the ResEdit 2.1.3 folder from the CD-ROM to this folder. For you CodeWarrior Lite users, this main folder is named CodeWarrior Lite. Now you have a copy of ResEdit on your hard drive all ready for creating and editing resources.

Copying the ...For Dummies Examples folder

To save you some effort, I've taken the time to type in all the source code for all the examples in this book. I know, I know. You can thank me later. I've saved the fruits of my labor in a number of files, all of which you find in a folder titled *...For Dummies* Examples on the CD-ROM. To make use of these files, you need to copy them from the CD-ROM to your hard drive. The easiest way to copy these files is to copy the entire folder to your main CodeWarrior folder. All you have to do is click and drag the *...For Dummies* Examples icon from the CD-ROM to your hard drive. To make it easy to run the examples, I suggest you drag the entire *...For Dummies* Examples folder from the CD-ROM to your main CodeWarrior folder (the CodeWarrior Lite folder for you CodeWarrior Lite users).

You're All Set

Now you're all set to try out the examples from the book. If you haven't already done so, start reading. No, don't turn the page — jump all the way to the start of the book! When you get to the first example project, you'll be all set to give it a whirl. . . .

Index

(continued)

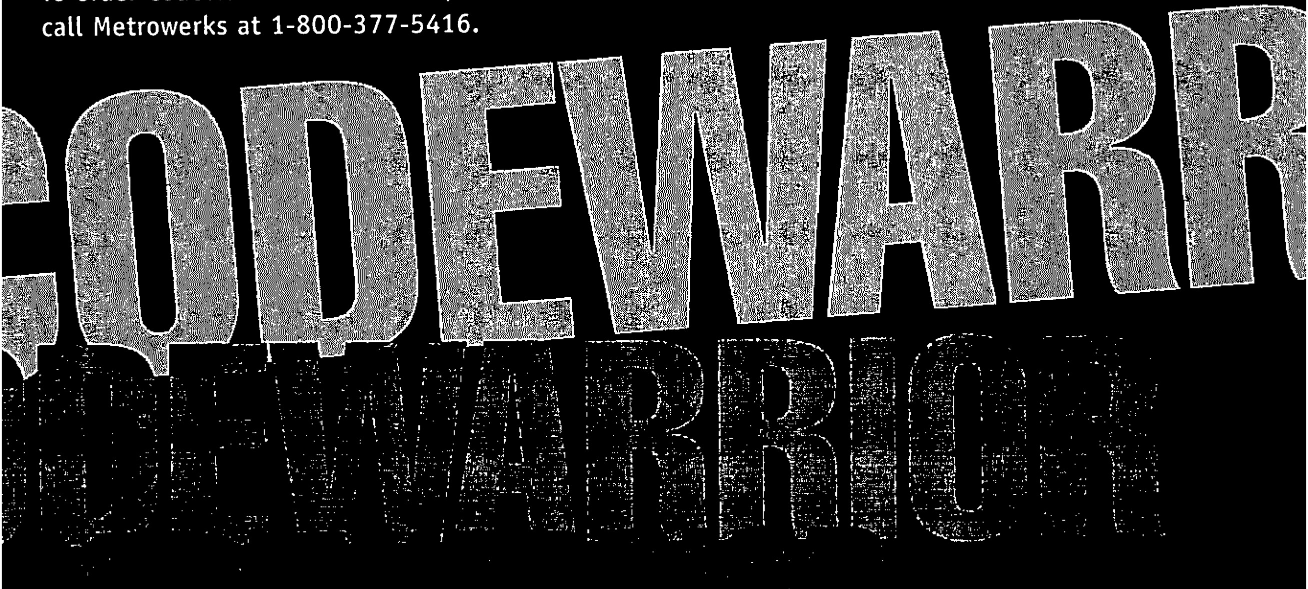
(continued)

(continued)

(continued)

(continued)

(continued)



CODEWARR
CODEWARRIOR

IDG Books Worldwide, Inc., End-User License Agreement

READ THIS. You should carefully read these terms and conditions before opening the software packet(s) included with this book ("Book"). This is a license agreement ("Agreement") between you and IDG Books Worldwide, Inc. ("IDGB"). By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

1. **License Grant.** IDGB grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the "Software") solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multiuser network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). IDGB reserves all rights not expressly granted herein.
2. **Ownership.** IDGB is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the disk(s) or CD-ROM ("Software Media"). Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with IDGB and its licensers.
3. **Restrictions on Use and Transfer.**

   (a) You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.

   (b) You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.
4. **Restrictions on Use of Individual Programs.** You must follow the individual requirements and restrictions detailed for each individual program in Appendix F of this Book. These limitations are also contained in the individual license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you will be agreeing to abide by the licenses and restrictions for these individual programs that are detailed in Appendix F and on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.

5. **Limited Warranty.**

   (a) IDGB warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of purchase of this Book. If IDGB receives notification within the warranty period of defects in materials or workmanship, IDGB will replace the defective Software Media.

   (b) **IDGB AND THE AUTHOR OF THE BOOK DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. IDGB DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.**

   (c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

6. **Remedies.**

   (a) IDGB's entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to IDGB with a copy of your receipt at the following address: Software Media Fulfillment Department, Attn.: *Mac Programming For Dummies,* 3rd Edition, IDG Books Worldwide, Inc., 7260 Shadeland Station, Ste. 100, Indianapolis, IN 46256, or call 800-762-2974. Please allow three to four weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

   (b) In no event shall IDGB or the author be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising from the use of or inability to use the Book or the Software, even if IDGB has been advised of the possibility of such damages.

   (c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

7. **U.S. Government Restricted Rights.** Use, duplication, or disclosure of the Software by the U.S. Government is subject to restrictions stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, and in subparagraphs (a) through (d) of the Commercial Computer–Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, when applicable.

8. **General.** This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.

Installation Instructions

The *Mac Programming For Dummies* CD-ROM contains software and files you can use to learn how to program your Mac. The installation instructions for each of these tools is too long to summarize here. For installation instructions for the programs and files, just turn to Appendix F at the end of this book.

YOUR
ONLINE
RESOURCE
WWW.DUMMIES.COM

SURF
THE
NET
WWW.DUMMIES.COM

IDG BOOKS WORLDWIDE BOOK REGISTRATION

Register This Book and Win!

We want to hear from you!

Visit **http://my2cents.dummies.com** to register this book and tell us how you liked it!

- Get entered in our monthly prize giveaway.
- Give us feedback about this book — tell us what you like best, what you like least, or maybe what you'd like to ask the author and us to change!
- Let us know any other *...For Dummies*® topics that interest you.

Your feedback helps us determine what books to publish, tells us what coverage to add as we revise our books, and lets us know whether we're meeting your needs as a *...For Dummies* reader. You're our most valuable resource, and what you have to say is important to us!

Not on the Web yet? It's easy to get started with *Dummies 101*®*: The Internet For Windows*® *98* or *The Internet For Dummies*®, 5th Edition, at local retailers everywhere.

Or let us know what you think by sending us a letter at the following address:

...For Dummies Book Registration
Dummies Press
7260 Shadeland Station, Suite 100
Indianapolis, IN 46256-3945
Fax 317-596-5498

...FOR DUMMIES™
BESTSELLING BOOK SERIES

Your Guide to Creating Software for the Mac — Covers Through OS 8.5

The Mac is back! With the success of the iMac and the power of high-end Macs, new Mac software programs are once again in demand. Whether you're a programming wannabe or a veteran developer, *Mac® Programming For Dummies®*, 3rd Edition gives you easy-to-understand, up-to-date guidance on Mac programming basics, compilers, programming languages, code writing, and more. So start creating new Mac OS 8.5 applications today — the fun and easy way!

Inside, find helpful advice on how to:

- Compile and run Mac source code easily with CodeWarrior Lite
- Understand why programming for the iMac is different
- Create menus that drop and windows that move — quickly and easily
- Avoid the most common Mac programming mistakes
- Become fluent in C, the most important Mac programming language
- Use ResEdit to edit 'MBAR' and 'WIND' resources
- Discover Dan Parks Sydow's debugging secrets — and get your programs up and running faster

Let These Icons Guide You!

Technical Stuff — **Highlights nerdy technical discussions you can skip if you want to**

Warning! — **Points out information that may help you avert disaster**

Mac Psychology — **Helps you understand unique Mac terminology and concepts**

About the Author

Dan Parks Sydow is the author of previous editions of *Mac® Programming For Dummies®* as well as *Mac® OS 8 For Dummies® Quick Reference* and *The Internet For Macs® For Dummies® Quick Reference*. Dan holds a degree in software engineering and has worked on a variety of software projects for Macintosh computers.

Technical Review
by Dennis Cohen

Valuable Bonus CD-ROM Includes:

- **CodeWarrior Lite** — Limited version of Metrowerks CodeWarri
- **ResEdit** — Resource editing software by Apple Computer, Inc.
- Source code and examples from the boo

Shareware programs are fully functional, free trial versions of copyrighted programs. If you like particular programs, register wit their authors for a nominal fee and receive licenses, enhanced versions, and technical support. Freeware programs are free, copyrighted games, applications, and utilities. Y can copy them to as many PCs as you like – free — but they have no technical support.

SYSTEM REQUIREMENTS: Macintosh with PowerPC processor with Mac OS 7.5 installe 24MB RAM; CD-ROM drive, double speed (2x) or faster

Printed in the U.S.A.

7 85555 00628 7

READER LEVEL
Beginning to Intermediate

COMPUTER BOOK SHELVING CATEGORY
Macs/Programming

$29.99 USA
$42.99 Canada/£28.99 Incl. VAT UK

FOR DUMMIES
BESTSELLING BOOK SERIES

Dummies Press™
a division of
IDG Books Worldwide, Inc.
An International
Data Group Company

see us at:
www.dummies.com
for info on other IDG Books titles:
www.idgbooks.com

ISBN 0-7645-0544-
5299
9 780764 505447