THINK C!

PROGRAMMING FUNDAMENTALS FOR THE MACINTOSH

Includes Macintosh Simulator C – an Innovative Tutorial Program on Two Disks!

- Unleash the Full Power of Your Macintosh – Even if You Are Not a Programmer
- Learn Macintosh Programming Fundamentals Quickly and Easily
- Master the Basics of the THINK C Programming Language from the Ground Up

Dan Parks Sydow
Think
THINK
Prima Computer Books

Available Now

*WINDOWS Magazine Presents*: Access from the Ground Up
Advanced PageMaker 4.0 for Windows
DESQview: Everything You Need to Know
DOS 6: Everything You Need to Know
*WINDOWS Magazine Presents*: Encyclopedia for Windows
Excel 4 for Windows: Everything You Need to Know
Excel 4 for Windows: The Visual Learning Guide
*WINDOWS Magazine Presents*: Freelance Graphics for Windows: The Art of Presentation
Harvard Graphics for Windows: The Art of Presentation
LotusWorks 3: Everything You Need to Know
Microsoft Works for Windows By Example
NetWare 3.x: A Do-It-Yourself Guide
Novell NetWare Lite: Simplified Network Solutions
PageMaker 4.0 for Windows: Everything You Need to Know
PageMaker 4.2 for the Mac: Everything You Need to Know
PageMaker 5 for Windows: Everything You Need to Know
Quattro Pro 4: Everything You Need to Know
QuickTime: Making Movies with Your Macintosh
Smalltalk Programming for Windows (with 3½" disk)
Superbase Revealed!
SuperPaint 3: Everything You Need to Know
Windows 3.1: The Visual Learning Guide
Word for Windows 2: The Visual Learning Guide
Word for Windows 2 Desktop Publishing By Example
WordPerfect 5.1 for Windows Desktop Publishing By Example
WordPerfect 6 for DOS: The Visual Learning Guide

Upcoming Books

CorelDRAW! 4 Revealed!
Improv 2.1 Revealed! (with 3½" disk)

How to Order:

Individual orders and quantity discounts are available from the publisher, Prima Publishing, P.O. Box 1260BK, Rocklin, CA 95677-1260; telephone (916) 786-0426; fax (916) 786-0488. If you are seeking a discount, include information on your letterhead concerning the intended use of the books and the number of books you wish to purchase.
Think

Dan Parks Sydow

Prima Publishing
P.O. Box 1260BK
Rocklin, CA 95667-1260
(916) 786-0426
To my wife, Nadine
Contents at a Glance

Acknowledgments xv
Introduction xvii

1 Introduction to C 1
2 Introduction to Macintosh Programming 21
3 C Data Types 31
4 Introduction to Functions 49
5 C Operators 73
6 Decision Making: Looping 87
7 Decision Making: Branching 107
8 All About Functions 125
9 Memory and Pointers 139
10 Arrays and Structures 161
11 Introduction to QuickDraw 187
12 Deeper into QuickDraw 205
13 Resources and ResEdit 229
14 Events 241
15 Menus 273
16 Windows 299
17 Dialog Boxes and Alerts 321
18 Complete Resources—ShapeMaker 365
19 Complete Source Code—ShapeMaker 387
20 One for the Road—MacCertificate 413
Appendix—ASCII Character Codes 459
# Table of Contents

Acknowledgments xv
Introduction xvii

**Chapter I** Introduction to C 1

- C: What and Why 2
  - What Is C? 2
  - Why Use C on the Macintosh? 2
- The Science of Programming 3
  - Programming C in Six Phases 3
  - The Cycle of Programming 7
  - But I'm Only a Beginner! 7
- A Comparison of ANSI C and Macintosh C 9
  - ANSI C 9
  - Macintosh C 9
- Ready for Some C? 10
  - Stepping through the Code 11
- Components of a C Program 17
- Program Debugging 18
  - Syntax Errors 18
  - Semantic Errors 19
- Chapter Summary 19
Chapter 2  Introduction to Macintosh Programming  21
  Why the Mac Is Unique  22
  The Macintosh Toolbox  22
  Resources  23
  The Structure of a Macintosh Program  24
  This Book’s Approach to Programming  25
     One Step at a Time  26
     The Role of the Simulator Software  27
     Using Pseudo Code  27
  Programming with THINK C  29
  Chapter Summary  29

Chapter 3  C Data Types  31
  ANSI C Data  32
  Data Types in Action  33
     What’s New  33
  Variables and Constants  35
  Why Different Data Types?  36
  C Data Types  37
     The int Type  37
     The long Type  39
     The char Type  39
     The float Type  41
  Type Conversion  42
  Symbolic Constants: #define  44
  Chapter Summary  46

Chapter 4  Introduction to Functions  49
  Flashback: What a Function Is  50
  Calling All Functions  50
  Toolbox Functions  54
     More Than Just DrawString()  55
     Function Naming  56
  Advantages of Functions  57
     Reusability  58
     Clarity  59
     An Aid in Planning  61
     Debugging  61
     Program Maintainability  61
  The Scope of a Variable  62
     Local Variables  62
     Global Variables  64
     Variable Naming  65
     Value Retention  65
     Local versus Global  66
  Cheating at Windows  68
  One More Function of Our Own  69
  Chapter Summary  71
Table of Contents

Drawing a Line and Entering the Toolbox 191
  Access to the Toolbox 192
  Using the Toolbox to Draw a Line 193
Where's the C? 194
The Graphics Pen 196
Rectangles 197
  The Rect Data Type 198
  Displaying a Rect 200
  Hiding a Rect 200
Animation, Already? 201
Chapter Summary 203

Chapter 12 Deeper into QuickDraw 205

Patterns 206
Shapes: Rectangles Revisited 207
  More Operations on Rectangles 207
  Ovals 210
  Round Rectangles 212
  Shape Summary 213
Drawing Text 215
  Drawing a String 216
  The Str255 Type 218
  Drawing a Character 219
  Text Characteristics 219
  Centering a String 222
Graphics Ports 224
Chapter Summary 226

Chapter 13 Resources and ResEdit 229

Code + Resources = Program 229
Introduction to ResEdit 231
Types of Resources 238
Chapter Summary 238

Chapter 14 Events 241

Events 241
The Simplest of Events 243
  The Resource File 243
  The THINK C Project 246
  The Source Code File 249
  But Where Are the Events and the Event Loop? 251
Event Records 254
  Retrieving an Event 256
Expanding the Main Event Loop 257
  Improved Event Loop Pass 1: Pseudo Code 257
  Improved Event Loop Pass 2: Adding GetNextEvent() 259
  Improved Event Loop Pass 3: Macintosh C Event Loop 260
  Improved Event Loop Pass 4: MultiFinder Aware 261
Event2: A Real Event Loop 265
Chapter Summary 271
## Chapter 15  Menus

- The Menu Manager 274
- Description of a Menu 274
- Menus and Resources 275
  - The MENU Resource 275
  - The MBAR Resource 283
- Creating a Menu in C 285

### Events and Menus 290

- Responding to a Mouse Click 290
- Responding to a Menu Click 293

### Chapter Summary 297

## Chapter 16  Windows

- The Window Manager 299
- Sorry to Disillusion You 300
- Description of a Window 301
  - Window Elements 301
  - Window Types 302
- The Window Structure 302
  - Windows and Resources 306
- Creating a Window in C 307
- Using a Window's Graphics Port 309
- Closing a Window 311
- Events and Windows 312
  - Mouse Events in a Window 313
  - Update Events in a Window 316

### Chapter Summary 319

## Chapter 17  Dialog Boxes and Alerts

- The Dialog Manager 322
- Description of a Dialog Box 322
  - Types of Dialog Boxes 323
- Dialog Items 324
  - Control Items 325
  - Edit Text Items 326
  - Static Text Items 326
  - Icons and Pictures 326
  - User Items 326
- Dialog Boxes and Resources 326
  - DLOG Resource 327
  - DITL Resource 330
- The Structure of a Dialog Box 336
  - Dialog Box Pointers and Records 336
- Creating a Dialog Box in C 338
- Handling a Modal Dialog Box 340
- Accessing Dialog Items 343
  - Getting Push Button Items 344
  - Getting and Setting Check Box Items 344
  - Getting and Setting Radio Button Items 348
# Table of Contents

- Getting and Setting Edit Text Items 352
- Getting Final Values of DITL Items 355
- Alerts: Stripped Down Dialog Boxes 359
- Alerts and Resources 359
  - ALRT Resource 360
  - DITL Resource 360
- Creating an Alert in C 361
- Chapter Summary 362

## Chapter 18  Complete Resources—ShapeMaker 365

- The ShapeMaker Program 366
- The ShapeMaker Resource File 369
  - Creating the Resource File 369
  - The DLOG Resource 370
  - The DITL Resource 372
  - The ALRT Resource 377
  - One More DITL Resource 379
  - The MENU Resource 379
  - The MBAR Resource 383
- Chapter Summary 385

## Chapter 19  Complete Source Code—ShapeMaker 387

- The ShapeMaker Program 387
- Function Prototypes 388
- Stepping through the Code 389
  - The Function Prototypes 389
  - The #define Statements 390
  - The Global Variables 392
  - The main() Routine 393
  - Program Initialization 394
  - Setting up the Menu 395
  - The Main Event Loop 395
  - Handling a Single Event 396
  - Handling a mouseDown Event 397
  - Handling a mouseDown in a Menu 398
  - Handling a mouseDown in the \text{\small{\sffamily{\selectfont{Mac}}}} Menu 399
  - Handling a mouseDown in the File Menu 401
  - Opening and Handling a Dialog Box 402
  - Setting a Check Box 405
  - Getting Information from a Dialog Box 406
  - Getting Edit Item Contents 408
  - Using Dialog Box Information to Draw in the Dialog Box 409
- Chapter Summary 411

## Chapter 20  One for the Road—MacCertificate 413

- The MacCertificate Program 413
- The MacCertificate Resource File 416
  - The DLOG and Its DITL 416
  - The ALRTs and Their DITLs 419
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The MENU Resource</td>
<td>423</td>
</tr>
<tr>
<td>The MBAR Resource</td>
<td>423</td>
</tr>
<tr>
<td>The WIND Resource</td>
<td>425</td>
</tr>
<tr>
<td>The PICT Resource</td>
<td>426</td>
</tr>
<tr>
<td><strong>The MacCertificate Code</strong></td>
<td>427</td>
</tr>
<tr>
<td>The Function Prototypes</td>
<td>427</td>
</tr>
<tr>
<td>The define Statements</td>
<td>428</td>
</tr>
<tr>
<td>The Global Variables</td>
<td>429</td>
</tr>
<tr>
<td>The main() Routine</td>
<td>430</td>
</tr>
<tr>
<td>Program Initialization</td>
<td>431</td>
</tr>
<tr>
<td>Setting up the Menu</td>
<td>431</td>
</tr>
<tr>
<td>Handling an Error Condition</td>
<td>432</td>
</tr>
<tr>
<td>Setting up the Window Drag Boundaries</td>
<td>433</td>
</tr>
<tr>
<td>The Main Event Loop</td>
<td>434</td>
</tr>
<tr>
<td>Handling a Single Event</td>
<td>434</td>
</tr>
<tr>
<td>Handling an Update Event</td>
<td>435</td>
</tr>
<tr>
<td>Updating a Window</td>
<td>437</td>
</tr>
<tr>
<td>Closing a Window</td>
<td>443</td>
</tr>
<tr>
<td>Handling a mouseDown Event</td>
<td>443</td>
</tr>
<tr>
<td>Handling a mouseDown in a Menu</td>
<td>445</td>
</tr>
<tr>
<td>Handling a mouseDown in the Apple Menu</td>
<td>445</td>
</tr>
<tr>
<td>Handling a mouseDown in the File Menu</td>
<td>446</td>
</tr>
<tr>
<td>Setting a Radio Button</td>
<td>450</td>
</tr>
<tr>
<td>Getting Information from a Dialog Box</td>
<td>451</td>
</tr>
<tr>
<td>Getting Edit Item Contents</td>
<td>452</td>
</tr>
<tr>
<td>Checking User Input</td>
<td>453</td>
</tr>
<tr>
<td>Opening a Window</td>
<td>457</td>
</tr>
<tr>
<td>Setting a Check Box</td>
<td>457</td>
</tr>
<tr>
<td><strong>In Conclusion</strong></td>
<td>458</td>
</tr>
<tr>
<td><strong>Chapter Summary</strong></td>
<td>458</td>
</tr>
</tbody>
</table>

**Appendix—ASCII Character Codes**                                       | 459  |

**Index**                                                               | 461  |
Acknowledgments

I would like to take this opportunity to thank the following for their roles in the Think THINK C project:

David Holzgang, for his technical review and re-reviews that resulted in numerous helpful suggestions.

Roger Stewart and Stefan Grünwedel, for keeping things rolling smoothly and quickly.

Betsy Ahl, whose editing tied together loose ends and added some polish.

Susan Glinert, for making the book look as good as it does.

Carole McClendon, for introducing me to Prima Publishing and for providing me with information at each step of the book’s development.

Dan Shafer, who, without knowing it, might just be responsible for this project becoming a reality.

Symantec Corporation, producers of the THINK C compiler!
Learning a programming language, any programming language, has always involved laboriously sifting through page after page of text and source code in an attempt to understand new terms, concepts, and syntax. Although the Macintosh computer is easy to use, many of the concepts involved in programming it are not. And even though the Macintosh is an exciting computer to use, the details of programming it are usually presented in a very unexciting way.

*Think THINK C* changes all of that. This book, written in a friendly style with plenty of illustrations, comes with an interactive software tutorial to provide you with an exciting, fun way of learning to program the Macintosh.

The tutorial, the Simulator software, uses the analogy of a book to teach you how to program your Macintosh using the C language. Screens are referred to as *pages* and can be flipped through like pages in a book. Pages are organized into chapters and topics that closely match those presented in the book. This makes it easy for you to go back and forth between the book and the software.
Whom This Book and Software are For

Think THINK C was created for the beginning to intermediate programmer who wants to learn to write Macintosh programs using the C programming language. The book is ideal for people who

- are new to computer programming.
- are experienced Macintosh users and want to know in depth how the Macintosh works.
- have programmed on other types of computers, such as PCs and mainframes, and now want to program on the Macintosh.
- have programmed the Macintosh using Pascal or Basic and now want to program using C.

What You Need

The Think THINK C package is a comprehensive Macintosh C programming guide, but there are a few things you will need to take full advantage of it.

Using the Book

As always, when learning a programming language, some programming background is helpful. But none is necessary. This book covers all of the basic concepts of the C language and provides all the information and examples you'll need to program your Macintosh in C.

Running the Software

To run the included Simulator tutorial software you'll need a Macintosh computer equipped with a hard drive. The Simulator software was designed to run on as many Macintosh configurations as possible. There are, however, certain minimum requirements your system must have. To use the Simulator software you must have

- a Macintosh Plus or higher, including any SE, II, LC, Performa, or Quadra model.
- System file 4.2 or higher.
- a hard disk drive with at least 4MB of free space.
- at least 512K bytes of free RAM memory.

The Simulator is System 7 compatible and can run on either a color or a black-and-white system. Any size monitor will suffice. The software is supplied on 800K disks so that owners of Macintosh computers with either 800K or 1.44MB floppy drives can load and use it.

Creating Complete, Stand-alone C Programs

When you are ready to create complete stand-alone Macintosh programs, you'll want to purchase a Macintosh C compiler. We recommend Symantec's THINK C compiler because of its tremendous popularity and ease-of-use. The source code examples included here will compile using the latest version of THINK C, version 6.0. Symantec also makes a C++ compiler, Symantec C++, that includes THINK C 6.0. Should you decide eventually to move on to object-oriented C++ programming, you'll be able to use the same compiler.

How to Use This Book and Software Package

The Simulator program is the software tutorial that accompanies the Think THINK C book. The software was included with the book because Prima Publishing realizes that every person has a different style and pace of learning. Having both a book and a software tutorial provides something for everyone.

The software uses the analogy of a book to teach you how to program your Macintosh using the C language. Screens are referred to as pages and can be flipped through like pages in a book. Pages are organized into chapters and topics that closely match those of the book. This makes it easy for you to move back and forth between the book and the software.

The topics in the Simulator program match chapters and sub-chapters in the book. If you want to turn to the book from any place in your Simulator program, just note the chapter and the name of the topic that you are working on; you will find a corresponding chapter and topic in the book. To reverse the process and go from the book to the Simulator program, look
in the Simulator's pop-up Table of Contents menu to find the corresponding topic.

Although the Simulator software mirrors the contents of the text, in some cases, it goes beyond the text. This happens particularly in later chapters where you are beginning to work directly with programming concepts and techniques. The text tells you how these work and gives directions for using them in THINK C projects. The Simulator software, in many cases, allows you to experiment directly with code, testing your ideas and understanding without using any additional software. And, in Chapters 18 and 19, where you build a complete Macintosh application, the Simulator software takes you beyond the text and shows you how to modify your software projects once they are created. All in all, the combination of the Simulator software and the text provides a significantly stronger learning environment than either one would on its own.

The following are a few tips on using this book/software package. Start by running the Simulator software. Then,

- if you find a concept extremely tough, turn to the same topic in the book. It may offer different wording or a different piece of source code that will help answer your questions.
- if you want to keep more information in front of you, leave the book open to one page while you study other screens in the Simulator.
- if you want to take notes, mark up the book.
- if you're away from your Macintosh, take the book with you!

**Features of the Simulator Software**

The Simulator software uses a friendly, interactive approach to teaching you the many concepts you need to master in order to write your own Macintosh programs. Among the many features of the software are

- A pop-up Table of Contents, which allows you to move to any chapter topic at any time.
- Highlight Words, which can optionally be clicked on to get more information. They provide another layer of learning.
- Simulation Pages, which provide on-screen animation to bring to life difficult concepts that just can't be explained clearly on the static pages of a book.
- **Code Pages**, which allow you to piece together short blocks of code by simply clicking on buttons. You can then watch your code execute right on your screen!

- **Question Pages**, which test your knowledge and provide helpful feedback.

- A **Status Page** at the end of each chapter topic, which lets you know how well you've answered the questions posed about the topic. The Status Page allows you to return to questions you've missed.

- An **Achievement screen**, which keeps track of your progress for each topic in the tutorial.

- A combination **Index and Glossary**, which allows you to look up a definition and then go directly to the Simulator page that contains information about the word you looked up.

- A **Related Topic Page** feature, which lets you jump to another screen to get more information about a topic. Click on the Related Topic Page icon, and you'll be whisked away to another topic in the Simulator. Click on the icon again, and you'll be returned to the page on which you started.

---

**Installing the Simulator Software**

The Simulator software is easy to install. Follow the steps provided here to get your Simulator up and running.

The Simulator software comes on two 800K disks, titled Disk 1 and Disk 2. Each disk contains one file, which has been compressed to save disk space. These files are self-extracting. This means that you do not have to own any special program to decompress the files back to their original sizes.

You'll want to copy the two compressed files to your hard drive before decompressing them. This will serve two purposes: it will make the decompression run smoothly, and it will allow you to work with copies of the files, thus preserving the original files and disks as backups.

You can copy them directly to your hard drive—you don't have to create any new folders. Copy the two files onto your hard drive at this time:

1. Insert Disk 1 into your floppy drive and copy the one file on the disk, Simulator.seal, to your hard drive.
2. Remove Disk 1 from your floppy drive and insert Disk 2. Copy its one file, Simulator.sea2, to your hard drive.
Both Simulator files should now be on your hard drive, as shown in Figure I–1. Of course, because the other programs and folders on your hard disk are different from ours, your hard disk folder won't look exactly like ours.

Now, it's time to decompress, or extract, the files. Double-click on the Simulator.sea1 file. You'll be presented with a dialog box like the one pictured in Figure I–2. Extraction starts with the latter of the two files, Simulator.sea2. Find that file in the list of files and folders and click on it to highlight it. Then click on the Load button, as shown in Figure I–2. Again,
A dialog box prompts you for the destination folder.

Your list of files will not be the same as ours, but it will contain the Simulator.sea2 file.

After clicking on the Load button, you will be presented with another dialog box—the one shown in Figure I–3. This dialog box asks you to specify where you want the extracted files to be placed. The extraction process will automatically create folders for you—you need specify nothing here. It doesn’t matter what folder or file name is highlighted. Simply click on the Extract button, as shown in Figure I–3. Again, your list of folders and files will not match ours.

After clicking on the Extract button, you have nothing to do but sit back and watch. The extraction runs on its own. The dialog box that marks the progress of this process is shown in Figure I–4.
When extraction is complete, you will be returned to the Finder. The files that make up the Simulator will be extracted. You will see a new folder, titled Simulator Folder, in your hard drive folder. Within this folder are two more folders and the Simulator program, as shown in Figure 1-5.

The last step is to move the Simulator.sea1 and Simulator.sea2 files into your trash can. You've extracted the files you need from these files; get rid of them and free up the disk space they occupy.

If something went wrong during the extraction process, you still have the original self-extracting files on your original two 800K disks, so you'll be able to repeat the process.

**IMPORTANT**

Keep both the Simulator_Files folder and the Macintosh Simulator C program in the same folder, as shown in Figure 1-5. As the Simulator program runs, it will look for files contained in the Simulator_Files folder. And it assumes that the Simulator_Files folder is right near by.

**Using the Simulator Software**

The Simulator software has many features that simplify the learning of Macintosh C programming. This section will give you a brief description of these features.
Running the Simulator C Software

To run the Simulator, simply double-click on the program Simulator C.

Pages and the Control Panel

The Simulator is analogous to a book in that screens are thought of as pages. Going from one screen of information to another is likened to turning a book page. The software has two windows—the page window and the control panel window. Figure I–6 shows a typical page. Figure I–7 shows the control panel.

Pages are turned by clicking on the Next or Previous Page icon on the control panel, as shown in Figure I–8.

If you want to go to a particular page, you can move more quickly through a topic by jumping five pages back or ahead. Pages are turned five at a time by clicking on the Back 5 or Forward 5 Page icon on the control panel, as shown in Figure I–9.

---

Chapter 9: Memory and Pointers

Topic: Memory

Computer memory is not one long ribbon of uninterrupted space. It is small cells grouped together.

We call each cell a bit.

At any given time, each bit is capable of taking on one of just two values: a one or a zero. The next Page illustrates this.

---

FIGURE I–6 A typical Simulator Page

FIGURE I–7 The control panel
The Simulator consists of 20 complete chapters. You click on the control panel’s Chapter icon to go to a different chapter. This icon is shown in Figure I–10.
Clicking on the Chapter icon calls up a hierarchical menu. Move the mouse to highlight any chapter title; then move the mouse to the right to bring up a submenu of chapter topics. Release the mouse button on a topic to start that topic. Figure I-11 shows an example.

When you complete a chapter topic, it will appear with a check mark beside it when you return to the Chapter/Topic menu. In Figure I-12, the Introduction and Memory topics of Chapter 9 have been completed.

**Simulator Pages**

Simulator Pages may contain text, graphics, or both. There are several special types of pages.
Simulation Pages

Some pages have Simulations on them. A Simulation can take numerous forms, such as a demonstration of code being run or a short sequence of animation to clarify a concept. If the control panel’s GO icon is enabled, click on it to view the Simulation. If the icon is dim, there is no Simulation on that page. Figure I–13 shows the GO icon in both an enabled and a disabled state.

Some Simulations allow you to slowly step through them. If that is the case, the control panel’s STEP icon will be enabled. Figure I–14 shows a typical Simulation Page. It allows the user to click on a bit in memory to view its contents.
Chapter 12: Deeper Into QuickDraw

Topic: Shapes Revisited

```
Rect the_rect;
Pattern the_pat;
SetRect(&the_rect, 20, 50, 350, 200);
PenPat(dkGray);
PaintRect(&the_rect);
FrameRect(&the_rect);
SetRect(&the_rect, 175, 170, 270, 290);
PaintOval(&the_rect);
InvertOval(&the_rect);
```

FIGURE 1-15 A typical Code Page

**Code Pages**

Code Pages are a form of Simulation. A row of 12 buttons appears across the top of the page. Some or all of the buttons will be named. Clicking on a button adds a line of Macintosh C source code to the page.

You use the buttons to add and take away lines of code to customize a mini-program. Then, when you click on the GO button, a window that displays the results of your code will open. The process takes place just as if the code were contained in a full Macintosh program. Figure I–15 shows a typical Code Page.

**Highlight Word Pages**

Some pages contain Highlight Words. If a word appears in bold on a page, click the mouse on it. The word will change colors if you have a color system or turn to an italic style if you have a black-and-white system. A window that contains more information about the word will open. Figure 1–16 shows `operators` as a Highlight Word.

Clicking on Highlight Words is optional. They exist to provide another layer of information. If you already know the meaning of a Highlight Word, you can feel free to move on without clicking on it.
operators Note:

An operator is a symbol that operates on the objects that surround it. In the simple math equation below, the "+" sign is the addition operator:

\[ 6 + 4 = 10 \]

As you progress through this software you'll encounter numerous operators—and not just mathematical ones.

The C language was created with the needs of the programmer in mind. It has a full set of operators, the symbols that allow you to process data in many ways.

Question Pages

Many topics include one or more Question Pages. The questions may be either multiple choice or true/false. Just click anywhere on the choice you think best answers the question. A check mark will be placed beside your choice. If you're right, you'll be congratulated. If you're wrong, you will receive helpful feedback. Figure 1-17 shows a typical Question Page.

QUESTION:
The purpose of the QuickDraw routine FrameRect() is to:

- Set and display one rectangle
- Frame a rectangle
- \( \checkmark \) Display a rectangle in a window
- Set the pixel coordinates of one rectangle

Click on one of the answers.

FEEDBACK: No. It frames a rectangle that has already been set up with SetRect().
You're allowed two chances to answer any one question. Subsequent choices will be ignored. Once you get to the last page of the topic, the Status Page, you'll be given the opportunity to go back to all missed and skipped questions to try again.

**Status Pages**
At the end of every topic, you'll find a Status Page. Here you can see how well you have mastered the topic and decide whether to go back and brush up or carry on with the next lesson. If you've skipped and missed some questions and you decide to go to the next topic without finishing them, your score for this topic is deleted when you leave the topic. Figure 1–18 shows a Status Page.

**Achievement**
The Simulator keeps a permanent record of all the questions you attempt. It uses this information to determine your achievement. Click the control panel's Achievement icon, shown in Figure 1–19, to see a dialog box that shows your current status for each chapter. Click on any chapter to see how you've done on each topic in that chapter.
Index/Glossary

You can look up a topic in the Simulator's built in index. To get to the Index/Glossary, just click on the control panel's Index icon, shown in Figure I-20, at any time.

At the Index/Glossary screen, you can scroll through the list of topics. Click on a topic of interest. Click on the Go To Topic button to go directly to the topic, or just use the index as a glossary by reading the definition of the selected topic. Figure I-21 illustrates this.

Related Topic Pages

Every page in the Simulator, with the exception of Simulation Pages and Question Pages, has a different Simulator page that is considered its Related Topic Page. If a topic or concept has not been explained to your satisfaction, you might want to try clicking on the control panel's Related Topic Page icon, shown in Figure I-22. When you click on this icon, the page you are on will close and a different page will open. This page could be from anywhere in the Simulator—the current topic, a different topic, even a different chapter.
GrafPort - the Macintosh C structure type that holds the attributes of a graphics port.

Once you've looked over the Related Topic Page, you can return to the original page, the page you were previously on, by simply clicking on the Related Topic Page icon again.
By choosing the exciting Macintosh computer to program, and the powerful C language with which to program it, you’ve made two excellent choices. And by reading this book and following along with the included Simulator software you’ll be able to understand the C language and how to unleash its power onto your Macintosh.

In this chapter we introduce you to the C language. You’ll learn about its origin, its power, and the reasons for its being the language of choice for programming on the Macintosh. You’ll also read about some of the basics of programming—techniques of programming that apply to any language and any computer.

You’ll learn the differences between the type of C that you, a Macintosh programmer, will use and the type of C that programmers of other types of computers use.

Here you’ll be introduced to what a C program looks like and what the key parts of it do.

This chapter introduces you to programming in C. Chapter 2 introduces you to programming for the Macintosh computer.
C: What and Why

There are many programming languages to choose from. We recommend the C language for the Macintosh. Here we provide a little background on what the C language is and why we recommend it so highly.

What Is C?

The C programming language was originally designed by Dennis Ritchie of Bell Labs in 1972. At that time he and Ken Thompson were trying to produce a version of the FORTRAN language for the UNIX operating system. Like many good creative efforts, there were several twists and turns along the way. What is important to you is the result—the C language, much as it is known today.

Ritchie was designing his language for a computer that had a very limited amount of memory. He therefore had to trim and simplify in order to be able to include all the features he wanted in his new language. The result, thankfully, was a tight, compact, straightforward language that retains the powerful features of other languages.

Why Use C on the Macintosh?

As we stated above, the C language is a compact programming language. This makes it an efficient language. The programs you create using C will be, like C itself, more compact than those written in many other languages, such as Pascal. And this makes them run more quickly than programs written in other languages.

The C language was created with the needs of the programmer in mind. It has a full set of operators, the symbols that allow you to process data in many ways. And C is less strict than other programming languages, giving you the freedom to delve into areas that other languages might restrict you from.

These reasons are good ones for choosing C as a programming language. But specifically, as a programmer about to embark on a Macintosh programming path, why should C appeal to you? Here are a few reasons.

When the Macintosh computer was first introduced its central processing unit, the CPU, was the 68000 microchip. This chip was already in use in many UNIX machines. Since C was written for UNIX, many time-tested C compilers that could be used with the Macintosh CPU were already in existence. Over time these compilers have been fine-tuned to run on the Macintosh and to take advantage of the features of the Macintosh.
The C language is popular at universities, and so is the Macintosh. C is also very popular with professional programmers—for more than half of them, C is the development tool of choice. Sometimes it’s good to be different; sometimes it’s not. In choosing a computer language, it’s not! Because C is so popular on the Macintosh, you’ll be able to find the programming information you need. You’ll also find greater acceptance and credibility when others hear that you are programming in a popular, proven language.

The Science of Programming

Computer programming is both an art and a science. The art is the creativity that goes into making exciting things happen on your Macintosh screen. To get to the creative stage, you must first master the science—the basics of a programming language and the basics of writing a program. This book devotes itself to both the science and the art of programming in the Macintosh C language.

Before we discuss the science of C, we must spend at least a moment on the science of not just the programming language, but the process of programming itself.

Programming C in Six Phases

The most successful programmers are those who plan out their work in advance and follow a sequence of phases that take them from the start of a program to its completion. The sequence of six phases that we now present is not of our own creation. It is a format tested and proven by many programmers over many years.

Phase I: Determine What the Program Is to Do

If you think this phase sounds too obvious for inclusion here, you’re making a very common and potentially time-consuming mistake. Although it is possible to make changes and additions to your program at any point in its development, it is infinitely easier to plan for all of the necessary features before you write a single line of C language code. Spend some time listing all the things you want your program to do. Also list the information your program will need, what the program will be required to do with the information, and what information the program is to display. In Phase 1, you should concentrate on the tasks you want to accomplish rather than the
Think THINK C

My CD Listing Program

What It Should Do
1. List user's CDs in alphabetical order
2. List user's CDs by musical group

What It Needs
1. The titles of all of the user's CDs
2. The name of the group for each CD

What It Needs To Do
1. Ask user how he wants to see the list
2. Sort by CD title
3. Sort by group name

What It Will Display
1. CDs, either by title or group

FIGURE 1–1 Phase 1 for a CD listing program

details of how you will accomplish them. Figure 1–1 illustrates Phase 1 for a very simple program that lists the program user's CD collection.

Phase 2: Design the Structure of the Program

In Phase 2 you will get into a little more detail. What will users see when they run the program? You're writing a Macintosh program, so of course you'll want menus. What options should appear in those menus? You won't write any code in Phase 2, but you might want to think about how you will hold your information—what C language data structures you will use. Figure 1–2 shows some of the notes that you might create for Phase 2 of your CD listing program.

Phase 3: Write the C Language Code

After completing Phases 1 and 2, you will have a sound idea of where you're going and how you're getting there. You are now ready to write your C lan-
guage code. The act of writing code will become quicker and easier as you progress through this book and the Simulator software.

To write the code, you use the text editor that is included with your C compiler. You then save it as a source code file. If you don't own a C compiler yet, don't worry. That is the purpose of the Simulator software—to let you learn about code, enter code, and see it carried out on your screen without the need for a compiler. Here is an example of what a small part of an actual Macintosh C source code file looks like:

```c
Number_One()
{
    int num;

    num = 1;
    MoveTo(10,40);
    if (num == 1)
```

**FIGURE 1–2** Phase 2 for a CD listing program
Phase 4: Compile the Code into a Program

By Phase 4 much of the difficult work is behind you. You must now compile, or transform, your source code into a form that the Macintosh can understand—executable code. This is the job of the C compiler. Most C compilers now actually perform two tasks. The compiler first compiles one or more source code files, creating an object code file from each source code file. The compiler then links those object code files together into one executable file. An executable file is another name for a ready-to-run program, or application.

As the compiler performs its task of compiling, it may come across a bug, or error, in your source code. If this happens the compiler will not continue on to create an executable file. Instead, the compilation process will stop, and the compiler will report the error to you. This feedback from the compiler allows you to correct your code and try again. Figure 1-3 shows a typical error message that the THINK C compiler returns to you.

Phase 5: Test and Debug

After completing Phase 4, you have an executable file—a program that is ready to run. Shouldn’t you be done at this point? It is very unlikely that you will ever be this lucky! Programming can be a tricky business, and even if your source code makes it successfully through Phase 4, there are no guarantees that it will perform in the manner you had hoped. Testing your program is the process of running your program and keeping a careful eye on what happens. This may result in the identification of a part of your program that does something unexpected. If this is the case, it becomes your
job to *debug* the program—to uncover the cause of the error and make the necessary correction in your source code.

Many beginning programmers become disheartened by bugs. We would like to console you with the idea that as you become a skilled programmer you will eventually cease to make programming errors. But we can’t. Errors are an inherent part of computer programming. As your skills increase, you will make fewer programming errors, but you will never reach the point where you forget what a bug is. If you can learn to think of the detection and correction of bugs as a challenging way to increase your programming knowledge, you’ll be much better off!

**Phase 6: Enhance the Program**

Phase 6, the final phase, is optional. But all good programmers perform it. When your program is complete and ready to use, you could consider your job done. But as people use your program, they may suggest new features that would appeal to them. And some may even discover hidden bugs lurking in untested areas of your program. As a responsible program developer, you’ll want to follow up on the suggestions of users so you can enhance your program.

**The Cycle of Programming**

The six phases of programming do not fall into a straight line that you step through from beginning to end. You repeat some phases. When you compile source code in Phase 4, you may encounter a bug. If so, you cannot move on to Phase 5 for testing and debugging. Instead, you must return to Phase 3 to make the necessary source code corrections. You then return to Phase 4 and try to compile again. You may find yourself repeating the process of going from Phase 3 to Phase 4 and then back to Phase 3 over and over.

When you get to Phase 5, you will no doubt meet with at least one bug. When that happens, you will have to return to Phase 3 to make the needed corrections. You might even have a flaw in your initial ideas and have to return to Phase 1 to determine what the program is to do or Phase 2 to rethink the ideas behind your program.

This repetitious sequence makes programming an iterative, or repeating, process. Figure 1–4 shows how the six phases can follow different paths.

**But I’m Only a Beginner!**

As someone new to Macintosh C and perhaps to programming itself, the six-phase process may be hard to follow. You might wonder how you can choose data structures in Phase 2 when you don’t know what data structures
are or what structures are available in C. That's a very good point. You'll perfect the process of going through these six phases as you program. For now, you can still follow them to some degree. In short, Phase 1 and Phase 2 merely stress that you should plan ahead and think before you act.
As you read this book and run the Simulator software, you will be learning the C language in bits and pieces. You will not start by writing a Macintosh program. So don't worry if you don't think you can work your way through six phases just yet. By the time you approach the end of this book, you will have Macintosh C mastered to the point that you will be able to write your own programs—and then you can make full use of the six-phase plan.

A Comparison of ANSI C and Macintosh C

Throughout this book we will talk about both ANSI C and Macintosh C. Are these two separate C languages? To program on the Macintosh do you have to learn two versions of the same language? This section will put your fears to rest on these issues.

ANSI C

After Dennis Ritchie created the C language, he and Brian Kernighan wrote the book that was to become the C reference that most others would follow: *The C Programming Language*. As time went on, C compilers were written for many computer systems. Most used the Kernighan and Ritchie text as their basis. But the Kernighan and Ritchie text left some issues unexplained. Thus developers of C compilers incorporated features into their compilers as they saw fit. Without an officially established standard, versions of C differed from one another in some aspects. With this in mind, the American National Standards Institute (ANSI) developed a set of C language standards. The result was ANSI C. When you buy a C compiler that claims to be an ANSI C compiler, you are assured that the compiler recognizes the C data types and structures, which are the basic components of the C language. You know that the compiler supports C code that complies with the rules of ANSI C.

Macintosh C

Much of the C language included in ANSI C is applicable to programming on the Macintosh. But it is not enough. ANSI C has no provisions for creating the graphical elements that separate Macintosh programs from those that run on text-based computer systems. Fortunately, Apple has provided
everyone who has a Macintosh computer with the means to program such exciting Macintosh features as menus and windows.

Every Macintosh has chips that contain a host of miniprograms that C programmers use to create and display menus, dialog boxes, windows, and more. Apple calls these miniprograms, or routines, the Toolbox. They exist in the ROM chips of every Macintosh computer. The Toolbox routines are accessed using the C language. Because the Macintosh did not exist when C was created, the C language made no provision for accessing Toolbox routines. So Macintosh C has taken the basic features of the C language and expanded upon them.

What is the result of all of this? There's good news and bad news. The bad news is that knowledge of ANSI C is not enough to enable you to write programs that will run on the Macintosh. You need to know the C that gives you access to the Toolbox routines. Fortunately, the good news outweighs the bad news—you do not have to learn all of ANSI C and the extended Macintosh C features. Just the basics of ANSI C will be enough. You can then supplement what you know of ANSI C with C that is particular to the Macintosh.

Like the positive-minded people we are, we'll end the chapter with more good news—this book, and the Simulator software, will provide just the right amount of ANSI C and Macintosh C to start you on your way to writing complete, functional Macintosh programs.

**Ready for Some C?**

In the following chapters we will be covering the details of the C language. Before we do, a little exposure to some of the basics of C will be helpful. This section gives you an overview of C by presenting a very small Macintosh C program. The intent here is not to teach you the details of C but, rather, to give you a brief look at some of basic concepts underlying the language. Before reading on, look at Listing 1-1.

**LISTING 1-1  A very simple Macintosh C program**

```c
main()
{
    int num;

    num = 1; /* assign num a value */
```
The number is one.

FIGURE 1-5 Result of running Listing 1-1

```
Make_A_Window();
if (num == 1)
    DrawString("\nThe number was one.");
```

If you run the program, you will see a window appear on the Mac’s screen, along with one line of text, as shown in Figure 1-5.

From looking at Listing 1-1, you might have determined what was going to happen. You may still have some questions, though. Is it really that easy to create a window? Why does one line have a single equal sign while another line has two equal signs? Breaking down Listing 1-1 and going over it line by line will answer many of your questions and give you a cursory understanding of the C language.

**Stepping through the Code**

Here we’ll break down Listing 1-1 to see what is really going on. If you don’t fully understand some of our explanations, don’t become alarmed. We mean for this to be only the barest of introductions to the C language. The rest of this book will give you all the detail you need.

**The `main()` function**

`main()`

Functions are the building blocks of a C program. A function is one or more lines of C code grouped together and given a name. Functions are also
called routines—the names are interchangeable. The function name is followed by parentheses that identify it as a function. Listing 1–1 groups several lines of code into a function called main(). When you write functions of your own, you'll be able to give them just about any name you want. But every C language program must have one function called main(). The function called main() marks the starting point of execution of your program.

Functions are very important elements of the C language. So important, in fact, that we've dedicated two chapters to them—Chapter 4 and Chapter 8.

Braces

{}  

Braces mark the beginning and end of a function. The lines of code within a pair of braces are referred to as a block of code. In some other computer languages, such as Pascal, the words begin and end serve the same purpose as C braces.

Declaration Statements

int num;

This line of code is a declaration. A line like this declares, or makes known to the compiler, that a variable is being created. A variable holds a value of a certain type. In this example, the variable has the name num, and it can hold a value of type int. The name num was of our choosing. The C type int is one of several standard C types available. It refers to an integer—a whole number that contains no decimal point. Your C compiler needs this information so it can set aside the proper amount of computer memory to hold the value that num will eventually have.

Words like int that are a defined part of the C language are special. They are keywords, and you may only use them as intended by the C language. You cannot override their built-in meaning. We'll cover the C keywords throughout this book.

Note the semicolon following num. The semicolon lets your C compiler know that the line is a C statement. In C, a statement is a complete instruction to the computer. The semicolon tells the compiler where a statement ends.
We mentioned that the variable name, num, was of our own choosing. You have a lot of freedom in choosing the names you give to variables, but there are a few rules:

- You must use a combination of lowercase letters, uppercase letters, digits, and the underscore.
- The first character of your variable name must be a letter or an underscore (not a digit).
- The C language is case-sensitive. That means that the compiler distinguishes between lowercase and uppercase letters: C views boy and Boy as two different names.

Some sample variable names, legal and illegal, appear in Table 1–1

<table>
<thead>
<tr>
<th>Legal Names</th>
<th>Illegal Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>$cost</td>
</tr>
<tr>
<td>animals</td>
<td>8years</td>
</tr>
<tr>
<td>total3</td>
<td>ready?</td>
</tr>
<tr>
<td>Grand_Total</td>
<td>boy-girl</td>
</tr>
<tr>
<td>_done</td>
<td>st*r</td>
</tr>
<tr>
<td>num_pets</td>
<td>two words</td>
</tr>
</tbody>
</table>

**NOTE**

Variable names should be descriptive; they should give the reader some idea what the variable is all about. If you want a variable to hold the final score of a game, by all means call it final_score.

The variable final_score brings up a point. If you think that two words best describe the intentions of your variable, you can use the underscore character within a variable name. A variable name must be one word without any spaces in it. Thus, final score is not a valid variable name, but final_score is.

One last point on variable naming. Because C is case-sensitive, it's wise to stick with one naming style. In the next several chapters, you'll notice that we've chosen to use only lowercase letters in our variable names.
Assignment Statements

num = 1;

This line of code is an *assignment*. An assignment gives a value to a previously declared variable. The previous line declared an integer variable named num. This line assigns a value of 1 to the variable num. The declaration told the compiler to set aside some memory for a variable named num. The assignment tells the compiler to put the value 1 in that memory. An assignment statement, like a declaration statement, ends with a semicolon. When the computer performs a statement, we say that the code is being carried out, or *executed*.

This line might help you understand why a variable is called a variable. You can change, or vary, the value of a variable. Even though you assign a value to a variable in your program, you are free to give that very same variable a different value later in your program.

Comments

/* assign num a value */

A comment is text that is readable by you, a human being, but not by the computer. You add comments to your programs so your work is more understandable to people (including yourself) who look at your code. You can put a comment on the same line as code or on a line of its own or even extend it to more than one line.

When it's time to compile the source code, the C compiler will ignore everything that appears between the /* and the */.

Function Calls

Make_A_Window();

Recall the description of the main() function. We said that function names are identified by the parentheses that follow them. So you may have guessed that this line of code represents a function name. And you're right. But in this case a function is not being defined; it is being used. A function is being *called*, or *invoked*.

Notice that braces do not appear below the line containing the function call as they would if you were writing a function. What this line does is tell the program to go off and examine the rest of the source code to find a
function called Make_A_Window(). Once found, the program executes the lines of code that make up that function.

When the code in the Make_A_Window() function has been executed, control returns to the main() function.

We'll explain the differences between functions and function calls in detail in Chapter 4. Until then, see Figure 1–6 for an illustration of what you've learned to this point about functions and function calls.

At this point we must come clean—Listing 1–1 is not a complete Macintosh program. A Macintosh program is minimally a couple of pages long. We're giving you only part of a Macintosh program—a code fragment. We did not include the code that would make up the function Make_A_Window(); we just carried on as if the program were complete. Underhanded? Sneaky? Perhaps. But we wanted to be able to demonstrate C concepts and code without having to force you to digest pages of unfamiliar code.
As we introduce topics, we'll supply just a few lines of code to let you see what's going on. The Simulator software will let you execute those lines as if they were part of a complete program. We think you'll come to appreciate not having to wade through page after page of code built on concepts not yet covered. We use this approach throughout the remainder of the book. Don't worry though; when you're ready to write a complete program of your own, we won't leave you stranded. The last three chapters of this book include listings for complete Macintosh programs.

**Conditional Statements**

```c
if (num == 1)
    // code block
```

A computer language is not of much use to you if it does not give you the power to include decision-making in your code. The C language has several types of control statements. The `if` statement is a branching, or conditional, statement. What this line says is that if the condition is met—that is, if variable `num` has a value of 1—execute the line of code that follows. If not, skip the line that follows. Because an assignment statement was used earlier to give `num` a value of 1, the `if` test will pass, and the program will execute the next line of code.

Why two equal signs? You'll have to trust us for now. We'll go into detail about this in future chapters.

**The DrawString() function**

```c
DrawString("The number was one.");
```

Earlier we described a function call—the invoking of the function `Make_A_Window()`. The line of code we're now examining is also a function call—with one key difference. You do not have to write the code that makes up the function `DrawString();` it has already been created for you.

Using Macintosh C, you can access a host of functions that were written by Apple programmers long before you got your Macintosh. `DrawString()` is just one such function. There are more than a thousand others, grouped together in the Toolbox. These Toolbox functions will be your ticket to writing real Macintosh programs that make use of windows, menus, and dialog boxes. We'll describe the Toolbox in greater detail in the next chapter.

The `DrawString()` function differs from the other two functions we've examined—`main()` and `Make_A_Window()`—in one other way. It has
something enclosed between its parentheses. The material that appears between the parentheses represents information that will be passed to the DrawString() function. In this example, the phrase “\pThe number was one” is passed. The DrawString() function will print to a window whatever appears between the quotes, minus the strange-looking \p characters. We'll discuss DrawString() in more detail in Chapter 12.

That completes our breakdown of Listing 1–1. If you've followed much of what we've discussed in this section, take a moment to congratulate yourself now. You're still not out of the first chapter, and already you are well on your way to mastering Macintosh C.

We have so much faith in what you've learned we think you're ready to take a look at another short program. Try to predict what the outcome will be for Listing 1.2.

LISTING 1–2  Watch where you swim.

main()
{
    int sharks = 1;

    Make_A_Window();

    if (sharks > 0)
        DrawString("\pSwim fast!");
}

If you guessed that the above program will produce a window with the line “Swim fast!” written to it, you were right!

Components of a C Program

From the breakdown of Listing 1–1 some generalizations can be made that apply to all Macintosh C programs:

- A function is a collection of C statements grouped between a pair of braces. A function has a name followed by parentheses.
- A function may contain different kinds of statements. The three kinds of statements seen so far are:

  declaration statements, which introduce a variable by giving its type and name.
assignment statements, which give values to declared variables.
control statements, like the branching if statement, which provide different paths your program can take.

- A program consists of one or more functions. One, and only one, of the functions must be called main().

Program Debugging

In the section “Programming C in Six Phases” we called Phase 5 the Test and Debug phase. When you debug a program, you are searching for the cause of errors and then making the necessary corrections in your source code. While your very brief introduction to C has not yet given you enough information to debug a C program, it has given you enough insight into programming that we can at least speak about these errors in general terms.

Programming errors are divided into two broad categories—syntax errors and semantic errors.

Syntax Errors

Syntax has to do with correct usage. If you use proper C language words and symbols but arrange them incorrectly, you are guilty of a syntax error. In our breakdown of Listing 1–1 we described a declaration statement:

```
int num; 8 /* a valid declaration */
```

First comes the variable type, then the variable name, and finally, a semicolon. If we had reversed the order:

```
um int; /* an invalid declaration */
```

we would have committed a syntax error. We used a valid variable name and variable type, but we presented them in an invalid order. Here are three other examples of incorrect usage, or syntactical, errors:

```
int num /* no semicolon */
int 2nums; /* illegal variable name */
integer num; /* "integer" is not a valid type */
```
Your C compiler will discover syntax errors. A C compiler knows the rules of C and knows when they are being broken. When you compile a program that contains one or more syntax errors, the compiler will present you with an error message for each.

**Semantic Errors**

The second category is *semantic* errors. These are errors in meaning or logic. When you are thinking about buying an orange but tell someone to buy an apple, you are committing a semantic error. A semantic error occurs when you follow the rules of C, but the code you write does not reflect what you meant. You may have heard the old saying: "Listen to what I mean, not what I say!" You can hope that a person will understand your true intent, but you certainly can't expect a C compiler to. An example of a semantic error would be writing code to divide one number by another, when in fact you meant to multiply the two numbers.

Semantic errors can be more troublesome than syntax errors because the compiler does not detect them. A compiler cannot know what you intend your code to do, only whether or not your code follows C rules. To find semantic errors, you must run your program and verify that it does what you intended it to do.

**Chapter Summary**

The C language is a compact, efficient programming language created with the needs of the computer programmer in mind. Learning to program in the C language will give you an understanding of concepts useful for all programming languages. Knowing how to program in C will give you credibility among other computer programmers.

Programming is fun and creative, but it is also structured. While there are several schools of thought on the methodology of developing a computer program, all agree that you should follow certain time-tested steps. One very popular method divides the creation of a program into six phases:

Phase 1: Determine what the program is to do  
Phase 2: Design the structure of the program  
Phase 3: Write the C language code  
Phase 4: Compile the code into a program  
Phase 5: Test and debug  
Phase 6: Enhance the program
A C compiler turns your C code into instructions that are understandable by the computer. Companies that sell C compilers usually refer to them as ANSI C compilers. This means that their C follows a set of C language standards developed by the American National Standards Institute. When we speak of Macintosh C, we are referring to the special features added to the C language that help programmers include all of the graphical features expected of a Macintosh program. As a Macintosh programmer, you will be using the ANSI C that all C programmers use as well as additional Macintosh C code that will add menus and windows to your programs.

A C program consists of functions. The one function required of all C programs is the main() function. The start of the main() function is the starting point of a program. All functions have braces ({} that mark the beginning and end of the function. Between the braces lie the C statements that make up the body of the function. To execute, or carry out, the code within a function, the program calls, or invokes, the function.

A special kind of function is a Toolbox function. The Toolbox functions were written by Apple and are included with your computer. You can use the C language to access them, thereby saving the programming effort of writing them yourself.

A declaration statement is one that sets aside memory for a variable. A variable is a name you associate with some data. A variable can take on different values. To give a value to a variable, you use an assignment statement.

An important phase of writing a program is debugging it. When you take the time and effort to remove all sources of error in a program, you are debugging the program code.
Chapter 2

Introduction to Macintosh Programming

The Macintosh computer is a very different breed of computer. So it should come as no surprise to you that the languages used to program it are also of a different breed. ANSI C was a language created for text-based computers. The Macintosh is a graphics-based computer. So to program on the Mac, the boundaries that defined ANSI C had to be expanded.

You want to be able to do more than write words across the Mac’s screen. You want to be able to create windows, pull-down menus, and pictures. You want to create programs that look as if they were written expressly for the Macintosh computer. The Simulator software that accompanies this book will enable you to do just that.

You’ll learn about the Toolbox—the collection of routines Apple has supplied to every owner of a Macintosh. You will read how resources allow you to easily create windows, menus, and dialog boxes.

This chapter introduces you to some of the broad concepts that are unique to Macintosh programming. We devote the remainder of this book to teaching you the specific fundamentals needed to create a true Macintosh program.
Why the Mac Is Unique

The Macintosh has a look and feel all its own. Back in 1984, when the Macintosh was introduced, the Mac's graphical interface was what set it apart from all other computers. The Mac graphical user interface, or GUI, consists of menus, windows, and icons. You use a mouse to point to items on the screen and to make selections. Many computers currently on the market have a graphical interface, but the Mac’s interface is still the one against which all others are measured.

Windows, mouse-input, icons—the design of the Mac makes the user comfortable with computing. As a Macintosh programmer, you'll want to be a part of the spirit of the Macintosh by including all of these user-friendly concepts in the Macintosh programs that you create. To aid you in this, Apple has made two very powerful utilities available for your use. We touched on one of these utilities, the Macintosh Toolbox, in Chapter 1. The second way to program the Macintosh is to use resources.

The Macintosh Toolbox

Memory is the area where a computer stores information. ROM, or read only memory, is the memory that is necessary for operating the computer itself. It contains special information that a programmer can access but cannot change. The Macintosh User Interface Toolbox, more commonly called the Toolbox, is the name given to a set of routines built into the ROM of every Macintosh computer. Each routine is a small program in itself. Each of these small programs devotes itself to a single purpose. For example, one Toolbox routine displays a window, and another draws a line in a window.

The Toolbox routines exist to make life easier for you, the programmer. As a programmer, you have access to these routines through the language you use—including C. Using these built-in tools will save you considerable programming time and effort.

The functions that make up the Toolbox are grouped by related purpose. The functions that exist to create and manage menus are collectively known as the Menu Manager. The Window Manager holds the functions used to create, move, and update windows. The Macintosh system software is the program that starts up your Macintosh computer and keeps it running. Over the years, Apple has improved the system software that runs your Macintosh by adding more features and upgrading existing ones. This has led to an increase in the number of managers. We'll say a little more about the most commonly used managers in later chapters. Figure 2–1 shows just some of the many Toolbox managers.
Because programming can be a difficult art to master, you'll want all the help you can get. The Toolbox routines provide that help. Throughout this book, and throughout the Simulator software, we'll make extensive use of the Toolbox routines. And, of course, we'll let you know when we do.

**Resources**

All Macintosh programs use resources. Each resource contains descriptive information about one part of the Mac interface. The menu bar of a Macintosh program has a resource that defines what menus, such as File and Edit, appear in it. And each individual menu has a resource that defines what items, such as Cut and Paste, are in it. A window has a resource that defines its size on the screen, and a dialog box has a resource that defines its size, too.

Resources are *templates* that you can use over and over again. Because each resource template is reusable, creating new Macintosh programs is easier. Modification of existing programs is easier as well. Think back on the various Macintosh programs you've used. How many of these programs make use of the same items under the Edit menu? If you look at the resources of some popular programs, you see that some of their resources are identical to one another.

Resources are stored in a resource file. This is a separate file from the one containing the source code you write. Each resource in the resource file has a four-letter type and an identification number. As an example, a resource file could contain a MENU resource with an ID of 1.
A moment ago we discussed the Toolbox and the idea that the Toolbox is actually a large collection of routines. These routines rely on resources. For example, to create a menu, you use the Toolbox routine GetMenu(), which requires that you specify the menu resource you want to use. Figure 2–2 shows how you tie your program’s code, the Toolbox, and resources together.

The information that makes up a resource is not C language code. It can, however, be edited—though not in the same manner that source code is edited. If resources are not C language code that you can include in your programs, how do you create them? Again, Apple makes it easy for you. Apple has a program called ResEdit that allows you to create new resources and edit existing ones. Apple wants developers to use ResEdit to give Macintosh programs a consistent look. For that reason they distribute the program freely. Many compilers, such as THINK C, include a copy of ResEdit. We’ll rely on ResEdit throughout this book. If you’re not familiar with ResEdit, don’t be alarmed; we’ll give plenty of examples of its use. And Chapter 13 will cover ResEdit in detail.

The Structure of a Macintosh Program

When a Macintosh program is running, it spends most of its time waiting. What is the program waiting for? For the user to tell it what to do. It is the
user who initiates some sort of program action by entering information via the keyboard or the mouse. Each time the user performs such an action, the program interprets it as an event to which it must respond. After responding, the program again waits patiently for the next event. This reliance on the occurrence of an event before taking action makes all Macintosh programs event-driven.

When the Macintosh recognizes that the user has performed some action that it interprets as an event, it doesn't respond immediately. Instead, it stores the event information in a holding area, called the event queue. The computer can then process events in an orderly manner after finishing the task it is currently performing. The event queue allows the Macintosh to keep track of what it should do next. The ability to examine and process more than one event at a time allows the Macintosh to properly handle an action such as the double-clicking of the mouse button, which is really more than one event.

Event-driven Macintosh programs share a similar structure. A program starts by initializing the Toolbox. This involves initializing global variables and internal data structures—concepts we'll discuss later. Next, the program waits for an event. This could be mouse or keyboard input, a disk insertion, or any one of several other actions. When an event does occur, the program retrieves and processes the event. How it processes the event is dependent on the type of event. These last two steps, retrieving and processing an event, are repeated over and over. Only when one of the events tells the program that it should end, or quit, will the processing of events cease. We refer to this repetition as an event-loop. Figure 2–3 diagrams this cycle.

This Book’s Approach to Programming

Macintosh applications can be complex entities. One factor that makes Macintosh programming difficult is the creation of the event-loop. Other factors adding to this complexity are the interaction of windows and dialog boxes and the handling of menu selections. Viewing it as a whole, you may be starting to get the impression that the development of a working Macintosh program is a hopelessly complicated ordeal. Cast your doubts aside and read on. We are about to introduce a method that will make this whole process much easier.
Now that you are familiar with the format of a typical Macintosh program, go ahead and forget about it. No, not for good. Just for several chapters. We’re going to teach you to program by removing the barriers that prevent you from concentrating on specific details. For now, you’ll forget about event-loops and the interaction of screen elements. You will learn individual concepts, one at a time. You’ll write code fragments—short pieces of code focused on one topic, not entire programs. When you’ve mastered each of
the basic concepts, you'll put them all together to form a real Macintosh program.

At this point, a question or two may come to mind. If you aren't going to start by creating a working program, how can you learn the C language techniques that allow you to bring windows, dialog boxes, and menus onto the Macintosh screen? You are going to want to experiment by changing values and adding or subtracting lines of code. If you write only code fragments, how can you see the results of your experimentation? The answers lie in the Simulator software that accompanies this book.

The Role of the Simulator Software

Besides being an on-screen tutor, the Simulator software is your programming shell. It will take care of the nitty-gritty programming details like event-loops. You get to start immediately with the fun stuff, like drawing shapes in a window. The Simulator software will first present a programming concept. You will then have the concept reinforced by viewing a simulation or by entering a line or two of code. The software will execute your code on your computer screen without the need for you to create an event-loop. The software does the work; you get to play.

With the Simulator software doing the dirty work, you can concentrate your efforts on the individual topics that make up a Macintosh program. If you first learn the details involved in creating graphics, menus, windows, and dialog boxes, you'll have a better understanding of what's going on when you finally put it all together to form a Macintosh application.

Using Pseudo Code

We've said that our approach is to cover individual concepts one at a time and then tie everything together at the end. Our goal is to avoid introducing complicated topics too early. When we do have to mention a difficult concept before its time, we'll use an aid that all professional programmers use at one time or another. The aid is called pseudo code.

In the first chapter we discussed the six phases of writing a C program. Phase 1 was to determine what the program is to do. There we recommended that you think in general terms. We warned that those who neglect the planning stage are condemned to hours of lost time, confusion, and frustration resulting from tracking down bugs and correcting sloppy code. We said you should jot down the objectives, or goals, of your program and outline the design. Now, we'll formalize this planning stage a little.

Pseudo code is the combination of a programming language with informal English statements. It is a means of capturing the essence of your
ideas for a program without getting bogged down in the details of the exact method of achieving your goal. In pseudo code, the particulars of programming, like function and variable names, become unimportant.

As an example, think back to the Chapter 1 example where the goal was to create a program that organizes the user's music collection. The program pseudo code might read something like the following:

Begin Loop
    Ask User For Title Of CD
    Ask User For Group Name
End Loop
Ask User How To Sort: Alphabetically Or By Group
Sort The CDs
Write CD List To Screen

Notice that there is no actual computer language code in this version. Assuming that you knew the C language, you might want to include some code in your pseudo code. You could replace the Begin Loop and End Loop statements with a C language for loop, which we discuss in Chapter 6. Here's a second version of our pseudo code, this time with a little bit of C added:

for (count = 0; count < number_of_cds; count++)
{
    Ask User For Title Of CD
    Ask User For Group Name
}
Ask User How To Sort: Alphabetically Or By Group
Sort CDs
Write CD List To Screen

Notice that the above pseudo code contains both code (a C language for statement) and English phrases. Also notice that tasks (such as sort the CDs) are listed, but there is no mention of specifically how these tasks will be accomplished. The pseudo code is an outline of what must be done. It does not tell exactly how to do it. There is no right or wrong way to write pseudo code. As long as it conveys information about the steps needed to achieve a goal, the pseudo code is correct.

We'll use pseudo code from time to time throughout the remainder of this book. We'll do so when we want to introduce a topic without getting involved in programming details. Pseudo code will allow you to see how a
single topic fits into the big picture—the complete Macintosh program. There is also one very big side benefit to all of this. Exposure to this technique will get you into the habit of jotting down your ideas before you jump right into writing source code. You'll find yourself planning before coding, and your reward will be a saving of countless hours of trying to track down errors that result from poor coding and quick fixes.

Programming with THINK C

The Simulator software included with this book allows you to experiment with Macintosh C code. When it comes time to create a full-blown Macintosh program, you'll want to get a Macintosh C compiler. The Macintosh C taught in this book applies to most Macintosh C compilers that are available. Because the THINK C compiler is very popular with Macintosh users, we've included some specific instructions and tips on its use in later chapters. In particular, Chapter 19 and Chapter 20 give examples of working in the THINK C environment.

Chapter Summary

Unlike many other computers, the Macintosh computer relies on graphics rather than text. Because of this, Macintosh C code differs from C code used to program other computers. Macintosh C allows you to add all of the program features users expect in a Macintosh program—windows, menus, icons, and more.

Two tools aid you in your programming endeavors. The first is the Toolbox, a collection of routines written by Apple programmers that simplify the task of creating Macintosh-specific programs. The second tool is the resource, a reusable template that contains descriptive information about one part of the Mac interface, such as a menu or window.

A Macintosh program is said to be event-driven. That means the program waits for the user to perform some action, such as clicking the mouse, before it carries on.

Accompanying this book is the Simulator software. This tutorial software allows you to learn Macintosh C one step at a time.
Data is information. Computer programs, no matter how simple or complex, need data. Data is most often thought of as numbers, but data can also exist in the form of letters or words. Once a computer has some data, perhaps typed in by the user, we expect the computer to do something with it. Any time a computer program adds numbers or sorts a list, it is working with data. And when it takes some information and works with it to return new information to you, we say that the computer is manipulating data.

In this chapter you will learn about the different types of data, starting with the two major categories of data—integers and floating point numbers. You’ll examine the varieties, or subtypes, of these major groups.

You’ll see how to choose names for your data by declaring variables and how to give values to those variables. And you’ll learn what types of variables to use in different circumstances.

Here you will learn how to convert data from one type to another. Finally, you’ll see the C method for setting a symbol to a value to add clarity to your source code.
This chapter introduces you to the fundamental C data types. All of the data types that we discuss in this chapter are ANSI C types. That means that they are used in both Macintosh C programs and non-Macintosh C programs. It also means that all of these data types are used by your friends who are unfortunate enough to be programming on IBM-compatible computers! In future chapters we'll introduce you to a few new data types that are specific to Macintosh programming—data types exclusive to the Macintosh. We'll use these new types to unleash all of the graphics power residing in your Macintosh. Figure 3–1 shows you the difference between the two data types.
Data Types in Action

We start with a sample program to introduce you to a few of the data types available in C and to show them in use. As always, we don't intend Listing 3–1 to be a complete Macintosh program listing.

**Listing 3-1  Data types in action**

```c
main()
{
    int num_right;
    float score;
    char grade;

    /* make a window here... */

    grade = 'A';
    num_right = 7;

    score = num_right * 12.5;

    if (score > 80.0)
    {
        DrawString("You get an ");
        DrawChar(grade);
    }
}
```

Because you haven't been exposed to the code needed to create Macintosh windows, we simply added a comment to the code fragment to point out where code would go to make a window. If you were to run the program, a window would appear on the Mac's screen, along with one line of text. This is shown in Figure 3–2.

**What's New**

Listing 3–1 contains several elements that are new to you. Let's look at them now.
Data Types
You’ve already seen the int data type before, in Listing 1–1. The floating-point type (float) and the character type (char) are new. The float type lets you work with numbers that contain decimal points—something the int type cannot handle. Variables of the char type can hold characters—such as letters of the alphabet.

Constants
Listing 1–1 introduced you to an integer constant in the line num=1;: In addition to an integer constant, the new listing has a floating-point constant and a character constant, as shown in Figure 3–3.

Math Operation
We’ve given you a hint as to how math operations are performed using C. The line of code that assigns a value to the variable score multiplies two numbers:

score = num_right * 12.5;

Displaying Text
The listing in Chapter 1 used the DrawString() function to display a line of text in a window. That same function is used again here. This example also makes use of a second text-displaying function, DrawChar(). This function displays a single character in a window.
Grouping Lines of Code
Like Listing 1–1, this listing makes use of an if control statement. Unlike Listing 1–1, Listing 3–1 uses braces after that statement. This allows more than one line of code to be included under the if statement. The lines of code between the braces are associated with the if statement.

The remainder of this chapter examines data types and constants. Chapter 12 explains DrawString() and DrawChar(). Chapter 7 explains the use of branching statements like the if statement we use here.

Variables and Constants
We introduced this chapter by saying that data is information. Before a computer can give us the information we request of it, we must supply information for it to work with. Once supplied with the necessary data, a computer can do just about anything with it: perform simple or complex math operations, sort lists, draw pictures, and run simulations. The data a computer needs may be in one of two forms. We call data that are preset at the start of a program and never change value constants. The second group of data has the ability to change in value as a program runs. Data in this group are called variables.

Listing 3–1 has both variables and constants. The three variables are num_right, score, and grade. Each can be, and is, assigned a value as the
program runs. And each can be reassigned a new value later in the program if we decide to do so. The constants in the listing are the letter A, the integer constant 7, and the floating-point constants 12.5 and 80.0. These values are set and will not change.

Now that we know that data can be either constant or variable and that it has a type, we’re ready to look at the different data types available in C.

**Why Different Data Types?**

When you declare a variable in a C program, you do so to let the C compiler reserve, or *allocate*, the amount of computer memory needed to hold the value that variable may take on. Different variable types require different amounts of memory. Figure 3–4 illustrates this.

To reserve memory, the compiler does not have to know the value of the variable, just its type. Knowing the variable type gives the compiler the information it needs to reserve the proper amount of memory. As an analogy, let’s look at the case of a man who has limited car parking at his house. He’s having several guests over for the afternoon, and he’s trying to organize a parking arrangement for them. Should he be concerned with the make of a car or just its type—large or small? Like the compiler, he cares only about the type, not the “value.” See Figure 3–5.

---

**FIGURE 3–4** Memory is reserved based on variable type.
The C language has several data types. Because you will have the most interest in the types that get extensive use in Macintosh programming, those are the types to concentrate on.

**The int type**

An integer is a whole number. An integer does not have a decimal point. Some examples of integers are:

0 1 2 -10 -99 412 8390

Numbers that have decimal points, or fractional parts, such as 1.44, -7.2, and 893.500, are not integers.

The int type is often called the fundamental data type in C. You will find that you use it often. The int type is signed, meaning that it can have a positive or negative sign in front of it. Thus, a variable of type int can take on a negative value, zero, or a positive value. On a Macintosh, the C compiler sets aside enough memory to hold an integer that lies in the range of -32768 to +32767.
Declaring an int

To declare a variable to be an integer, use the `int` keyword followed by a variable name and a semicolon. Recall from Chapter 1 that keywords are words that are part of the C language and, as such, are reserved by the C language. All variable types that you encounter will be C keywords. The main point you should remember about keywords is that you may not use them as variable names:

```
int num;          /* a legal variable name*/
int an_int;       /* legal: int can be part of a name */
int int;          /* illegal! int cannot be a variable name */
```

To declare more than one int variable, the `int` keyword can be followed by several variable names. Variable names are separated by commas:

```
int knives, forks, spoons;
```

The above declaration could also be done on three lines:

```
int knives;
int forks;
int spoons;
```

Which is the better method? The choice is up to you. We prefer to list variables separately, one per line. This makes it very clear what variables are of what type. And it has the advantage of allowing for a descriptive comment after each declaration.

Declaring a variable reserves memory for the variable, but it does not give it a value. As you have already seen, you can assign a value to a variable at any point in a program by using an assignment statement:

```
knives = 7;
```

Initializing and Assigning an int

A second way in which you can give a variable a value is by initializing it. Initializing a variable gives it a value during the variable's declaration. To initialize a variable, include the assignment operator (=) and the value you want the variable to have:

```
int knives = 7;
```
This has the effect of both reserving memory for the variable and placing a value (the number 7) into that reserved memory. As with declarations, you can initialize more than one variable on a single line:

```c
int knives = 7, forks = 12;
```

Our preference, again, simply for clarity, is to place a single initialization on a line. The following gives the same result as the one-line initialization:

```c
int knives = 7;
int forks = 12;
```

### The long Type

A variable of type long, like an integer, is a whole number without a decimal point. What can the long type do that the int type can't do? Declaring a variable to be type long causes your C compiler to reserve more memory than it would for an int type. And the additional memory can hold a larger value than the standard amount of memory set aside for an int type can hold. On a Macintosh, the range of numbers that a long variable can have is -2,147,483,648 to +2,147,483,647.

If a long holds the same kind of value as an int, only with an extended range, why not just play it safe and declare all integer variables to be of type long? The answer has to do with memory. A long uses up more memory than an int. And if you don't need the extended range of a long, there is no need to reserve the extra memory.

You declare, initialize, and assign a long variable just as you do an int, except you use the long keyword. Below are some examples:

```c
long atoms;
long num_people;
long miles = 212765;
atoms = 4500000; /* an int couldn't hold this number! */
```

### The char Type

To store characters, C uses the char type. Because computers work with numbers, not letters, characters are converted to numbers and then stored in memory. Each character has a number associated with it. A computer uses the ASCII code to keep track of which character is represented by which number. ASCII is an acronym for the American Standard Code for
Information Interchange. That's quite an eyeful, and it's unlikely you'll ever see those words again. But we didn't want to leave you wondering.

The Macintosh, and most other computers, use the ASCII code. Figure 3–6 shows a part of the ASCII table. The figure shows that the Macintosh C compiler views the letter B as the integer 66. Because the variable letter was declared to be of type char, the compiler will know to treat this 66 different from an int variable with the value 66. What does all this boil down to? As it turns out, the char type is actually an integer type.

When you use a character in the C language, you normally don't have to worry about what integer value the computer views the letter as. If the time does come when you do need to know, you can refer to the ASCII Table, which we've included in its entirety in the Appendix.

### Assigning a char

At any point in your program you can assign a new value to a char variable through the use of an assignment statement. Enclose the character in single quotes.

---

**FIGURE 3–6** Characters are converted to integers.

<table>
<thead>
<tr>
<th>ASCII Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>! 33</td>
</tr>
<tr>
<td>* 34</td>
</tr>
<tr>
<td># 35</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>A 65</td>
</tr>
<tr>
<td>B 66</td>
</tr>
<tr>
<td>C 67</td>
</tr>
<tr>
<td>D 68</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>a 97</td>
</tr>
<tr>
<td>b 98</td>
</tr>
<tr>
<td>c 99</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

```
char letter = 'B';
```

© 1984, 1989 The Macmillan Publishing Company. All rights reserved.

40 Think THINK C
char high_grade;  /* declare a char variable */
high_grade = 'B';  /* assign it a value of B */

It's important to notice that single quotes are used in char assignments. In previous chapters you saw that strings, as used in DrawString(), are enclosed in double quotes. Don't confuse the use of single and double quotes:

high_grade = 'B';  /* this is correct */
high_grade = "B";  /* wrong, can't assign using a string */
high_grade = B;  /* wrong, thinks B is a variable */

Now that you know the computer views characters as integers, the fact that the following assignment is legal might not surprise you:

high_grade = 66;  /* the ASCII code for B */

Although this is legal, we recommend that you use the actual letter in assignment statements, rather than its ASCII equivalent. This will avoid confusion for other people who might someday look at your code.

Initializing a char
As we demonstrated in Figure 3-6, char variables can be initialized at the time of declaration:

char letter = 'b';
char question = '?';
char exclaim = '!';
char four = '4';
char fail = 'F', pass = 'C';

As the above shows, char variables do not have to be letters. Any keyboard character is considered a valid char, and each has an ASCII equivalent.

The float Type
Integers are simple and useful data types to work with. But as we've stressed, they cannot be used with numbers that contain decimal points. Numbers that contain decimal points are floating-point numbers:

3.14159  1267.2  -8.5  32.98  0.23456
To handle such numbers, C uses the data type float. Other computer languages, such as Pascal, call this type *real numbers*. Here are some examples assigning values to float variables:

```c
float miles;
float population;
float tiny;
float atoms;
float loss;

miles = 513.5;
population = 245000000.0;
tiny = 0.000008;
atoms = 5101234500.0;
loss = -10.42;
```

### Type Conversion

The C language is a liberal language when it comes to using variables of different types within a single line of code. Some computer languages will not let you write *mixed mode* statements—statements like the following that use different variable types:

```c
int int_num;
float float_num = 3.0;

int_num = float_num + 2;  /* int_num becomes 5 */
float_num = int_num + 6.5; /* float_num becomes 11.5 */
```

The above results in `int_num` equaling 5, an integer, and `float_num` equaling 11.5, a floating-point. The C language converts the result of the statement to the type of the variable to which the result is assigned.

Although C allows mixed mode statements, it is good programming practice to try to avoid them. The danger lies in the assignment of a larger type to a smaller type. A floating-point variable can hold a number much larger than an integer type can hold, so statements like the following can be unpredictable:
int int_num;
float float_num = 89100.0;  /* bigger than max int */
int_num = float_num;       /* int_num becomes ?? */

The maximum value an integer can have is 32,767. So the assignment of 89,100 to an integer type is asking for trouble. Most C compilers would give int_num the lowest integer value, -32768, which is not the intended result. A second problem with assignments of floating-points to integers is that C compilers truncate the result. That is, any decimal part is dropped off:

int int_num;
float float_num;

float_num = 32.1;
int_num = float_num;       /* int_num becomes 32 */

float_num = 32.98;
int_num = float_num;       /* int_num again becomes 32 */

The lesson here is to keep results predictable, avoid mixed mode statements if at all possible.

While it is best to avoid converting types, you may find it necessary at times. When this happens, it is best to take control and make the conversion yourself. To do this, C allows you to cast a variable of one type to another type. To change a variable from one type to another, precede the variable name with the name of the type to which you want it converted, enclosed in parentheses:

int int_num;

int_num = 4.8 + 5.9;       /* int_num becomes 10 */
int_num = (int)4.8 + (int)5.9;       /* int_num becomes 9 */

Without using the cast operator, int_num first becomes 10. The result of 4.8 added to 5.9 is 10.7, which is truncated to 10. By using the cast operator on each of the two floating-point numbers, we treat them individually as integers. The result is the addition of two truncated numbers: 4.8 becomes 4 and is added to a truncated 5.9, or 5. The result is 9.

In future chapters we'll give specific examples of when casting is useful and even necessary.
Symbolic Constants: \texttt{\#define}

Now that you know what a constant is, you're ready to make use of a C feature that deals specifically with constants. Look at the following fragment:

\begin{verbatim}
#define STATES 50

main()
{
    int continental_states;

    continental_states = STATES - 2;
}
\end{verbatim}

The \texttt{\#define} statement sets a name to a value. In the above, the \texttt{\#define} makes the name \texttt{STATES} equal to 50. What makes the \texttt{\#define} special is that you can now use the name \texttt{STATES} anywhere in your program in place of the number it equals. So the line:

\begin{verbatim}
continental_states = STATES - 2;
\end{verbatim}

is the same as the line:

\begin{verbatim}
continental_states = 50 - 2;
\end{verbatim}

\textbf{IMPORTANT}

Notice that a \texttt{\#define} statement contains no equal sign (=) and does not end with a semicolon (;).

One advantage to the use of \texttt{\#define} is clarity. When looking through page after page of code, numbers can become meaningless. A name adds clarity to the code. That was our reasoning when we suggested that you use descriptive variable names like \texttt{taxes} and \texttt{total\_score} rather than \texttt{x} and \texttt{y}. We use \texttt{\#defines} for the same reason. Figure 3–7 shows two listings, both of which do the same task. Listing 1 uses nondescriptive variable names and no \texttt{\#defines}. Which listing appears clearer to you?
Another reason to use #define statements in your code is maintainability. If your final program is a good one, it may be in use for years. Some things, like tax rates, don't remain the same for years. To maintain your program, you will have to change the source code if the tax rate changes. If you use a #define to establish your tax rate, you will have to make only one change—to the #define statement itself—in your program. Without the #define, you would have to search through all of your source code, looking for occurrences of 0.05 and changing them to the new rate—0.07, for example. And could you be sure that every 0.05 represented the tax rate and not something else? Figure 3–8 illustrates this.

In Listing 1 in Figure 3–8, the number 0.05 appears twice in main(). If the tax rate changes to 0.07, you might mistakenly think both occurrences of 0.05 should be changed to 0.07. But in fact, only one of the 0.05 values represents the tax rate. The other represents a rate of loss from theft. Listing 2 in the same figure makes it obvious that this is the case. In Listing 2, all you have to change is one line:

```
#define TAX_RATE 0.05
```

is changed to:

```
#define TAX_RATE 0.07
```

For a large program, the saving in time and effort should be very apparent. Use #define statements at the very start of your source code every chance you get. We'll close with a few more examples:
Computers work with data. To take advantage of the computer's power to manipulate, or work with, data, you need to know about the different data types available in C.

Data that are given a value at the start of a program and never change value are called constants. Data that can change value during the course of a program's execution are called variables.

Different data types occupy different amounts of memory. And different types are best suited for different tasks. The data type of a variable tells the compiler how much memory to reserve for that variable.

The int data type holds integer, or whole number, values. The long type also holds whole numbers, but it can hold larger values than the int type can hold.
A variable of type char holds a character. Computers work with numbers, not characters. For this reason a char is converted by the computer to an int. The conversion is made by means of the ASCII table—a table that matches, or maps, each character with an integer value.

A variable of type float is a floating-point number. Floating-point numbers are also known as real numbers. A floating-point number contains a decimal point. A float contains a fractional portion that lies to the right of the decimal point.

Variables of all of the above listed types can be declared, initialized, and assigned values.

The C language #define statement sets a name to a value. Once set, the name, rather than the number, can be used throughout your code. If you make the name descriptive of what the number stands for, your code will be much easier to read and modify.
Structured coding is a style of writing computer code that is highly readable and understandable. The C language encourages structured coding. One of the primary concepts that enable structured coding to work is the function. A function is a small section of code that performs a specific duty—it has a specific function.

This chapter introduces you to functions—how to write them and how to use them. Functions can use any and all of the elements of the C language. At this point we have touched on only a few of the many elements that make up the C language, so any functions you create now will be quite simple. Future chapters will enable you to take better advantage of the power of functions. Because functions are such an important part of programming, we want you to become familiar with them early in the game.

You'll learn how to call, or invoke, a function. You will see how to use toolbox functions that you write. And you will learn to appreciate the advantages of functions.
FIGURE 4–1 Components of a function

Flashback: What a Function Is

In Chapter 1 you had your first look at a function—the main() function. We introduced you to main() by telling you that functions are the building blocks of a C program. We also told you that a function is C code grouped between braces and given a name. Functions are also referred to as routines; the two words are used interchangeably.

You can spot a function by the parentheses that follow the function's name. Between the braces that define the start and end of a function are lines of code. Figure 4–1 summarizes what you know about functions at this point.

Calling All Functions

The main() function is the starting point of a C program. So all C programs must contain the main() function. Most programs, though, don't stop
there. They can, and almost always do, contain other functions as well. Below is a very simple function that draws a line of text on the screen. It makes use of the Toolbox routine DrawString() that we introduced in Chapter 1:

```
Draw_Hello()
{
    DrawString("\pHello!";
}
```

To make use of the Draw_Hello() function or any other function, you call, or invoke, it. To call a function, you give the function name followed by parentheses and a semicolon. Figure 4-2 emphasizes the difference between calling and defining a function.

When we talk about source code, whether it be C code or any other language, we speak of lines of code. Figure 4–3 shows a very simple program. To the left of the listing, we’ve numbered each line of code. We do this here to help you follow the path the program takes. In C, lines of code are not numbered. Notice that we didn’t number the blank lines. Blank lines are added to code for clarity—they do not affect the code itself.
Programs execute, or run, line by line. The exception to this is when a conditional statement is met or when a function call is encountered. We cover conditional statements in Chapters 6 and 7. In this chapter we cover the program flow as it pertains to functions. Figure 4–4 shows the flow of control in a very simple program that contains no functions other than main().
Figure 4–4 shows how each line of code executes in sequential order from the first line to the last. The arrowheads by the line numbers indicate that those lines of code have been executed. The next example, Figure 4–5, modifies the program of Figure 4–4 to include a function call. Figure 4–5 shows the flow of control for the first several steps in the program execution.

Notice in Figure 4–5 that execution jumped from line 7 to line 10. When the program reaches the call to Animal_Message(), it searches the code listing for the definition of this function. Once found, the program will jump to the start of it, no matter where in the program the code is located. Line 8 will just have to wait.

Figure 4–6 shows the completed execution of the program.

Figure 4–6 shows that when a function call occurs, the program jumps to the start of the definition of that function, then executes that entire function. Only after the function executes does the program return to the point at which the function call occurred. After the call to Animal_Message(), the program resumes with line 8 in main() and continues on to line 9 to complete the program.

```c
1) main()
2) {
3)     int cats, dogs;
4)     int pets;
5)     cats = 2;
6)     dogs = 5;
7)     Animal_Message();
8)     pets = cats + dogs;
9) }

10) Animal_Message()
11) {
12)     DrawString("\n\nAll animals have values.");
13) }
```

Order of Execution: 1, 2, 3, 4, 5, 6, 7, 10

**FIGURE 4-5** Part of the flow for a program with a function call
Toolbox Functions

Up to this point, the focus has been on functions that you, the programmer, write. But there is also a large group of functions that are already written and available for your use—Toolbox functions. You've used one of them—the DrawString() function—several times already.

The Toolbox function definitions, that is, the function names and bodies, exist in your Macintosh ROM chips. Apple programmers who were very familiar with the workings of the Macintosh wrote these routines. Therefore, you do not have to include the function definitions in your own source code.

Every time you use the DrawString() function in your code, you are calling, or invoking, the DrawString() function:

```
DrawString("This line of code is calling a function!");
```

In Macintosh programming there are two types of functions: those you write yourself and the Toolbox functions that already exist. The func-
Main/


def main:
    def Draw_Hello:
    DrawString("\pHello!")

DrawString("\pToolbox call"); /* call Toolbox function */
Draw_Hello(); /* call your own function */

When you call the Toolbox function DrawString(), you are directly accessing a routine in ROM. When you call your own function Draw_Hello(), you are accessing your own routine, which is located somewhere in your own source code. Figure 4-7 elaborates on the flow of control during these function calls.

More Than Just DrawString()

The DrawString() function is just one of more than a thousand Toolbox routines available for your use. No, that isn’t a misprint; there are actually that many Toolbox functions. Another Toolbox routine is the MoveTo()}
A few of the many Toolbox routines

<table>
<thead>
<tr>
<th>Toolbox Routine</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoveTo()</td>
<td>Set QuickDraw to a screen location</td>
</tr>
<tr>
<td>LineTo()</td>
<td>Draw a line to a screen location</td>
</tr>
<tr>
<td>DrawString()</td>
<td>Draw one line of text</td>
</tr>
<tr>
<td>SetRect()</td>
<td>Define the size of a rectangle</td>
</tr>
<tr>
<td>FrameRect()</td>
<td>Frame, or outline, a rectangle</td>
</tr>
<tr>
<td>FillRect()</td>
<td>Fill a rectangle with a pattern</td>
</tr>
<tr>
<td>ShowWindow()</td>
<td>Show a window on the screen</td>
</tr>
<tr>
<td>HideWindow()</td>
<td>Hide a window</td>
</tr>
</tbody>
</table>

In the above example, the first call to MoveTo() prepares screen drawing to start at row 30 and column 20. A line of text is then drawn. The next call to MoveTo() sets the computer to draw to row 45 and column 20. These numbers are pixel values. Pixels are the small dots that your monitor screen is composed of. The effect is that of moving down one line.

Function Naming

As your source code file grows, you'll have many, perhaps hundreds of, calls to functions. If you've written several of your own functions, it might not be obvious which calls are to your routines and which are to Toolbox rou-
Toolbox routines contain no underscores

Give your own routines underscores

FIGURE 4-8 A function naming convention

routines. To make it readily apparent to you and to others, use a function naming convention. A convention is simply a rule or set of rules that you follow throughout your programming effort. The convention we present below is the best kind—simple and straightforward. When Apple named its Toolbox functions, it did not use any underscores in the function names. So when you name your own functions, always use at least one underscore. Figure 4–8 illustrates this convention.

Advantages of Functions

Functions are nothing more than lines of code grouped, or packaged, together and given a name. Except for main(), functions are not necessary or required in a program. Any program that contains functions could be re-written without them. For proof of this, look at the two program listings in 4–9.

Yes, we hear your shouts of protest! If both listings in Figure 4–9 achieve the exact same result, why expend the effort in creating a function? More code was added, and the flow of control was broken with no apparent benefit. In this case, you’re absolutely correct. But the examples presented here and elsewhere in this chapter are for demonstration purposes. As always, we like to keep examples as simple as possible so we do not detract from our goal.

With this all said, what then are the advantages of functions? A program that uses functions has several advantages over a program written without functions:
Reusability

Code tends to be repetitious—the same several lines of code are often repeated throughout a program. By writing one function and calling it repeatedly, you'll save programming effort and reduce the size of your program. Listing 4–1 is a code fragment for a program called MacMillionaire. It demonstrates reuse of a function. A description follows the listing.

LISTING 4–1  Calling the same function several times

main()
{
    /* If this were complete code we would open a window */
    /* here, then call: */
}
Listing 4-1 opens a window, then a dialog box, then another window. You can safely assume other events take place between the opening of the windows and the opening of the dialog box. The program is set up such that any window or dialog box that opens has a few lines of program credits displayed in it. So every time a new window or dialog box opens, the Draw_Program_Credit_Info() routine is called to write to the new window. Figure 4-10 shows what the screen might look like.

The example shows that a function can be called more than once; Listing 4-1 calls the same function three separate times. The body of our Draw_Program_Credit_Info() function contains eight lines of code. If this function was not in the listing, you would have to write those eight lines of code three separate times. By the way, if you write a program like the one in this example, please feel free to forward a copy to us!

Clarity

Functions can make your code clearer for yourself and others. It's easier to examine and understand small sections of code than it is to try to follow
pages of uninterrupted code. When a reader of your code comes across a line like this:

```
Draw_Program_Credit_Info();
```

that reader will have a good idea of what is to take place, without even looking at the definition of the function itself. Only if more details are needed will the reader have to page through your code and read the body of your `Draw_Program_Credit_Info()` routine. Here’s an example that shows how a couple lines of code using function calls would be easier to digest than pages of code:

```
if (draw_map_button_clicked == TRUE)
    Draw_Map_Of_United_States();
```

Here’s one more example:

```
main()
{
    Open_Dialog_Box();
```
Get_Input_From_User();
Process_User_Input();
Open_Window();
Display_Results_In_Window();

An Aid in Planning

Back in Chapter 1 we listed the phases of creating a C program. In Phase 1, we said, you should determine what the program is to do and list those tasks. Now that you know something about functions, you can use this knowledge to aid you in this planning phase. Without actually writing the functions, you can list the functions needed to carry out a program. The previous example is a case in point. We repeat it here:

Open_Dialog_Box();
Get_Input_From_User();
Process_User_Input();
Open_Window();
Display_Results_In_Window();

Without going into any detail, you get an idea of what your program should do. Writing and reviewing the function names gives you an opportunity to see if you have a good, solid understanding of what is expected of your program. Knowing the number of functions you'll have to create also gives you some indication of the amount of effort you will have to put into writing the program.

Debugging

When something in your program is not working as you intended, having your program broken into several functions will help you narrow your search for the cause of the problem. Figure 4-11 illustrates this.

Program Maintainability

When it comes time to change a feature in your program, you'll be able to find the affected section of source code easier if the program is written in separate functions.
main()
{
    Open_Dialog_Box();
    Get_Input_From_User();
    Process_User_Input();
    Display_Results_In_Window();
}

Open_Dialog_Box()
{
    ....
}

Get_Input_From_User()
{
    ....
}

FIGURE 4-11 Functions aid in debugging.

The Scope of a Variable

In Chapter 3 we covered one property of a variable—its type. When you declare a variable, you give it a type, such as int or float. Another property of a variable is its scope. Which functions know of the existence of a particular variable depends on where in your C code that variable is declared. The scope of a variable tells you which parts of your C program know of the existence of a variable.

Local Variables

We've said that a function can declare variables. When you declare a variable in a function, it is local to that function. Figure 4–12 shows three functions. Two of them have their own variables; one does not.

Variables declared within a function are local to that function. This means that only the function that declares them knows of their existence. In Figure 4–12, main() has one local variable, main_int. Function Function_One() has two local variables, func1_int and func1_float. Function Function_Two() has no local variables.
Only a function that declares a variable can use that variable in a C statement. Notice in Figure 4–12 that only `main()` uses `main_int`, and only `Function_One()` uses `func1_int` and `func1_float`. Listing 4–2 is an example of incorrect variable usage and would result in a compiler error.

**LISTING 4–2 Incorrect variable usage—out of scope**

```c
main()
{
    int main_int;
    Double_Number();
}

Double_Number()
{
    main_int = 3; /* Double_Number() doesn't */
    main_int = main_int * 2; /* know about main_int! */
}
```
What if you want more than one function to know about a single variable? Does C provide a way for the information contained in one variable to be shared by several functions? The answer is yes, and global variables are one way of doing so.

**Global Variables**

To make a variable known to more than one function, declare it outside and before all functions. This makes the variable *global*—it is known globally throughout the entire source code listing. Figure 4–13 shows both global and local variables.

In Figure 4–13 Total_Score is declared before any of the functions. That makes it global in scope. Any function can use it. Notice that both the main() function and the Add_To_Score() function use it. Variable score1, on the other hand, cannot be used by Add_To_Score(). It is declared inside the main() function, making it local in scope to main().
Variable Naming

In Figure 4–13 you may have noticed that we used uppercase letters in the name of our global variable Total_Score. This is not a C requirement. Rather, it is our way of distinguishing between local and global variables. By always starting each word of a global variable with an uppercase letter and each word of a local variable with a lowercase letter, we have a way to distinguish between the two. Now, if we have a long source code listing with many variables, we'll always know at a glance which variables are local and which are global. Figure 4–14 illustrates this convention; we recommend that you use it too.

Value Retention

One distinction between local and global variables is their scope. Another important distinction is in their ability to retain their values:

- **Global variables** keep their value for the life of a program or until changed within the program.
- **Local variables** lose their value when the execution of the function in which they are declared ends.

Look at Figure 4–15 to see an example of how local and global variables differ. Then read the explanation that follows.

When the program in Figure 4–15 calls Function_One() for the first time, it assigns global variable Global_Var a value of 5 and local variable func1_var a value of 3. After the first call to Function_One() is complete,
control returns to main(). The DrawString() line will execute next. At that point, Global_Var still retains its value of 5. Variable func1_var, however, no longer has a value of 3. The variable func1_var is local to a function other than main(), so main() does not know its value. In fact, func1_var no longer even exists! When function Function_One() is called for the second time, the function recreates local variable func1_var, which does not then have a value of 3. Its value will not become 3 until the assignment statement later in the function.

Local versus Global

By making a variable local to a function, you limit its power. So why should you ever use local variables? Why not just make all variables global so they always exist and always keep their values? The temptation is great, but there are several reasons why this is a poor programming strategy.

One reason to use local variables is to preserve computer memory. Your computer has a finite amount of memory that it can devote to the running of your program. When you declare a global variable, the computer allocates, or reserves, memory for it. This memory remains reserved and devoted to this global variable for the duration of the program. When you declare a local variable, the computer again reserves memory for it. But when execution of the local function ends, the computer destroys the vari-
able by releasing the memory, and the memory is once more available for your program’s use.

Many variables that you use serve a single, limited purpose. There is no need for the rest of your program to know anything about them. For example, C provides a way for the same block of code to execute several times in a row. We discuss such *looping* statements in detail in Chapter 6. Here’s a code fragment that shows a C loop:

```c
Loop_Three_Times()
{
    int count;

    for ( count = 0; count < 3; count++ )
        DrawString("\pLooping!");
}
```

The result of running the above code is that the `DrawString()` line will execute three times. Nowhere else in the program will you care about variable `count`. The program uses it here as a counter, or index, to determine how many times the looping code has been executed. The `count` variable is local to `Loop_Three_Times()`. Would there be any benefit to making it a global variable? No.

Reserve the use of global variables for variables that are used throughout your program. An example would be the use of the `My_Score` variable in Listing 4–3.

**Listing 4–3  Proper declaration and use of a global variable**

```c
int My_Score;

main()
{
    Get_My_Score();
    Compare_My_Score();
    Print_My_Score();
}

Get_My_Score()
{
    /* get My_Score from user */
}
Compare_My_Score()
{
    /* compare My_Score to some other score */
}

Print_My_Score()
{
    /* print My_Score */
}

Cheating at Windows

Chapter 8 contains a more complete discussion of functions than we've provided here. The intent of this chapter is to give you a basic understanding of functions. We had a second motive for introducing you to functions so early in this book. Books that are written to teach computer programming on text-based computers, like IBM-compatibles, can give you short, simple examples like the following:

main()
{
    printf("Hello, World.\n");
}

This example uses the ANSI C printf() function to print the line "Hello, World." to the computer screen. In the Macintosh world, things aren't quite so simple. With a Macintosh, you do not write text directly to the screen. Instead, you always write text to a window. Since we won't cover the creation of windows until Chapter 16, we'll write a function that creates a window and puts it on the screen, ready to be written or drawn to. We'll name the function Make_A_Window(). Then, our short example programs can include a call to this function:

main()
{
    Make_A_Window();
    DrawString("\pNow we're writing to a window!");
}
Now we’re writing to a window!

Figure 4-16 Using a window to convey information

If you were to execute the above example code, the result would be as shown in Figure 4-16.

What does the body of the Make_A_Window() function look like? We won’t concern you with that now. If you’re curious, you can look ahead to Chapter 16, where we develop the function.

**One More Function of Our Own**

You’ve seen that to display information to the user of a Macintosh program you need a window. Just as showing you output is not as easy as it is in a nonwindowed environment, getting values from the user is a little trickier as well. With a Macintosh this is usually done by putting up a dialog box that has an edit text box in it. This allows the program user to type input. Figure 4-17 shows an example.

To make upcoming examples more practical, we want to take advantage of the Macintosh’s ability to accept user input. So, as with our

Enter Your Bowling Score: 134

Figure 4-17 Getting user input from a dialog box
Make_A_Window() routine, we’re going to introduce one more of our own functions without showing you the C code we used to write it. The routine is Get_User_Input().

Our Get_User_Input() routine introduces one important property functions can have—a property that we haven’t yet discussed. A function is capable of returning a value. Not all functions return values. You must write the function in such a way that it does so. We’ll cover the topic of function return values in Chapter 8. Here, we’ll just let you know that a function is capable of returning a value to the function that originally called, or invoked, it. We’ll also show you how you can recognize a function call that returns a value. Figure 4–18 introduces this concept.

As Figure 4–18 shows, when a function returns a value, it has to return it somewhere. So when this function is called, the call is placed on the right side of an equal sign. The variable named on the left side (score in this example) will receive the returned value. When the function call is complete and the code of function Get_User_Input() has been executed, score will have a new value.

```
main()
{
    int score;
    Make_A_Window();
    score = Get_User_Input();
}
```

**FIGURE 4–18** Identifying a function that returns a value
Chapter Summary

In this chapter you were reminded that a function is a block, or group, of code that can be brought into action, or executed, by another block of code. When a block of code starts the execution of a function, we say that function is being called, or invoked.

Toolbox functions, like DrawString(), are functions that were written by Apple programmers and are built into the chips of your Macintosh computer. This is the only distinction from functions that you write yourself. You invoke Toolbox functions in the same manner that you invoke functions that you write yourself. With a Toolbox function, you don't get to actually see the code that makes up the body of the function.

A Macintosh C program could be written without ever writing a function other than the main() function. If you did that, you would be forgoing the advantages of functions. Advantages include reusability of code, more understandable code, easier debugging, and easier program maintainability.

Every variable has scope. The scope of a variable defines what functions know of that variable’s existence. A local variable is known only to the function that declares it. A global variable is known to all functions in a program.
Chapter 3 showed you that the C language has a variety of data types to choose from. To further aid you in your programming endeavors, the C language also gives you many ways to process the data you have stored in variables of these different types. Operators are the symbols that do the processing, and C has a rich set of them. The C language lets you perform math operations, make comparisons, alter the values of variables, and much more.

In this chapter you'll examine the most commonly used C operators. We'll also explain in greater detail two concepts—expressions and statements—you've used already.

Operators in Action

As we've done before, we'll start with a sample program. Listing 5–1 is a very small program that demonstrates just a few of the operators available in C.
LISTING 5–1  C operators in use

main()
{
    int dogs;
    int cats;
    int pets;

    Make_A_Window();

    dogs = 8;
    cats = 15;

    pets = dogs + cats;

    if ( pets > 20 )
        DrawString("Get a farm!");
}

If you ran the program, you would see a window with one line of text, as shown in Figure 5–1.

Listing 5–1 contains three different operators. You've encountered the first, the assignment operator (=), many times already. Here we use it twice:

dogs = 8;
cats = 15;

Get a farm!

FIGURE 5–1  Result of running Listing 5–1
Chapter 5  C Operators  75

The second operator used is the addition operator (+). It is used here to add two variables. The assignment operator is then used to assign to pets the sum of dogs and cats:

```
pets = dogs + cats;
```

The third operator used in Listing 5-1 is one of several C comparative operators, the greater than (>) operator. Here it compares the value of pets to the integer constant 20:

```
if ( pets > 20 )
    DrawString("\pGet a farm!");
```

We'll examine these three operators and numerous others in this chapter.

The Basic Operators

The C language has a wealth of operators, which is one of the main reasons C has become so popular. Operators give a programmer the freedom to carry out programming ideas in a straightforward manner. In this section you'll become familiar with the arithmetic operators. Because math operations are the most common and most familiar of all operations, the operators involved in mathematical operations are called the basic operators. Later chapters will introduce you to other operator types.

The Assignment Operator

By now you're certainly familiar with the assignment operator. Here, we want to formalize its definition. While it may be intuitive to read the following line of code:

```
dogs = 8;
```

as "dogs equals 8," this interpretation is not correct. The name of the operator gives you the clue as to how you should think of this operation: "the value 8 is assigned to the variable named dogs." If you're accusing us of being unreasonably picky, like your 10th grade English teacher was, give us a chance to defend ourselves. The distinction is important. The variable dogs is just that, a variable. Before this assignment statement, dogs may or may not have a value of 8. It could be 0, 3, or 8002. So to say that "dogs
equals 8” is not true. Only after the assignment statement is complete, after “the value 8 is assigned to the variable dogs,” can we say with certainty that “dogs equals 8.”

With assignment statements, the variable name always appears on the left, while the value being assigned to it appears on the right:

birds = 21;
sports_cars = 2;

To place the variable on the right side would not be valid, would not do us any good, and would make no sense.

7 = books; /* invalid! */

The Math Operators

There are four basic mathematical operators in C—the addition, subtraction, multiplication, and division operators.

You’ve already encountered the addition operator in previous examples. The operator is placed between two values for the purpose of adding them:

cost = 100 + 5; /* cost becomes 105 */

Here we added two integer constants. In arithmetic operations like addition, either one or both of the values to be added may be variables. The code fragment below demonstrates this idea:

int price = 100;
int tax = 5;
int cost;

cost = price + tax; /* cost becomes 105 */
cost = price + 5; /* in all three of */
cost = 100 + tax; /* these statements */

The subtraction operator is used in the same manner as the addition operator. It is placed between two values for the purpose of subtracting the second from the first:

good_fruit = total_fruit - rotten_fruit;
good_fruit = 120 - 35;
So far, arithmetic operations have been shown only on integers. You can also use operators on floating-point values:

```c
float price;
price = 100.0 - 5.0;
```

When you insert the *multiplication operator* between two values you tell C to multiply them. The following are examples:

```c
paper_area = 8.5 * 11.0;
total_cost = items * price;
inches = feet * 12.0;
```

Place the *division operator* between two values to divide the first by the second. Here are two examples:

```c
half = total/2;
cost_each = total/items;
```

The results of division depend on the types of the values involved. If you use two floats, the result will be a float. In the following example, `total` will be assigned a value of 3.2:

```c
float total;
total = 16.0/5.0; /* total will become 3.2 */
```

If you use two ints in division, the result will be stored as an int—even if the answer has a fractional part. We call this *truncation*—the dropping of the fractional part of a number. In the following, `answer` will become 3. The intermediate result of the division, 3.2, will be truncated to 3:

```c
int answer;
answer = 16/5; /* answer will become 3 */
```

The C compiler forces the left side of an assignment to have the same type as the right side. So the above truncation will take place even if `answer` is declared to be a float:

```c
float answer;
answer = 16/5; /* answer will again become 3 */
```

What can be done in the above case? Make sure the values used on the right side are floats:
float answer;
answer = 16.0/5.0;       /* finally, answer will become 3.2 */

Although C will allow it, you should avoid the unexpected results that may occur if you mix types in division operations. Divide integers by integers and floating-points by floating-points.

The Sign Operators

When you use the minus sign on just one value, it is a unary operator—also called a sign operator. It indicates that the value is to be negative:

    float loss;
    int below_sea_level;
    int feet = 10;
    loss = -24.5;
    below_sea_level = -feet;  /* below_sea_level = -10 */

The same applies to the addition sign. When used with just one value or variable, it too is a unary operator. It is used to designate a value as positive:

    profit = +128;

When used as a sign operator, the addition sign does not change the value of the quantity it appears in front of:

    profit = +128;       /* profit becomes the same */
    profit = 128;        /* for both statements. */

While not necessary, you may find it desirable at times to use the + operator for clarity in your source code—to emphasize a positive value.

One Up, One Down

The operators covered in the previous sections are the most commonly used of the dozens of C operators. Now, we'd like to cover two more useful operators—the increment and decrement operators.
The Increment and Decrement Operators

The increment operator exists to carry out a task that you will perform over and over in your computer programs—increasing, or incrementing, a value by 1:

```c
++toys;
```

If `toys` has a value of 4 before executing the above statement, it will have a value of 5 after execution. The increment operator can be used in two different ways. If you place the operator before a variable, as above, C increments that variable before its value is used. This usage is referred to as the prefix mode. If you place the increment operator after the variable name, the operator is used in the postfix mode:

```c
++toys;
```

C won’t increment the variable until after using the variable in the statement in which it appears. When used on a single variable, the increment operator yields the same result, whether used prefix or postfix. In the following code, both `boys` and `girls` will have a value of 10 after the code is executed:

```c
int boys = 9;
int girls = 9;

boys++;
++girls;
```

When the increment operator is used in a statement containing more than one variable, its mode becomes important. Consider the following prefix usage:

```c
int total;
int counter = 5;

total = ++counter;
```

After a program executes the above statement, both `total` and `counter` will have a value of 6. This is because C first increments `counter` by 1, from 5 to 6, then assigns to `total` the value of `counter`. Here’s the same example, except we now use postfix mode:
int total;
int counter = 5;

total = counter++;

After executing the above statements, variable total will have a value of 5, while counter will have a value of 6. This is because the assignment takes place before counter is incremented. Only after counter is used in the statement does it get incremented by 1.

The decrement operator decreases, or decrements, a value by 1. The decrement operator works in the same manner as the increment operator. Like the increment operator, the decrement operator can be used in either the prefix or the postfix mode. Below are two examples:

total = --count;       /* prefix mode */
total = count--;       /* postfix mode */

But Are They Necessary?

In C, the following two statements yield the identical result:

count = count + 1;     /* first method */
count++;               /* second method */

And in a similar fashion, both of the following lines of code yield the same result:

count = count - 1;     /* first method */
count--;               /* second method */

If you can increment (or decrement) a variable using the first method, why create two new operators like ++ and --? One of the many benefits of the C language is the freedom it gives programmers—freedom to express their ideas in different ways and freedom to write compact, elegant code. Given the choice of incrementing in either of the above methods, we think you'll find yourself using the abbreviated ++ form.

Mathematical Precedence

When a statement contains only like operators, it is easy to see how the statement is evaluated into a final value:
cost = 6 + 2 + 5;      /* cost becomes 13 */

Adding the three numbers, in any order, results in a final value of 13. But what if a statement contains two or more different operators? Then the order of evaluation determines the final value:

cost = 6 + 2 * 5;      /* cost becomes ? */

Adding the 6 and the 2 gives 8, and multiplying that by 5 results in a final value of 40. But if you perform the multiplication first, then 2 times 5 gives 10, and 10 added to 6 gives 16. Which is the correct answer? The answer relies on a concept called precedence.

So that the execution of statements always yields uniform results, C has a set of rules for choosing which operations to perform first. The term for this ordering of evaluation is precedence. Each of the many C operators is assigned a level of precedence.

Multiplication and division are at the highest level of precedence, while addition and subtraction are at a lower level. Math involving the higher level operators is carried out first. When two operators are at the same level of precedence, such as multiplication and division, they evaluate from left to right. Figure 5–2 shows an example of the evaluation of a statement with more than one operator.

```
cost = 5.0 + 3.0 * 2.0 - 7.0;

Multiplication is of a higher precedence than either addition or subtraction
5.0 + (3.0 * 2.0) - 7.0
5.0 + 6.0 - 7.0

Addition and subtraction are the same level of precedence, perform from left to right
(5.0 + 6.0) - 7.0
11.0 - 7.0

Subtract:
4.0
```

**FIGURE 5–2** Example of evaluation using precedence
Expressions

An expression is a value or a combination of operators and values. Expressions can be quite simple, like these two:

7
-4

More involved, like these:

18 + 4
19.0 * 5.2 + 2.0
Or downright complex:

\[ 7.68 + ((6.22 \times ((42.54/3.7) - 7.125))/1.2) \]

Expressions do not have to contain only values along with operators. They can contain variables as well. After all, a variable represents a value. Here are four more examples of expressions:

- \( 100 + \text{bonus} \)
- \( 470.0 \times \text{tax\_rate} \)
- \( \text{dollars} + \text{cents} \)
- \( \text{dollars} + (\text{pennies}/100) \)

Every expression has a value. You can pretty well assume that if something in C has a value, it qualifies as an expression. All the examples given so far evaluate to a single result. Because you don’t know the values of the variables in some expressions, you may not be able to determine the final value of each expression. In a program, the computer would calculate a value for each expression based on the current value of each variable.

And no, we didn’t determine the value of the “downright complex” expression given above. We leave that as an exercise for you!

**Statements**

Way back in Chapter 1 we first mentioned statements. There, we said that a statement is an instruction to the computer. A C language statement ends with a semicolon. The following is a statement:

```c
winners = 5;
```

In the previous section we said an expression was anything that evaluates to a value. When you take an expression, or value, and assign it to something, you have a statement. Thus:

```c
7 + 8
```

is not a statement. But once you assign it to a variable (and end it with a semicolon):

```c
total = 7 + 8;
```
it becomes a statement—an assignment statement, in fact. You’ve also encountered the declaration statement:

    int total;

So far, you’ve encountered three types of statements, all of them as early as Chapter 1. They are:

- **Declaration** statements, which introduce a variable by giving its type and name.
- **Assignment** statements, which give declared variables values.
- **Control** statements, like the branching if statement, which provide different paths your program can execute. This type of statement is also called a conditional statement.

You’ll encounter still more statement types in later chapters.

**Chapter Summary**

Operators are the C language symbols that allow you to process, or manipulate, data. C offers you a full set of operators that help you perform math, make comparisons, and alter the values of the variables you work with.

The assignment operator (=) assigns a value to a variable. The addition, subtraction, multiplication, and division math operators (+, −, *, /) allow you to perform basic math operations on variables.

The sign operators (+, −) are used with a single variable to designate its value as positive or negative.

The increment operator (++) is placed prefix, or before, a variable to increase its value by 1 before its use in a statement. The same operator can be placed immediately after a variable name, or postfix, to increment its value after its use in a statement. The decrement operator (--) is used in the same manner as the increment operator. It decreases the value of a variable by 1.

To determine which of several operations should be performed first in a statement, C uses precedence rules. Multiplication and division are performed before addition and subtraction. Operations enclosed in parentheses have the highest precedence—they are performed first.
An expression is a value or combination of operators and values. An expression has a value. That is, it can be evaluated, or reduced, to a single value.

A statement is an instruction to the computer. Assigning an expression to a variable and ending the line with a semicolon creates a statement.
Chapter 6

Decision Making: Looping

One of the most valuable features of a computer language is its ability to make the computer perform easily tasks that human beings consider tiresomely repetitious. Looping provides a means of doing just that. You write a small amount of code and then, by adding just a line or two more, you tell the computer to repeat your code over and over.

Computers are only as powerful as the control you have over them. Looping is one means of controlling the computer’s actions. A second method is branching. In this chapter we’ll thoroughly cover the methods of looping that C offers. We’ll cover branching in the next chapter.

A Sample Loop

Before we examine each of the three C loop structures—the while, the for, and the do-while—we’ll start with a short program that demonstrates one of them—the while loop. Look at Listing 6-1 and see if you can determine what it does.
LISTING 6-1 Looping in C

main()
{
    int count = 0;
    Make_A_Window();
    while (count < 3)
    {
        ++count;
        DrawString("\pIn the loop.");
    }
    DrawString("\pOut.");
}

If you ran the code in Listing 6-1, you'd see a window like the one shown in Figure 6-1.

Let's take a closer look at Listing 6-1 to see just how the while loop works.

**Initialization**

```c
int count = 0;
```

This line is easy—declare an integer named `count` and give it an initial value of 0.

---

**In the loop. In the loop. In the loop. Out.**

---

**FIGURE 6-1** Result of running Listing 6-1
A Function Call

Make_A_Window();

Another easy one. This line calls a function. The call to Make_A_Window() executes the function named Make_A_Window(), assumed to be elsewhere in the listing. Then control returns to the next line.

The while Loop

while (count < 3)

Our first loop! The while loop tests a condition. If the condition is true, the statements below the while statement will execute. If it is not true, the program skips the statements. The while statement tests the condition that lies between the parentheses. In this example the less than operator (<) is used. We’ll cover this and other comparative operators later in this chapter. Is the value of variable count less than 3? It was initialized to 0, so you know this expression evaluates to true. Because the test passes, the program performs the statements that make up the while loop.

The Loop Block

{
  ++count;
  DrawString("\nIn the loop. ");
}

Statements between braces form a compound statement, or block of code. If the while statement passes its conditional test (count<3), then the block below it will be carried out.

The first line of the body increments count using the increment operator (++). Then a line of text is written to the window. When the closing brace is reached, the flow of control does not move on to the next line. Instead, it loops back up to the while statement and checks to see if the comparison is still true.

During the first pass through the loop block, variable count was incremented from 0 to 1. Because count is still less than 3, the same block of code will execute again.

This process repeats until the while loop test fails—count becomes equal to 3 or greater than 3. When the test finally fails (after three passes
through the loop block), the block of code is skipped and control moves on to the first line of code following the block.

You can see that it is vital to increment count at some point in the loop block. If count is not incremented, what happens? Variable count never becomes equal to or greater than 3, and the loop block repeats indefinitely—an infinite loop.

**Outside the Loop Block**

```c
DrawString("\pOut. ");
```

This is the line of code that follows the while statement block. It executes only after the while loop is complete—after the loop block has been carried out for the last time. It is not part of the while statement or block, so it is executed only once.

Congratulations! You've made it through your first loop. If some of the points still seem fuzzy, don't worry. The rest of this chapter provides more detail about the while loop and the other two C looping structures—the for loop and the do-while loop.

**TRUE and FALSE**

In Listing 6–1 the while loop tests a condition—is the value of variable count less than 3? If the condition is true, the body of the loop executes. If it is not true, the program skips the body. The condition is in the form of an expression, and as discussed in the previous chapter, all expressions evaluate to a value. In the C language, an expression that evaluates to 0 is considered false, and an expression that evaluates to any value except 0 is considered true.

If you could ask your C compiler what the value of false is, it would tell you 0. And if you asked it the value of true, it would tell you 1. Although any number other than 0 is considered true, C will assign to true the value of 1. Most C compilers, like THINK C, allow you to use the words TRUE and FALSE (in uppercase) in your C code, as if you had created them using #define statements:

```c
int c_is_fast;

c_is_fast = TRUE;           /* c_is_fast will have a value of 1 */
```
Why does C have this relationship between values and the words TRUE and FALSE? It becomes especially important when you consider C statements that involve an expression used as a comparison:

```c
if (pets > 20)

You may have noticed in the past that we called what lies between the parentheses an expression. Well, you now know that all expressions evaluate to some value. So the expression `pets > 20` must evaluate to a value. If `pets` is in fact greater than 20, then the expression evaluates to a value of TRUE, or 1. If `pets` has a value less than or equal to 20, then the expression will evaluate to FALSE, or 0.

**Comparative Operators**

All three of the C looping statements make use of a comparison expression. Looping statements compare one side of an expression to the other to determine the truth of the expression. To make the comparison, C offers several comparative operators. We list them in Table 6-1. For each operator, Table 6-1 lists two examples—one that evaluates to true and one that evaluates to false.

In the while loop program of Listing 6-1 you saw an example of the less than operator:

```c
while (count < 3)
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Expression Is True If:</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>left side less than right side</td>
<td>2 &lt;  5</td>
<td>11 &lt; -4</td>
</tr>
<tr>
<td>&lt;=</td>
<td>left side less than or equal to right side</td>
<td>3 &lt;= 14</td>
<td>6 &lt;= 5</td>
</tr>
<tr>
<td>&gt;</td>
<td>left side greater than right side</td>
<td>32 &gt; 31</td>
<td>0 &gt; 3</td>
</tr>
<tr>
<td>&gt;=</td>
<td>left side greater than or equal to right side</td>
<td>34 &gt;= 21</td>
<td>2 &gt;= 4</td>
</tr>
<tr>
<td>==</td>
<td>left side equal to right side</td>
<td>6 == 6</td>
<td>-3 == -7</td>
</tr>
<tr>
<td>!=</td>
<td>left side not equal to right side</td>
<td>10 != 9</td>
<td>4 != 4</td>
</tr>
</tbody>
</table>

**TABLE 6-1** Comparative operators
This tells you that you can use comparative operators to compare variables to numbers. You can also compare the value of one variable to the value of another:

```c
int count = 3;
int end_value = 5;

while (count < end_value)
```

In a moment we'll give an example of each comparative operator. In the examples, we'll use the Get_User_Input() function that we first mentioned in Chapter 4. Like Make_A_Window(), Get_User_Input() is one of our own functions. Recall from Chapter 4 that the Get_User_Input() function returns an integer value. In the following example, variable score is assigned a value by calling the function:

```c
int score;
score = Get_User_Input();
```

When we introduced the Get_User_Input() function we didn't show you the code for it, we just asked you to trust us when we told you that it exists elsewhere in our example listings. DrawString() is another function we've made use of in our example programs. Besides having different purposes, these two functions differ in type. Do you recall how they differ? If not, read the following Note.

**NOTE**

Apple supplies a host of functions that you can use in your C language programs. They are built into your Macintosh, and exist in what is figuratively called the Toolbox. You do not (and cannot) see the code for these functions. You simply invoke them from your program and trust that they work. The DrawString() function is one such Toolbox function.

Apple could not anticipate your every programming need. The Toolbox functions do quite a bit, but you'll eventually want to write functions of your own. You'll learn how in Chapter 8. We've written a couple of our own—Make_A_Window() and Get_User_Input(); we just haven't included the code in our listings. In a real program, you would be able to look over the entire code listing and see the actual code for these two routines.
The function Get_User_**Input**() gets a number the user types into an edit box of a dialog and returns this integer to the line of code that called Get_User_**Input**(). Again, we aren't ready to expose you to all of the code that makes up this function. But knowing that there is a function that accepts typed input from the user will make the examples seem a little more useful.

**The Comparative Operators**

Now, on to the examples. Though comparative operators have many uses, we show them off in while loops, because you are familiar with this loop type.

The less than operator (<) compares the left side of an expression to the right side. If the value on the left side is less than the value on the right side, the expression evaluates to true. Otherwise, the expression evaluates to false. Here's an example:

```c
int user_num;
int total = 0;
int entries = 1;

while (entries < 11)
{
    user_num = Get_User_**Input**();
    total = total + user_num;
    ++entries;
}
```

This example allows the user to enter 10 numbers, which are added within the loop body.

The routine first declares and initializes three int variables. It then checks to see if the variable entries is less than 11. Because we initialized entries to 0, the expression evaluates to true and the body of the loop will execute.

The loop body calls our Get_User_**Input**() function to retrieve a number the user types in, saving it as user_num. It then adds the entered number to total and increments entries. The while comparative expression is again checked for truth. When does the loop stop executing? Only when the comparison is false—when the value of entries is 11.

The less than or equal to operator (<=) checks to see if the left side of an expression is less than or equal to the right side. The while statement
used in the above example could be rewritten using the less than or equal operator:

    while (entries <= 10)

    The greater than operator (>) causes an expression to evaluate to true if the left side of an expression is greater than the right side:

    while (entries > 0)

    The greater than or equal to operator (>=) checks to see if the left side of an expression is greater than or equal to the right side:

    while (entries >= 1)

**The Equality and Inequality Operators**

The equality operator (==) is another comparative operator. It compares the left side of an expression to the right side. If the two sides are equal, the expression evaluates to true:

    int count = 3;
    int total = 20;

    while (count == 3)
    {
        count++;
        total = total + 10;
    }

    Does count equal 3? That is the comparison made here. If count does equal 3, the body of the loop will execute. The first time we encounter this loop, count does have a value of 3. The comparison expression is true, and the body of the loop executes. The loop body will run only once.

    Why the double equal sign? Keep in mind that the loop statement is a comparison expression, not an assignment. Let's look at two of the statements in the above example:

    int count = 3;

    This is a declaration and initialization of the variable count. There is only one equal sign involved, because count is being assigned the value of 3.
Remember, this statement reads “the value 3 is assigned to the variable count.”

\[
\text{while (count == 3)}
\]

This is a comparison statement, not an assignment statement. The variable count is not being given the value 3; it is being examined to see if it is already 3. You could read this statement as “while variable count has a value of 3.” What would happen if you inadvertently wrote the while loop as:

\[
\text{while (count = 3)} /* wrong! */
\]

You would not end up with the desired result. Because there is only one equal sign, the program would assign to count the value of 3. And because count can be assigned the value 3 any time, it is a valid, or true, expression. The test would never fail, and the loop would execute repeatedly—certainly not the intention.

**IMPORTANT**

Whenever a program gives unexpected results, check your use of comparative operators.
Examine closely your choice of operators and look for incorrect use of operators, such as = when you mean to use ==.

The inequality operator (!=) compares the left side of an expression to the right side. If the two sides are not equal, the expression evaluates to true. Here’s an example:

```c
int user_num;
int total = 0;
while (user_num != 0)
{
    user_num = Get_User_Input();
    total = total + user_num;
}
```

This example accepts a number entered by the user. It checks to see if the number is not equal to 0. If it is not equal to 0, the program adds the number to variable total. Many loops end when a counter variable reaches a certain value; this is an example of a loop that does not use a counter vari-
able. When the user enters a 0, the test will evaluate to false, and the loop will not execute.

**Comparative Operators—Not Just for Integers**

All of the examples up to this point have used integers in their comparisons. You can also use floating-points in comparisons:

```c
float tax_rate = 5.0;

while (tax_rate < 10.0)
{
    /* body of loop */
}
```

This example uses a floating point variable. When using floating-points in comparison, use only the `<` and `>` operators. Why? Computers do not always store numbers in an exact manner. Comparisons that use the equal sign can thus cause trouble:

```c
float tax_rate;

tax_rate = 9.0 * 1/3;

while (tax_rate == 3.0) /* legal, but DANGEROUS */
{
    /* body of loop */
}
```

If the computer stores the value of `tax_rate` as 2.9999999, as it might, this while loop will not perform as intended, because 2.9999999 is not equal to 3.0.

Comparative operators can also be used with characters:

```c
char quit = 'N';

while (quit != 'Y')
{
    /* body of loop */
}
```

In the above example, the loop statement checks to see if the char variable `quit` has a value of ‘N.’ If it does, the loop body is executed. This
approach is useful for determining if a user wants to quit a program or run it again. Within the body of the loop, you could allow the user to perform an action, such as input a number or draw a shape. You would then ask the user to type in the letter \textit{Y} if he or she was done.

**Assignment Operators**

In Chapter 5 you were introduced to several operators. The most common is the assignment operator (=). The C language offers several variations of the assignment operator: +=, -=, *\textendash\textendash, and /=. All are used to update the variable on the left of the assignment statement. Here's an example:

```c
int ducks = 10;

ducks += 5; /* same as: ducks = ducks + 5; */
```

When the assignment is complete, ducks will have a value of 15. This notation is simply a shorthand way of writing an assignment statement like the following that updates one variable:

```c
ducks = ducks + 5;
```

Here are examples of the four variations:

```c
pets += 6; /* same as: pets = pets + 6 */
radius -= 2.3; /* same as: radius = radius - 2.3 */
age *= 3; /* same as: age = age * 3 */
small /= 100; /* same as: small = small / 100 */
```

These assignment variations are not required—they are simply a more compact way of writing code. Whether you use them or not is up to you. Be aware that they exist. Many programmers, ourselves included, use them. So you'll want to be familiar with these operators when you look at the code of others.

**The while Loop**

By this point you should have a good idea of how the while loop works. We formalize our explanation here. All while loops have the same general form:
while (expression)
statement

Let's examine both parts of the while loop—the while statement itself and the statement that lies below it.

The while Expression

while (expression)

The first line of a while loop is a conditional test. If the expression that lies between the parentheses evaluates to true, the statement below executes. If the expression evaluates to false, the statement below is skipped.

The while Body

statement

The statement part of the while loop lies below while (expression) and can be a single statement ending in a semicolon:

while (x < 20)
x++; /* a single statement body */

Or it can be a compound statement enclosed in braces:

while (x < 20)
{
x++; /* a compound */
    loss = loss - 3; /* statement body */
}

Pre-execution Check

The while statement checks the conditional expression at the top of the loop—before the loop body. Checking at the start of the loop allows for the possibility that the loop body will not execute even once, as in this example:

int x = 5;

while (x > 10) /* check before the loop body */
{
What would be the point of writing a loop that never executes? Recall that a program does not usually execute from start to finish without any variation. Programs branch, as we've mentioned before and as we'll see in detail in the next chapter. Depending on the path taken, a variable involved in a while conditional test may have different values.

Figure 6–2 illustrates the idea of a loop being skipped entirely. Variable user_num is entered by the user. In the first case, the user enters a 0. The while test evaluates to false, and the loop body is skipped. In the second case, the user enters a value of 1. The while test evaluates to true, so the loop body is executed. Within the loop body, a function is called to draw a shape, and variable user_num is decremented. When the while test is performed again, user_num is 0, the test fails, and the loop body is skipped.

Figure 6–3 shows the program flow when a program encounters a while loop. Note that the conditional check, the expression, is performed before the loop body is executed.

```c
/* body of loop */
}

User types in 0, user_num = 0
Loop body is skipped.

User types in 1, user_num = 1
Loop body is executed once.

DrawString("\pHow many more shapes?");
user_num = Get_User_Input();
while (user_num > 0)
{
    Draw_A Shape();
    user_num = user_num - 1;
}
```

FIGURE 6–2  A while loop may or may not be executed.
The for Loop

The for loop performs three actions in one line: it initializes a counter, it compares the counter to an end value, and it increments the counter. Here's an example of a for loop:

```c
int count;

for (count = 1; count < 4; count++)
    DrawString("\pAll for one and one for all.");
```

The three expressions in a for loop are separated by semicolons. Let's look at the three parts of the for loop shown above:

```c
count = 1;
```
We could have picked any name for our counter variable; we opted for the very descriptive name of count. The first expression is the initialization of the counter variable. Like any initialization, the variable is assigned an initial value. Note the use of the single equal sign—this is an assignment, not a comparison. The counter is being given a value. This part of the for loop executes only once:

\[ \text{count} < 4; \]

The second expression is the comparison test. If this expression evaluates to true, the program performs the body of the for loop. The for loop examines the expression before each pass through the loop. At the first examination, count has a value of 1, so the value of this expression is true:

\[ \text{count}++ \]

The third expression increments the variable used as the counter. This prevents the loop from repeating infinitely. This part of the for loop is performed after each pass.

Figure 6–4 summarizes the three distinct parts that make up every for loop statement.
Pre-execution Check

The for statement, like the while statement, checks the conditional expression at the top of the loop. And, like the while statement, a for loop can be written such that it does not execute at all:

```c
int x;
int start = 10;

for (x = start; x < 5; x++)
{
    /* body of loop */
}
```

Figure 6-5 shows the program flow of a for loop. The test of the expression occurs before the loop body ever executes.
The do-while Loop

The third type of loop used in the C language is the do-while loop. In this type of loop, the test condition appears at the end of the loop body:

```c
int count = 0;

do {
    ++count;
    DrawString("Third and final loop type.");
} while (count < 3);
```

Optionally, the while expression part of the do-while loop can change the loop counter value. The following gives the same results as the example shown above:

```c
int count = 0;

do {
    DrawString("Third and final loop type.");
} while (++count < 3);
```

Because the test condition appears after the loop body, the do-while loop guarantees that any do-while loop will execute at least once.

Post-execution Check

We say that the do-while loop is a post-execution check, which means that the loop performs the conditional test at the end of the loop body—after the loop body executes. Both the for loop and the while loop had their checks at the start of the loop—pre-execution. The distinction is this:

- **Pre-execution**  The loop body may or may not be skipped.
- **Post-execution**  The loop is always executed at least once.

Figure 6–6 shows the program flow of a do-while loop. Notice that the body of the loop executes before the loop test expression is evaluated.
When "do" You Use "while," and What "for"?

You are now familiar with the three types of C loops—while, for, and do-while. When faced with a program that requires a loop, how do you choose
which loop to use? There are no hard and fast rules, only a few general
guidelines, which we will now pass on to you.

Most programmers prefer one of the two pre-execution loops, while or
for, to the post-execution do-while loop. The primary reason is caution
(and it's hard to be too cautious when programming). It is best to examine
your condition beforehand, then proceed with the loop.

Between the while and the for loop, the choice is usually based on per­
sonal preference. As the following examples show, both loops can be written
to perform in exactly the same manner:

    /* loop 3 times with a for loop */
    for (count = 0; count < 3; count++)
        DrawString("\pLooping...");

    /* loop 3 times with a while loop */
    count = 0;
    while (count < 3)
    {
        DrawString("\pLooping...");
        count++;
    }

As mentioned, we use the do-while loop only occasionally. You should
use it when you are sure that a loop will be executed at least once. An exam­
ple would be when you are sure that you want the user to enter a value.
Using the Get_User_Input() routine that saves the user's typed response in
a variable called user_num, you could do the following:

    int user_num;

    do
    {
        DrawString("\pEnter your age.");
        DrawString("\pIt must be a number less than 100:");
        user_num = Get_User_Input();
    } while (user_num > 100);

Because the loop test is performed after the first pass through the
body, this loop is guaranteed to execute at least once, so you can be certain
that the program will prompt the user to type in his or her age. If the num-
ber entered is greater than 100, the loop is repeated and the user is forced to
re-enter an age. The loop will repeat until the user types in a number less
than or equal to 100.

Chapter Summary

Computers are good at performing repetitious tasks. The C loop statements
provide a way for you to program the computer to execute portions of code
over and over again.

A loop has a loop statement and a body. The body appears below the
loop statement and is enclosed in braces if it is composed of more than one
line of code. The loop statement evaluates a condition to see if it is true or
false. If the condition is true, the statements below the loop, the body, are
executed. If the condition is false, the statements are skipped.

The test of the loop statement’s condition is performed using compar-
ative operators. The comparative operators are less than (<), less than or
equal to (<=), greater than (>), greater than or equal to (>=), equality (==),
and inequality (!=).

The three types of C loops are the while loop, the for loop, and the do-
while loop.
In the previous chapter we said that a computer's ability to perform repetitious tasks was one of its most valuable features. By learning how to use loops effectively, you gain the ability to add complexity to your programs. By setting the terms of the loop—the number of times the loop will be performed—you control the program flow. Another way to control program flow, add complexity, and add decision-making ability to your programs is to use branching statements.

You are already familiar with one type of branching—the if statement. In this chapter you'll learn more about the if statement and other C language methods of branching.

The if Statement

We introduced the if statement way back in the first chapter of this book. Because you have some familiarity with it, we'll use it to talk about branching statements in general.
About Branching Statements

Listing 7–1 gives a very simple example of a branching statement—an example that was first introduced in Chapter 1.

LISTING 7–1 Branching with an if statement

```c
main()
{
    int sharks = 1;

    Make_A_Window();

    if ( sharks > 0 )
        DrawString("\pSwim fast!");
}
```

If you ran the code in Listing 7–1, you'd see a window with one line of text, as shown in Figure 7–1.

What would happen if you changed the declaration of sharks to the following:

```c
int sharks = 0;
```

**FIGURE 7-1** Result of running Listing 7–1, Shark Attack
Chapter 7  Decision Making: Branching

We reran the program in Listing 7-1, this time with sharks set to 0. The result appears in Figure 7-2.

As Figure 7-2 shows, the result is an empty window. The line of text was not written, because the result of the if statement was not true. The value of sharks was not greater than 0, it was 0. Like looping statements, branching statements have a test that a program evaluates for truth. You use one of the comparative operators covered in Chapter 6 to form a test. If the test passes—that is, if the test evaluates to true—then the program carries out the body of the branching statement. If the test evaluates to false, the program skips the body. That's what happened in the second running of the shark program.

While the empty window resulting from the second running of the shark program may not be all that exciting, it gives you good cause to celebrate. Branching gives you the power to control program flow and to create code that offers decision making.

The if Statement—Formalized

By now you've had a lot of exposure to the if statement, and you should know how it works. Now we'll formalize our explanation.

All if statements have the same general form:

if (expression)
    statement

The first line of the if statement is a test. The program evaluates the expression between the parentheses to see if the statement below the test
should execute. If the expression evaluates to true, the statement will execute. Should the expression evaluate to false, the program will skip the statement.

The second part of the general form of the if statement is either a single statement or a group of statements. If it is one statement, it ends in a semicolon:

```c
if (little_num > 5)
    little_num = 5;
```

If a group of statements lies below the if statement, you place the statements between braces:

```c
if (radius < 10.0)
{
    DrawString("\pRadius is less than 10.");
    circumference = 2.0 * 3.14 * radius;
    area = 3.14 * radius * radius;
}
```

Figure 7–3 shows the program flow when the program encounters an if statement.

**The if-else Statement: Adding Power**

The if statement either executes a body of statements or it doesn’t, depending on the results of the evaluated test expression. There will be times when you will want your program to execute a second body of statements if the test fails. At such times, you use an expanded form of the if statement, the if-else statement:

```c
if (expression)
    statement1
else
    statement2
```
Imagine that you want to print one message if a variable is greater than 100 and a different message if the same variable is less than or equal to 100. You might be tempted to write something like:

```c
if (score > 100) /* This is */
    DrawString("\pGreater than 100."); /* NOT what */
DrawString("\pNot greater than 100."); /* we want! */
```

But this will not give you the desired results. Keep in mind that the line of code following the body of an if statement always executes, regardless of whether the body of the if statement was executed or not. So the above example is not correct. There are two possible results, only one of which is correct. We show both results in Figure 7-4.

It should be obvious from Figure 7-4 that what will occur and what you intended to occur are two very different things. The C language, of course, provides a solution to this dilemma. When you want to be able to handle two separate cases, you can supplement the if statement with an else section. Let’s use an if-else to correct our example:
The if-else statement greatly improves the usability of the if statement. But what if you have more than two options to handle? C provides two methods for dealing with situations like this. One solution is the else-if; the other is the switch statement. Here we'll continue with our discussion of the if statement and its variations. Later in this chapter we'll discuss the switch statement.

When a problem has the possibility of more than two outcomes, you can expand on the if-else statement, as shown on page 114.
score = 200;
if ( score > 100 )
   DrawString("\pGreater than 100.");
else
   DrawString("\pNot greater than 100.");

score = 50;
if ( score > 100 )
   DrawString("\pGreater than 100.");
else
   DrawString("\pNot greater than 100.");

FIGURE 7-5  If-else: one body is always executed.

FIGURE 7-6  The flow of the if-else statement
if (expression1)
    statement1
else if (expression2)
    statement2
else
    statement3

The following fragment prints a message that depends on the number of pennies the user has:

if (pennies >= 100)
      DrawString("\pYou've got at least a buck.");
else if (pennies >= 25)
      DrawString("\pYou've got at least a quarter.");
else
      DrawString("\pDon't spend it all in one place!");

If pennies is greater than or equal to 100, the first expression will evaluate to true and the first DrawString() will execute. Because any number of pennies greater than or equal to 100 is also greater than 25, can you expect the second DrawString() to execute? The answer is no. All forms of the if statement execute one and only one of the bodies—the body below the first expression that evaluates to true. Figure 7–7 demonstrates that for the case of pennies greater than or equal to 100, only the body of the first true expression will execute.

---

**FIGURE 7–7** If statements: only one body is ever executed.
Figure 7–8 shows the program flow for an else-if statement.

We conclude our discussion of the else-if statement with a few miscellaneous topics that might answer any questions you have about the use of else-if.

So far, the same variable—pennies, for example—has been used in both the if and the else-if test expressions:

```c
if (pennies >= 100)
    DrawString("\pYou've got at least a buck.");
else if (pennies >= 25)
    DrawString("\pYou've got at least a quarter.");
else
    DrawString("\pDon't spend it all in one place!");
```
C permits you to use different, even unrelated variables, in tests. Below is an example using farm_animals in one expression and num_cars in another:

```c
if (farm_animals > 100)
    DrawString("\pBecome a vet!");
else if (num_cars > 10)
    DrawString("\pBecome a mechanic!");
```

You can also cascade else-ifs to provide for as many options as you need:

```c
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
else if (expression4)
    statement4
else
    statement5
```

Keep in mind that as with all if statements, one and only one of the bodies executes. C allows you to use any number of else-ifs. When faced with a problem that has several possible outcomes, however, you might find the C switch statement a better solution.

## The switch Statement

The if and else-if statements are two very useful C tools for branching. You'll find that you use them over and over again. There will be situations, however, in which you will need a programming device that does a better job of handling multiple options. In these situations the if statement can be a little awkward. The C language solution is the switch statement. Here's an example:

```c
int job_offers;

job_offers = Get_User_Input();
```
switch (job_offers)
{
    case 0:
        DrawString("\pZero? Mail more resumes!");
        break;
    case 1:
        DrawString("\pOne? Take it!");
        break;
    case 2:
        DrawString("\pTwo? Let them bid for you!");
        break;
    default:
        DrawString("\pName your price!");
        break;
}

The first thing the switch statement does is evaluate the expression that lies between the parentheses following the word switch. In the above example, the program evaluates job_offers. Following the switch line is the body of the switch statement, enclosed in braces. After evaluating the expression, the program checks the list of cases, starting with the first one.

A value follows the word case. In the example, the values are the numbers 0, 1, and 2. When the program reaches a case value that matches the evaluated expression (the value of job_offers), the code below that case executes. If the evaluation doesn't match any of the labels, the code under the default line executes. The default line is optional. If the evaluation doesn't match any case label and there is no default line, the program jumps to the next statement following the switch body.

You've certainly noticed the break keyword; it appears four times in the example. When the program reaches a break it "breaks out" of the switch—it jumps to the end of the switch body. You end the code under each label with a break. If you didn't do this, the program would continue executing each line of code, right up to the end of the switch body. Once the matching label is found, you want to execute only the code that lies below that label. Figure 7–9 illustrates this.

There will be times when you want more than one value to execute the same code. The switch statement allows for this. Here's an example that has two values, a 7 and an 11, both of which execute the same code:
switch (job_offers) {
    case 0:
        DrawString("\pZero? Mail more resumes!");
        break;
    case 1:
        DrawString("\pOne? Take it!");
        break;
    case 2:
        DrawString("\pTwo? Let them bid for you!");
        break;
    default:
        DrawString("\pName your price!");
        break;
}

FIGURE 7-9 The flow of the switch statement

switch (dice_count) {
    case 7:
    case 11:
        DrawString("\pYou're a winner!");
        break;
    default:
        DrawString("\pSorry, try Black Jack...");
        break;
}

This works because there is no break after the case 7. If dice_count has a value of 7, the program will go to case 7. Because there is no break, the program will proceed to the next line of code, the case 11.

We've shown examples with just a single line of code and a break under each case. You are free to have more lines of code following any case.
math_char = 'a';
switch ( math_char )
{
  case 'a':
  case 'A':
    DrawString("\pYou chose Addition.");
    Do_Addition();
    Print_Results();
    DrawString("\pYour numbers were added.");
    break;
  case 's':
  case 'S':
    DrawString("\pYou chose Subtraction.");
    Do_Subtraction();
    Print_Results();
    DrawString("\pYour numbers were subtracted.");
    break;
}

FIGURE 7-10  A switch case value can have more than one line of code.

Logical Operators

Both looping statements and branching statements evaluate test expressions to determine if the body of the statement should execute. C allows you to make these expressions more complex by using logical operators. Logical operators allow the use of more than one expression in one test. One such operator is the and operator (&&). Here is an example:

if (score >= 0 && score <= 100)
  DrawString("\pValid percentage entered.");
else
  DrawString("\pInvalid percentage entered.");
The if statement in the above example reads, “if score is greater than or equal to 0 and score is less than or equal to 100....” Here we assume that a percentage score must fall within the range of 0 to 100, inclusive. Now, with the use of the && logical operator, you can test score to verify that it meets both of these conditions.

In order for an expression that contains an && operator to evaluate to true, both conditions must be met. Meeting one or the other condition is not enough. In the above example, score could have a value of 200, and it would meet the first condition—greater than or equal to 0. But it would not meet the second condition—less than or equal to 100. So the entire expression would evaluate to false, and the else DrawString() would execute.

Another logical operator is the or operator (||). When evaluating an expression that includes an || operator, only one of the conditions must be true to have the entire expression evaluate to true. Here’s an example:

```c
if (snakes > 10 || lizards > 10)
    DrawString("\p Too many reptiles!");
```

The if statement here would read, “if snakes is greater than 10 or lizards is greater than 10....” Remember, with the || statement, the entire expression is considered true if either of the two conditions is true. The following conditions would all cause the if statement to evaluate to true:

- snakes is 20 and lizards is 3
- snakes is 0 and lizards is 13
- snakes is 11 and lizards is 11

The third logical operator C offers is the not operator (!). We used this operator in conjunction with an equal sign in the last chapter and called it a comparative operator, as in:

```c
if (char != 'q')
    DrawString("\p Continuing, not quitting...");
```

You can also use the ! operator alone with the effect of asking, “if what follows is not correct, then the result is true.” Here is an example:

```c
int players = 5;
if (!(players > 10))
    DrawString("\p The value is not greater than 10");
```
The above expression first says 5 is greater than 10. By preceding that expression with the ! operator, you are asking if 5 is not greater than 10. The following three expressions would evaluate to true:

```
!(4 > 100)       /* 4 is not greater than 100 */
!(FALSE)         /* FALSE is not TRUE*/
!(5 == 0)        /* 5 is not equal to 0 */
```

Up to this point we've shown logical operators combined with two expressions. You can use more than one logical operator in one statement. Programs often end when the user types the letter q for quit. But if you wanted to allow the user to type a q for quit or a d for done, you could check for the following:

```
if (char == 'q' || char == 'Q' || char == 'D' || char == 'd')
{
  DrawString("\pYou're done, I'm quitting...");
    ....
}
```

Then, any one of the four characters would cause the DrawString() to execute along with, presumably, some other code that would cause the program to end.

**switch or if?**

A few pages back we gave you a switch example for dice. It's repeated below:

```
switch (dice_count)
{
  case 7:
  case 11:
    DrawString("\pYou're a winner!");
    break;
  default:
    DrawString("\pSorry, try Black Jack...");
    break;
}
```

The above dice example could just as well have been carried out with an if-else statement:
if (dice_count == 7 || dice_count == 11)
    DrawString("\pYou're a winner!");
else
    DrawString("\pSorry, try Black Jack...");

If you can get the same result with either a switch or an if-else statement, how do you go about choosing between the two? Let's look at an example that should make the decision easier. If you want to write some C code to display a message about a letter the user types in, you could do so as follows:

char c;

c = Get_User_Input();

if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
    DrawString("\pThat's a vowel.");
else if (c == 'q' || c == 'x' || c == 'z')
    DrawString("\pThat letter is seldom used.");
else
    DrawString("\pThat's a consonant.");

You can achieve the same results using a switch, and most people would consider it easier on the eyes:

switch (c)
{
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        DrawString("\pThat's a vowel.");
        break;
    case 'q':
    case 'x':
    case 'z':
        DrawString("\pThat letter is seldom used.");
        break;
    default
        DrawString("\pThat's a consonant.");
break;
}

As a programmer, you make the choice. Here we offer two situations in which we recommend that you use a switch over an if-else statement:

- When you have several values that yield the same result, such as our example in which any one of five vowels caused the same DrawString() to be called.
- When you have numerous options. A dozen case statements are easier to read than a dozen cascaded else-if statements.

**Chapter Summary**

Using looping statements is one way to take advantage of your Macintosh's computing power. A second way to take control of your computer is to add decision-making to your programs. Branching statements do just that.

The if statement evaluates an expression. If the result is true, the statements in the body of the if statement are executed. To add power to the if statement, use the if-else form. Depending on the outcome of the if evaluation, the program flow will perform one of two bodies of statements.

Logical operators like the and operator (&&) allow you to use more than one expression in an if statement's test. A second logical operator is the or operator (||).

When you want your program code to branch in one of several directions based on a test, use the switch statement.
Chapter 8

All About Functions

After Chapter 4 you should have a sound understanding of what functions are and how your program calls them. In this chapter we’ll elaborate on the use of functions. You’ll learn about arguments and parameters—the programming tools used to transfer data from one function to another. You’ll see that functions can return the favor by sending data back to the source that originally called the function. In short, this chapter will give you the information you need to make functions the powerful programming tool they are intended to be.

Quick Review

In Chapter 4 you learned that functions are the basis of C programming. Functions are a way of grouping code into useful, reusable packages. Figures 8–1 and 8–2 are from Chapter 4. We repeat them here as a means of recapping the basic properties of functions.
Think THINK C

FIGURE 8-1 Components of a function

```
main()
{
    int total;
    int a = 5;
    int b = 8;
    total = a + b;
}
```

FIGURE 8-2 The difference between calling and defining a function

```
main()
{
    Draw_Hello_There();
}
```

```
Draw_Hello_There()
{
    DrawString("\nHello there!");
}
```
Function Arguments and Parameters

Listing 8–1 is a very simple program that contains the main() function and one other function, Cheerleader(). See if you can determine what the result of running this program would be.

LISTING 8–1 Calling a function using a parameter and an argument

```c
int count;

for (count = 1; count <= num_cheers; count++)
    DrawString("\pGO! ");

Figure 8–3 shows the screen output after running the program in Listing 8–1.

FIGURE 8–3 Result of running Listing 8–1
```
Although the program shown in Listing 8–1 may appear trivial, it is, in fact, a great step forward for you. For the first time, we have defined a function such that it is capable of producing different results each time it is used! The first call to Cheerleader() resulted in GO! being written twice. The second call wrote GO! four times. With one small addition to a function, you’ve gained almost infinite flexibility. Let’s take a look at this addition.

Arguments

Listing 8–1 contains a function definition that differs in one way from the functions you’ve seen in the past—it has an argument between the parentheses that follow the function name. An argument is a declaration of a variable that can be used in the function. It is a local variable that can be used only by the function itself. Figure 8–4 emphasizes this idea.

Both num_cheers and count are local variables. Both exist only within the function itself—they are valid only within the function. You declare both in a similar manner: you list the variable type first, then the variable name. As local variables, how do num_cheers and count differ? They differ in the way that each gets its initial value. A local variable declared within a function, such as count, receives its initial value within the function. An argument local variable receives its initial value from outside of the function. In Cheerleader(), the local variable count is given its initial value of 1 in the for loop. A local variable can also be initialized when first declared:

```
int count;
for (count = 1; count <= num_cheers; count++)
    DrawString("\pGO! ");
```

**FIGURE 8-4** An argument in a function
We said that an argument local variable obtains its initial value from outside of the function. Next, you'll see how that happens.

**Parameters**

When you make a call to a function that has an argument, the line of code that calls it contains a *parameter*. Figure 8–5 illustrates this.

A parameter is a value that the argument takes on. Thus, an argument variable receives its value when a function is called. That's how it can take on different values—its value is dependent on the call to it. Figure 8–6 shows the program flow when there is a call to a function. It also illustrates that the argument *num_cheers* gets a value when control goes to the Cheerleader() function and that *num_cheers* takes on the value of the parameter used in the call.

```
main()
{
    Make_A_Window;
    Cheerleader(2);
    DrawString("You're in!");
    Cheerleader(4);
}

Cheerleader( int num_cheers )
{
    int count = 1; /* initialize count to one */
    ....
    ....
}
```

**FIGURE 8–5**  A parameter and an argument
Parameters are often referred to as *passed parameters*. Our discussion and Figure 8–6 should make this term clear. In a sense, the parameter is being passed on to the function, where the argument takes on its value.

Up to this point you’ve seen parameters that are constants, like the numbers 2 and 4. Parameters can also be variables. The following two calls to Cheerleader() would both pass the value 10:

```c
int number = 10;

Cheerleader(10);
Cheerleader(number);
```

Notice there is no association between the name of a variable used as a parameter in the function call and the name of the called function’s argument. Function Cheerleader() has an argument named `num_cheers`. That doesn’t mean that when you call the function the passed parameter must also have the name `num_cheers`. In Figure 8–7 a variable called `number` is the passed parameter. The argument, `num_cheers`, takes on its value, 10. The name of the passed parameter is not important. What is important is that the passed parameter be of the same type; it must be an int, because `num_cheers` is an int.

```
Before call:
num_cheers = ??

Once called:
num_cheers = 10
```

**Figure 8–6** An argument gets its value from a parameter.
main()
{
    int number = 10;
    Cheerleader(number);
}

Cheerleader(int num_cheers)
{
    int count;
    for (count = 1; count <= num_cheers; count++)
        DrawString("\pGO");
}

FIGURE 8-7 Passing a variable as a parameter

Keep in mind that an argument is a local variable. That means that when the function ends, the variable disappears. The value of the argument disappears and does not affect the value of the passed parameter. C does, however, offer a way in which a function can change the value of a passed parameter so that when the function ends, the passed parameter retains changes made to it by the function. We'll cover this concept in the next chapter.

A function can have more than one argument. If it does, they are separated by commas. And when called, the number of parameters must match the number of arguments. Here's an example:

main()
{
    int num1 = 5;
    int num2 = 7;

    Check_Total(num1, num2);
}

Check_Total(int a, int b)
{
    int total;
total = a + b;

if (total > 10)
    DrawString("You're over the limit of 10.");
}

Function Return Values

In the preceding pages you learned that parameters are a way of passing information from the calling function to the function being called. Is it safe to assume that C provides a way to pass information back in the other direction? But of course. We briefly mentioned this idea in Chapter 4 when we talked about our own Get_User_Input() function. We showed the result of a function returning a value, but we never showed you how the function managed to do this task. The C keyword return allows a function to pass back, or return, one value to the calling function. Listing 8–2 is a short program that demonstrates the use of the return keyword:

LISTING 8–2 Using return in a function

main()
{
    int sum;

    sum = Add_Two_Integers(5, 7);
}

int Add_Two_Integers(int a, int b)
{
    int total;

    total = a + b;

    return total;
}

The function definition for Add_Two_Integers() in Listing 8–2 has an element not found in any of the previous functions—a function return type. When a function returns a value, you must include the type of the
FIGURE 8-8 Function definition and return value

returned value in the first line of the function definition. Figure 8–8 illustrates this.

The preceding example uses a variable as the return value. The C language also allows you to return the value of an expression. You could rewrite the Add_Two_Integers() function as follows:

```c
int Add_Two_Integers(int a, int b)
{
    int total;
    total = a + b;
    return total;
}
```

We'll end our discussion on function return values with one last example function. This one should find practical applications in C programs you write. The Determine_Minimum() function does just what its name implies. Given two integers, it determines and returns the minimum value of the two:

```c
int Determine_Minimum(int a, int b)
{
    int min_value;

    if ( a < b )
```
min_value = a;
else
    min_value = b;
return min_value;

With very little effort, you could change Determine_Minimum() into another useful function—Determine_Maximum(). We leave it to you to make the necessary changes.

Function Prototypes

When a C compiler compiles your source code file, it starts at the top and works its way down. As it does so, it examines your code and translates it into object code—symbols more readily understood by the computer. As it does this, it may come across a call to a function before it reaches the function definition itself. Figure 8–9 shows this.

The compiler compiles line-by-line

main()
{
    int total;
    ....
    total = Add_Numbers();
}

int Add_Numbers()
{
    /* body of function */
}

The compiler encounters the function call ...

... before the function definition.

FIGURE 8–9  A call to a function may occur before the function definition.
The situation shown in Figure 8–9 can present a problem to some compilers. You declare variable total as an integer. When the compiler reaches the line that calls Add_Numbers(), it has yet to see the function definition. It cannot be sure that Add_Numbers() does indeed return an integer value. Some compilers will stop at this point and notify you that an error condition may exist.

From our description of this problem, you can see that you need a way out of this dilemma. And, of course, C provides one—function prototypes. A function prototype is a way of declaring a function before using or defining it. If you put a prototype of a function at the start of your code, the C compiler will encounter the function there before it sees it elsewhere. And if your prototype declares all facets of the function—its return type, its name, and the type of each of its arguments—then the compiler will know exactly what to expect at the first occurrence of that function. Below is the function prototype for the Determine_Minimum() function we used in our last example:

\[
\text{int Determine\_Minimum(int, int);} \quad /* \text{function prototype} */
\]

The prototype tells the compiler what to expect of the function, as shown in Figure 8–10.

When a function has no arguments, the prototype tells us to use the C keyword void, as we've done in Figure 8–11. If a function has no return value, you use the void keyword when declaring the function return type, as shown in the example in Figure 8–12.

To see a more complete example of function prototypes, look at Listing 8–3, which shows two functions called from within main() and their prototypes.
Function Prototype:
no argument

int Do_Something( void )

FIGURE 8–11 A prototype for a function with no arguments

Function Prototype:
no return two arguments:
value an int and a float

void Do_This_Or_That( int, float );

Call To The Function:
Do_This_Or_That( 5, 36.24 );

Function Definition:
Do_This_Or_That( int a, float z )
{    /* body of function */
}

FIGURE 8–12 A prototype for a function with no return value
LISTING 8-3  Function prototypes in use

```c
/* function prototypes */
void Show_Grade(char);
int How_Many_Wrong(int);

main()
{
    int score = 80;
    int num_wrong;

    Show_Grade('B');
    num_wrong = How_Many_Wrong(score);
}

Show_Grade(char grade)
{
    DrawString("Your test score was: ");
    DrawChar(grade);
}

int How_Many_Wrong(int score)
{
    int wrong;

    wrong = (100 - score)/10; /* assume each question */
    /* is worth 10 points*/
    return wrong;
}
```

Some C compilers are smart enough to determine what many functions look like, even if no prototypes are given. But they can’t determine what all functions look like. To help the compiler out and to avoid introducing bugs into your code, always use prototypes. We’ll be giving you more examples of prototypes throughout the remainder of this book. And when we develop complete applications, such as the ShapeMaker program
in Chapter 19 and the MacCertificate program in Chapter 20, we'll give
prototypes for every function we write.

Chapter Summary

Functions are at the heart of C programming. A function groups code into
a reusable package that serves a specific purpose.

Functions communicate with one another via parameters and arguments. A function that calls another function passes information in the
form of parameters. The receiving function—that is, the called function—
receives this information in the form of arguments.

When a function calls another function, it can pass either a variable or
a value as a parameter. Regardless of what it passes, the called function
receives this information as a value—either the value passed or the value of
the variable passed. The called function places this value into a variable
local to the called function—an argument.

A function can return a value to the calling routine by using the C
keyword return. The return keyword can be used to return a constant value,
the value of a variable, or the value of an expression.

A function prototype is used to give the C compiler advance notice of
what to expect of the functions contained in a program. Each prototype
describes the return type of a function, the function name, and the type of
each function argument. Prototypes are listed at the start of the source code.
Chapter 9

Memory and Pointers

When you write a computer program, all the code you write is eventually loaded, or placed, into computer memory. Most of the time, you won't be too concerned about where in memory your particular program code goes. But there will be times when you, as a C programmer, must know. This chapter introduces you to the basic terms and concepts used in discussing memory. You'll learn how the Macintosh stores a value in memory. You'll also learn about memory locations and how the computer keeps track of them.

After learning about memory, you'll be introduced to pointers. You'll learn how pointers help you track down the specific location of a variable in memory. You'll see how you can increase the power of functions by using function arguments that are pointers.

Memory

Perhaps more than any other language, C relies on the use of pointers. One of the main goals of this chapter is to make you feel comfortable working
with pointers. Pointers point to things in memory—so they cannot be discussed separately from memory. Before we delve into pointers, we want to make sure you have a thorough understanding of memory.

**Memory: Bits and Bytes**

Computer memory is not one long ribbon of uninterrupted space. It is small cells grouped together. Each cell is called a bit. At any give time, each bit is capable of taking on one of just two values—a 1 or a 0. Figure 9-1 shows this.

Figure 9–1 shows, arbitrarily, 17 bits of memory. A typical computer has millions of bits of memory, so we’re sure you’ll forgive us for not showing them all. You need only look at a handful of bits to see how memory is organized. One bit holds very little information. As we said, it holds only the number 1 or the number 0. So to organize data in a more manageable way, computers group bits together. A group of eight adjacent bits is called a byte of memory. People like to count and think in groups of ten; unfortunately for us, computers use groups of eight. Figure 9–2 shows a segment of memory grouped into bytes.
When we speak of units of memory, we are most often talking about bytes of memory. Figure 9–3 shows a little more memory than the previous figure. This time the emphasis is on bytes rather than bits.

A Bit of What?

Memory holds numbers. In particular, each bit of memory can hold a 1 or a 0. The idea of numbers is a concept devised by, and for, the minds of people. You may be wondering how an electronic machine like a computer can know what a 1 and a 0 are. Or how a bit of memory can really capture a 1 or a 0. It can’t. People use the numbers 1 and 0 for their own reference—it’s a concept that is easy for humans to grasp. What does a bit of memory really hold? We hinted at the answer when we stated that computers are electronic machines.

A bit of memory can hold an electrical signal—a voltage. If a memory bit has a voltage above a certain level, the computer considers that bit to be a 1, or on. If there is no voltage, or just a very small amount, the computer considers the value of the bit to be a 0, or off. Figure 9–4 illustrates this concept.

We’ve answered our question of what a bit of memory holds. But with every answer comes another question. Knowing that a bit can contain just one of two values, how do we then represent a number such as, say, 7? The answer lies in the sequence of ones and zeros we use.

Storing a Number in Memory

A computer stores numbers in binary form. That means it uses 2 as its base unit rather than 10, the decimal form of number counting you are used to. In the base 10 system, when you get to 9 you follow by rolling over into a
new column—10. In the base 2 system, the roll over occurs right away—from 1 to the next column. Figure 9–5 shows the binary representation of the numbers 0 through 8.

In Figure 9–5, each of the four columns in the binary column represents a power of 2. A power of 2 is the number of times we multiply 2 by itself. Thus the far left column, read “2 to the 3rd power,” represents:

\[ 2 \times 2 \times 2 = 8 \]

Look at integer 6 in Figure 9–5. Its binary representation is 0110. That means the 4 column is on and the 2 column is on. Adding 4 to 2 gives 6.

Figure 9–6 shows how the Macintosh stores the number 1 in a byte of memory. Simply set the rightmost bit on and all others off.
\[ 2^3 = 8 \]

<table>
<thead>
<tr>
<th>Integer</th>
<th>Binary</th>
<th>Addition of the numbers each row represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0+0+0+0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0+0+0+1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0+0+2+0</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0+0+2+1</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0+4+0+0</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>0+4+0+1</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>0+4+2+0</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0+4+2+1</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8+0+0+0</td>
</tr>
</tbody>
</table>

A 1 here is like a checkmark signifying that the 8 column is on.

**FIGURE 9-5** Binary representation of 0 through 8

**FIGURE 9-6** Binary representation of the integer 1
Figure 9–7 shows a byte containing the number 2 and another byte containing the number 3.

If all eight bits of a byte are on, then that byte represents the largest integer a single byte can hold—255. This is shown in Figure 9–8.

Figure 9–8 shows you that the biggest integer a single byte of memory can hold is 255. If you want to store a larger number, you need more memory. The C language defines an int type to be two bytes in size. The two bytes that make up one int variable are always consecutive, or adjacent, bytes—no matter where in memory they end up. Because you now have two bytes to hold a single number, you can hold a much larger number.
The bits of this second byte represent powers of 2 from $2^8$ up to $2^{15}$. A two-byte integer can hold a number that is anywhere in the range of $-32768$ to $+32767$.

### Memory Addresses

Because there are thousands, perhaps millions, of bytes of memory in your computer, it is obvious that some plan is needed to keep track of individual bytes. The system computers use is called *addresses*. An address is a number. Each byte of memory has its own unique address, different from the addresses of all other bytes in memory. Figure 9–9 shows a simplified view of this concept. The portion of memory we chose to show has its first byte at address 2000, its next byte at address 2001, and so on.

Every byte of memory has two attributes—a value and an address. Figure 9–10 shows a few bytes of memory, each of which has an address and a value. For simplicity, we show the byte values as integers; the computer would actually store the values in memory in their binary forms.

### Pointers

The C language makes greater use of *pointers* than many other computer languages do. Pointers can be thought of in two ways:
Every byte has both a value and an address.

- A pointer points, or directs you, to the memory location of a data structure, such as a variable.
- A pointer is an address.

Let's tie these two definitions together. We use the word *point* in the sense of directing. If you ask me for directions to Main Street, I will point to it. If you want to know the location of a variable in memory, you get something to point you to it. Imagine that you declare an int variable named states and initialize it to a value of 50. Earlier we said that int variables occupy two bytes in memory. If a variable occupies more than one byte, we say its address is the address of the first byte. If your declaration happens to put variable states in the two bytes starting at address 2000, memory will look like Figure 9–11.

Figure 9–11 shows the memory location of the states variable. You know the value of states—50—from the declaration. But you can't derive a variable’s address from the declaration alone. Your declaration does not tell you where in memory variable states ends up. You can, however, request that information—in the form of a pointer.

**What Pointers Point To**

Pointers are a type of variable. Their purpose is to point, or direct, you to another variable. Because a memory address tells you where a variable resides in memory, that's what a pointer points to—an address. Like any
variable, you give a pointer a name when declaring it. You also specify the type of the variable you are going to be pointing to, such as int or float. Finally, to let the C compiler know that you are declaring a pointer, you preface the pointer name with an asterisk (*). Here we have a declaration of an int and a pointer to an int:

```c
int states = 50; /* declare and initialize an int */
int *state_ptr; /* declare a pointer to an int*/
```

Now, whenever you use `state_ptr` in your code, you refer to an address. You declared `state_ptr` to be a pointer to an integer, but you didn't tell it which integer to point to. To get your pointer to point to what you want, use the & operator. Preceding a variable name with the & operator gives the address of that variable. Here's how you get `state_ptr` to point to variable `states`:

```c
state_ptr = &states;
```

Instead of a normal numerical value, `state_ptr` now holds the address of int variable `states`. The previous figure showed that the address of variable `states` is 2000. So the contents, or value, of the pointer that points to `states`, `state_ptr`, must be 2000. Figure 9-12 shows how memory now looks.
Figure 9–12 shows that state_ptr occupies four bytes of memory, from addresses 2002 to 2005. On a Macintosh computer, this is the size of a pointer. Addresses can get pretty large, so the memory reserved for a pointer has to be large enough to hold a big number. Figure 9–12 also shows the pointer to states residing in memory right next door to states. Is this always the case? No. A pointer can be just about anywhere in memory. What is important about a pointer is not its location, but its contents; the contents tell you where it’s pointing. The contents of state_ptr is 2000, so state_ptr is pointing to address 2000. And what is at memory address 2000? Variable states. This pairing of state_ptr with states came about through the magic of the & operator when you wrote:

```c
state_ptr = &states; /* make state_ptr point to states */
```

Other than the size of the two variables, how does the value of states, 50, differ from the value of states_ptr, 2000? The first is a number that can stand for anything—in this case, the number of states in the United States. The second is an address and stands only for a memory address in the Macintosh. How does your C compiler know the difference? From your declaration. Remember, when the pointer was declared, the * operator was included in the declaration. This let the compiler know that the contents of this variable represents an address:
int states = 50;        /* declare and initialize an int */
int *state_ptr;         /* declare a pointer to an int*/

With pointers, the * operator is not the multiplication operator. Here, in the context of pointers, you are using the * operator as a unary operator—you are using it with just one variable. When used in multiplication, it behaves as a binary operator—you multiply two variables.

If the United States were to annex 1950 more states, for a total of 2000, then both states and state_ptr would have a value of 2000. The wise C compiler would still know that only one represented an address—the one declared with the * operator, state_ptr.

Dereferencing a Pointer

Once you've assigned a pointer to point to something, it holds the address of that something. If you want to follow the trail from the pointer to that actual something, you dereference the pointer—you get at the contents of the memory that the pointer points to. Dereferencing your state_ptr pointer would lead you to the value 50. Figure 9–13 shows this process.

Let's look at the C code that you use to dereference a pointer. To let the C compiler know you are working with a pointer, the * operator is used when declaring the pointer. When dereferencing, you are once again dealing with pointers. And once again you use the * operator. Below is a code fragment that first assigns state_ptr to point to states, then dereferences state_ptr. This code fragment uses the int variable temp to hold the dereferenced value. When complete, the final value of temp will be 50:

```c
int states = 50;        /* declare and initialize an int*/
int *state_ptr;         /* declare a pointer to an int */
int temp;              /* to be used for dereferencing */

state_ptr = &states;    /* make state_ptr point to states */

temp = *state_ptr;      /* temp = what state_ptr points to */
```

Let's summarize what's been covered to this point. When you declare a pointer, you do so in the following manner:

```c
int *state_ptr;
```
Then whenever you refer to the pointer, you use state_ptr. Whenever you refer to the thing that state_ptr points to, you use *state_ptr.

This may appear to be a round about way of doing things. And in our simplistic example, it may just be. But pointers are a vital part of the C language, as you'll see in the next section of the chapter.

**Changing Parameter Values**

In the previous chapter you saw that C lets you write a function that sends a value back to the function that called it. You saw that the return keyword provides a way to change the value of one variable. In Listing 9–1 the value of variable the_int is changed from 50 to 100.
LISTING 9–1 Changing the value of a variable using return

```c
main()
{
    int the_int;

    the_int = 50;
    the_int = Set_Int_To_100();
}
int Set_Int_To_100()
{
    return 100;
}
```

The return keyword is a very useful and powerful programming tool. It does have a limitation though—it can return only one value. What do you do if you want a function that yields more than one result? In particular, what do you do if you want a function to change one or more of the passed parameters? Listing 9–2 shows what may appear to be the obvious answer.

LISTING 9–2 The parameters won’t be changed here.

```c
main()
{
    int first_int = 1;
    int second_int = 6;

    Change_Us(first_int, second_int);
}

Change_Us(int first, int second)
{
    first = 10;
    second = 20;
}
```

When all is said and done, what will the final values of `first_int` and `second_int` be? Not 10 and 20, as you might have thought. After you run the program, `first_int` will still have its initialized value of 1, and `second_int` will have its initialized value of 6. What function `Change_Us()`
does is change the values of local variables first to 10 and second to 20. What happens to local variables in a function has no bearing on what happens to variables outside of a function.

So just how do you go about changing the values of passed parameters? What do you do if you want parameters first_int and second_int to take on the values that are assigned to their argument counterparts? The answer lies in a device you’ve just covered—pointers.

When you make a function call and pass a variable, you pass only the value of the variable. If an int variable called dogs has a value of 8, using dogs as a passed parameter in a function call results in the value 8, not the variable itself, being passed. Figure 9–14 shows this.

What if you want the value of dogs to be affected by the function to which it’s passed? How do you preserve changes made by a function? Pass the address of a variable, rather than the value of a variable. This is all the information your C compiler needs to change a variable’s value. You know that a pointer is a variable’s address, so you might guess that pointers are somehow involved. To pass an address of a variable, use C’s & operator in a function call. Preface the variable name with the & operator, like this:

Add_One_Animal(&dogs);
That covers the passing end of things. You also have to make the receiving end aware of the fact that an address, not a variable value, is being passed. For that, use the C language's * operator. Here is a new definition of the Add_One_Animal() function:

```c
Add_One_Animal(int *animals)
{
    ++*animals;
}
```

Any changes made to a function's argument will now apply to the passed parameter of the calling routine as well. Figure 9–15 shows the & and * operators being used by function Add_One_Animal() to change the value of dogs.

Has it been a while since we mentioned how powerful the C language is? We'll say it again: C is powerful! Simply by prefacing a few variables with the & and * operators you can change the workings of a function. A function changes from one that has no effect on an outside variable to one that impacts any variable you want to pass to it.

```
main()
{
    int dogs = 8;  /* dogs = 8 */
    int pets;
    Add_One_Animal( &dogs );
    pets = dogs;   /* pets and dogs = 9 */
}

Add_One_Animal(int *animals)  /* animals = 8 */
{
    ++*animals;  /* animals = 9 */
}
```

**FIGURE 9-15** With the & and * operators, a changed argument affects the passed parameter.
Pointers in Action

In C, as in any programming language, swapping the value of one variable for that of another variable is a common task. It is a technique used in several types of sorting functions. Before we give you the listing, we first mention two problems you must overcome.

First, you cannot simply pass two variables to a swap function. You want the swap to "stick." That is, after the swap function, you want the passed variables to have and maintain their newly acquired values. So you know that a swap routine will require pointers.

The second problem is in the swap itself. Your initial thought may be to try something on the order of:

```c
int a = 10;
int b = 20;

a = b;       /* so far so good, but... */
b = a;       /* ...a is now 20, and so is b! */
```

This, alas, is doomed to fail. After the first assignment, both variables have a value of 20, so any further switching is pointless. You can overcome this problem by using a temporary variable, as shown in the integer swapping listing, Listing 9-3.

**Listing 9-3  Swap: a function that uses pointers**

```c
main()
{
    int a = 10;
    int b = 20;

    Swap_Ints(&a, &b);

    DrawString("\pReturned from swap.");
}

Swap_Ints(int *x, int *y)
{
    int temp;
```
temp = *x;
*x = *y;
*y = temp;
}

About the Listing

Let's take a look at Listing 9–3. It first declares and initializes the two values to be swapped, a and b. Figure 9–16 shows the main() function and a segment of memory.

The Swap_Ints() function is defined with two integer pointers as its arguments:

\[
\text{Swap\_Ints}(\text{int } *x, \text{int } *y) \quad / \ast \text{ two pointers } / \ast
\]

So when you call it, you must pass two pointers. What is a pointer? An address. How do you get the address of a variable, any variable? Use the & operator. That's exactly what is done when Swap\_Ints() is called:

\[
\text{Swap\_Ints}(&a, &b); \quad / \ast \text{ passing two addresses } / \ast
\]

\[
\begin{array}{cc}
\begin{array}{cccccccc}
 a & b \\
10 & 20 & ?? & ?? & ?? & ?? & ?? & ?? \\
5010 & 5012 & 5014 & 5016 & 5018 & 5020 & 5022
\end{array}
\end{array}
\]

main()
{
    int a = 10;
    int b = 20;
    Swap\_Ints( &a, &b );
    DrawString("\pReturned from swap.");
}
Think THINK C

Recall that a function's arguments are variable declarations. So the definition of `Swap_Ints()` creates two integer pointers, `x` and `y`, for use in the function. Figure 9-17 shows how memory might now look.

Once in the `Swap_Ints()` function, you first dereference `x`. By definition, `x` is a pointer. To make use of what `x` points to, dereference it:

```c
temp = *x;
/* dereferencing x */
```

With the above assignment, variable `temp` is assigned the contents of what `x` points to. We illustrate this in Figure 9-18.
With the value that \( x \) points to saved in \( \text{temp} \), set the contents of what \( x \) points to equal to the contents of what \( y \) points to:

\[
* x = * y ; 
\]

/* dereferencing both \( x \) and \( y \) */

This statement is the same as writing \( a = b \). Figure 9–19 diagrams this.

Finally, dereference \( y \). Set the contents of what \( y \) points to equal to \( \text{temp} \). The swap is complete. When the function exits and control goes to \( \text{DrawString()} \), \( a \) will be 20 and \( b \) will be 10. Figure 9–20 illustrates the final step.

**FIGURE 9–19** Variable \( a = b \), through pointers

**FIGURE 9–20** The final step of the swap
Passing Two Ways

The swap example shows one way to pass a parameter as an address:

```c
int a = 10;
int b = 20;
Swap_Ints(&a, &b);
```

You could also have created and passed two pointers. This method is a little more involved, but it is also a little more descriptive of what is going on:

```c
int a = 10; /* declare and initialize an int */
int b = 20; /* declare and initialize an int */
int *a_ptr; /* declare a pointer to an int */
int *b_ptr; /* declare a pointer to an int */

a_ptr = &a; /* assign pointer to point to a*/
b_ptr = &b; /* assign pointer to point to b*/

Swap_Ints( a_ptr, b_ptr ); /* pass the two pointers */
```

In future chapters we’ll be exploring a number of the Macintosh Toolbox functions. Many of these functions require pointers as passed parameters. For simplicity we’ll use our first method. We want you to be aware that both methods achieve the same goal and both are acceptable.

Handles

Macintosh memory has one other device you should know about—the handle. A handle is a pointer to a pointer. Handles exist because of something the Macintosh computer does on its own—memory compaction.

Throughout the execution of your program, memory is assigned and given up. Sometimes the amount of memory is very small; sometimes it is a larger block. In any case, the result is the same—memory can become fragmented. That is, there are patches of memory that are dedicated to data structures and, between them, patches that are free.
Occasionally, the Macintosh will shift data around in memory for the sake of efficiency. It will attempt to group several small free areas of memory into one larger area—this is memory compaction. In doing so, the potential exists to leave your program in the dark. Your program, and you, expect variables and other data structures to be in particular memory locations. You may have pointers to certain variables. These pointers contain the addresses of variables. Shifting and relocating memory could play havoc with your program’s ability to keep track of the memory location of variables. That’s where handles come in.

Once created, a handle is immune to this memory shifting. So the computer uses it as a reference point that it can always count on. It keeps track of the memory shifting that takes place during compaction and appropriately updates pointers so they continue to point to the variables they were assigned to point to.

We won’t go into depth on just how memory and handles interact. Memory compaction and pointer updating are done automatically—you don’t have to worry about when these actions occur. For your own satisfaction, we do want you to know of their existence. Macintosh C uses handles in some Toolbox routines. In fact, Macintosh C has a type called handle. We’ll have a little more to say about handles in future chapters.

**Chapter Summary**

Programs you write get placed in memory when they are executed. So it is important that you know something about computer memory.

A bit is the smallest unit of memory. It is capable of holding only one of two values—a 0 or a 1. When eight bits are grouped together, they form a byte. A computer’s memory consists of thousands or millions of bytes.

A computer uses the binary system to represent any number as a sequence of ones and zeros.

Every byte of memory has a unique address. An address is a number that allows the computer to keep track of its memory locations. The value of each variable is held in memory. To coordinate what value is associated with what variable, the computer uses addresses.

You can use pointers to find the address of a particular variable. A pointer is the memory address of a variable. If you have a pointer to a variable, you know the address of that variable. To find the value from a pointer, you dereference the pointer. That is, you find out the contents of
the memory location to which the pointer points. C uses the * and & operators when dealing with pointers.

If you precede a passed parameter with the & operator in a call to a function, changes made to that passed parameter by the called function will remain with that passed parameter after the called function ends.

Handles are a special form of pointer used to prevent other pointers from getting lost in the shuffle when memory is compacted. Memory compaction is an efficiency measure. Fragmented memory is removed to allow larger areas of free memory.
Computer programs often store and manipulate large amounts of data. In fact, that's the computer's claim to fame. It can store and manipulate huge amounts of data far better than any human being can. In the preceding chapters you were introduced to several data types. Although they differ from one another, they all share one property—they are all meant to hold a single value, whether that value is an integer, a floating-point number, or a character. As a programmer, you'll want to be able to store, access, and work with more than one value at a time. The C language gives you two data types for working with multiple pieces of data. The more you program, the more you will come to depend on these two very important data types—the array and the structure.

**Arrays**

An array is a group, or series, of data of the same data type. You declare an array in much the same manner as you declare any other variable. The difference is that you specify *how many* of the variables you want. You do this
by following the variable name with the number of variables in the array, enclosed in brackets:

```c
int temperatures[3]; /* an array of three integers */
```

The above declaration will give you an array named temperatures that contains three integers. The computer places the three integers in three consecutive memory locations. Recall from the previous chapter that an integer occupies two bytes of memory. Assuming the array starts at memory location 3400, the temperatures array will then occupy the six bytes of memory at addresses 3400 through 3405. We illustrate this in Figure 10–1.

We've placed question marks in the array memory locations in Figure 10–1 because we declared an array but we did not initialize it. We'll discuss initialization in a moment.

We said the temperatures array holds three integers—as defined by the number between the brackets following the array name. To access any one of these three elements, or array members, you use brackets with the array element's subscript, or index. Subscript and index mean the same thing—they refer to the position in the array of one element. In C, array subscripts always start with 0. So an array of three elements, such as our temperatures array, will have three elements, called temperatures[0], temperatures[1], and temperatures[2]. Figure 10–2 illustrates this idea.

Below are the declarations of several more arrays.
Array Initialization

When you declare a variable, such as an int, you are given the option of assigning it an initial value:

```c
int animals = 5;    /* initialize animals to 5 */
```

The C language gives you this same option with array elements. To assign a value to each element, list the values between braces. Separate them with commas. Here's an example using the temperatures array:

```c
int temperatures[3] = { 0, 32, 212 };
```

The above initialization will result in the temperatures array elements having the values shown in Figure 10–3.

Assigning Values to Array Elements

In many cases, you won't know the values that go into the array until after your program starts. In such cases, you want to be able to assign values to array elements in a way other than initialization. To do this, treat array elements as variables of the array's type:
```c
int temperatures[3] = { 0, 32, 212 };

int high_scores[4];
int current_score;

current_score = 145; /* current_score is an int */
high_scores[0] = 145; /* high_scores[0] is an int */
```

In this example, both `current_score` and `high_scores[0]` are variables of type `int`. The computer treats and works with both in the same way, as shown by the assignment of 145 to both variables. You can assign to an array element a constant value or the value of a variable:

```c
current_score = 145;
high_scores[0] = 145;
high_scores[1] = current_score;
```

One very common way of assigning values to array elements is to do so within a loop. If you had an array of 500 elements and wanted to give each a value of 0, you could do the following:

```c
int big_array[500];
int count;
```
for (count = 0; count < 500; count++)
    big_array[count] = 0;

You do not have to give each element the same value when you use a loop. The example in Listing 10–1 will fill an array with odd numbers starting with the number 101.

**LISTING 10–1** Filling an array using a loop

```c
int odd_array[10];
int count;
int odd_num = 101;

for (count = 0; count < 10; count++)
{
    odd_array[count] = odd_num;
    odd_num += 2;
}
```

Each pass through the for loop in Listing 10–1 assigns an odd number to an element in odd_array and then increments odd_num by two. After the tenth and final pass through the for loop, the variables will have the values shown in Figure 10–4.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>odd_array[0]</td>
<td>101</td>
</tr>
<tr>
<td>odd_array[1]</td>
<td>103</td>
</tr>
<tr>
<td>odd_array[2]</td>
<td>105</td>
</tr>
<tr>
<td>odd_array[3]</td>
<td>107</td>
</tr>
<tr>
<td>odd_array[5]</td>
<td>111</td>
</tr>
<tr>
<td>odd_array[6]</td>
<td>113</td>
</tr>
<tr>
<td>odd_array[7]</td>
<td>115</td>
</tr>
<tr>
<td>odd_array[8]</td>
<td>117</td>
</tr>
<tr>
<td>odd_array[9]</td>
<td>119</td>
</tr>
<tr>
<td>count</td>
<td>10</td>
</tr>
<tr>
<td>odd_num</td>
<td>121</td>
</tr>
</tbody>
</table>

**FIGURE 10–4** Final values from Listing 10–1
Using Array Elements

You use array elements just as you use individual variables of the same type. For example, you can assign to an int variable freezing the value of the second element of the temperatures array as follows:

```c
int temperatures[3] = { 0, 32, 212 };  
int freezing;  
freezing = temperatures[1];  /* the 2nd array element */
```

Here's a code fragment that uses all three of the temperatures array elements:

```c
int temperatures[3] = { 0, 32, 212 };  
int freezing;  
int boiling;  
int zero;  
zero = temperatures[0];  /* zero = 0*/  
freezing = temperatures[1];  /* freezing = 32 */  
boiling = temperatures[2];  /* boiling = 212 */
```

Passing an Array Element to a Function

In Chapter 8 you learned about parameters and arguments. There you saw how communication occurs between functions. Listing 10–2 should refresh your memory.

**LISTING 10–2**  Function communication using a parameter and an argument

```c
main()
{
    int grade = 72;  
    Check_Score(grade);  /* pass the value of grade */
}

Check_Score(int number)
{
    number += 5;
```
DrawString("\pWe added a 5 point bonus.");

if (number >= 70)
    DrawString("\pYou passed.");
else
    DrawString("\pSorry, you failed.");
}

You also saw that if you wanted to change the value of a parameter you had to pass the address of a variable, rather than its value. Notice, in the above example, that after Check_Score() has executed, the value of grade will not have changed—the value of grade, not its address, was passed. Let's make a couple of minor changes to the preceding example—the addition of the & and * operators—and call the new program Listing 10-3.

**LISTING 10-3 Using a pointer as a parameter and an argument**

```c
main()
{
    int grade = 72;
    Check_Score(&grade); /* pass the address of grade */
}

Check_Score(int *number) /* number is a pointer */
{
    *number += 5; /* dereference the pointer*/
    DrawString("\pWe added a 5 point bonus.");

    if (number >= 70)
        DrawString("\pYou passed.");
    else
        DrawString("\pSorry, you failed.");
}

In the above example the value of grade will change from 72 to 77 after Check_Score() has executed.

Like other types of variables, array elements can be used as parameters. In Listing 10–4 a single element is passed.
LISTING 10–4 Using a single array element as a parameter

```c
main()
{
    Give_Pay_Info(hourly_wage[2]);
}

Give_Pay_Info(float pay);
{
    if (pay < 9.00)
        DrawString("You're not at the top wage yet.");
    else
        DrawString("You're at the top wage.");
}
```

Because `hourly_wage[2]` was not passed as a pointer, no lasting change can be made to it by `Give_Pay_Info()`. The `Give_Pay_Info()` routine just uses the value of `hourly_wage[2]` in an if statement test. Each element of the array is a float, not a pointer to a float. So you’ll have to pass a pointer if you want any changes made by a function to be permanent. Listing 10–5 shows an example.

LISTING 10–5 Using a pointer to an array element as a parameter

```c
main()
{

    Give_Pay_Info(&hourly_wage[2]);
}

Give_Pay_Info(float *pay);
{
    if (*pay < 9.00)
        DrawString("You're not at the top wage yet.");
    else
    {
        DrawString("You're at the top wage.");
        DrawString("We've added a dollar an hour bonus!");
        *pay += 1.00;
    }
}  ```
After Give_Pay_Info() has executed, hourly_wage[2] will be 10.40.

**Passing an Array to a Function**

Although it may not be obvious, an array name is a pointer. Consider the following declaration:

```c
int scores[3];
```

When you use an array name (scores) without the brackets or with an empty pair of brackets (scores[]), you are referring to a pointer to the *first element* of the array. This notation is used primarily when you want to use an entire array as a function argument. The C language does not allow the passing of an entire array as an argument. It does, however, allow the passing of a pointer to an array. Listing 10-6 provides an example.

**LISTING 10-6 Using a pointer to an array as an argument**

```c
main()
{
    int scores[3] = { 100, 200, 300 };  
    int total;

    total = Add_Scores(scores, 3);
}

int Add_Scores(int int_array[], int num)
{
    int sum = 0;
    int count;

    for (count = 0; count < num; count++)
        sum += int_array[count];

    return sum;
}
```

Listing 10-6 introduces a couple of new twists, so we'll examine it closely. The scores declaration in main() declares an array of three integers and initializes the array elements to values of 100, 200, and 300. An int variable called total is also declared. It will hold the sum of the values in the scores array. Figure 10-5 shows how memory might look after these
two declarations. Note in particular that scores is a pointer and that it contains the address of the first scores element, scores[0].

The body of main() has one assignment statement. The variable total is assigned the sum of the array scores by calling the Add_Scores() function. Look carefully at how the array is used in the call to the function:

```c
    total = Add_Scores(scores, 3);
```

Notice that only the array name is passed. Earlier we said that an array name is a pointer to an array. The effect is to pass a pointer to the entire array scores. The pointer points to the first element of the array. The second parameter, 3, is the size of the array—three elements.

Now let's look at the receiving end of the function call. The Add_Scores() function is defined as follows:

```c
    int Add_Scores(int int_array[], int num)
```

When an array is used as an argument, as above, use an array name followed by empty brackets. This lets the compiler know that you are declaring a pointer to an array. Remember, C does not allow you to pass an entire array, but it does allow you to pass a pointer to an array. The second argument is the number of elements that the array holds. Figure 10–6 illustrates this idea.

The Add_Scores() function does the work of adding the array elements using a for loop:

```c
    for (count = 0; count < num; count++)
        sum += int_array[count];
```
The loop counter, variable count, starts at 0, the first index to all arrays. The number of array elements is used as the stopping point. Finally, the array total, \( \text{sum} \), is returned to the calling routine, \( \text{main}() \).

If you've followed this discussion of arrays and pointers closely, you may have one question. If \( \text{Add_Scores()} \) was passed a \textit{pointer} to an array, how is it that we then went on to write \( \text{int_array[count]} \), using the pointer, \( \text{int_array[]} \), as if it were an array? Did we forget to dereference the pointer? As demonstrated by the fact that an array's name is a pointer, an array is a special C type. The concepts of arrays and pointers are inseparably bound. When scores was used as a parameter, a pointer to the first element in the array was actually passed. The function received this pointer and set \( \text{int_array} \) to point to the same address. So any operations that we perform on \( \text{int_array} \) are taking place on the array scores.

In summary, remember that when you use an array as a parameter, you must pass the name of the array. Make the corresponding argument of the receiving function an array name followed by empty brackets. Additionally, tell the receiving function the number of elements in the array by passing that value as well.

![FIGURE 10-6 Passing a pointer to an array](image)
Global Arrays

As with any variable, if you declare an array globally, any function can alter the array elements without the array or the element being passed to it. Listing 10–7 provides an example.

Listing 10–7 Global arrays are accessed by all functions.

```
#define NUM_ELEMENTS 3
int High_Scores[NUM_ELEMENTS]; /* array of 3 ints */

main()
{
    Fill_Array_With_Zeros();
    ....
    ....
}

Fill_Array_With_Zeros()
{
    int count;

    for ( count = 0; count < NUM_ELEMENTS; count++)
        High_Scores[count] = 0;
}
```

In Listing 10–7 High_Scores is declared before any of the functions, thereby making it global. Without passing it as a parameter, the program makes it known to all functions. To abide by our naming convention, uppercase letters are used for the first letter of each word to show that High_Scores is a global variable. Because it’s a global variable, any time an element of High_Scores is altered, the change remains in effect until that element is changed again. After the call to Fill_Array_With_Zeros() has been completed and control returns to main(), each element of High_Scores will have a value of 0.

Applications of Arrays

In introducing this chapter, we said that computers and computer programmers are known for keeping track of large amounts of data. Arrays provide one means of doing this record keeping. We finish the section on arrays by offering a practical use of an array.
Snowfall, rainfall—anything that is measured on a regular basis—can be stored in an array:

```c
#define MONTHS 12

float monthly_snowfall[MONTHS]; /* snowfall in inches */
monthly_snowfall[0] = 9.2; /* January snowfall*/
monthly_snowfall[1] = 7.0; /* February snowfall */
```

To process an array is to do something to it—examine it, sum the values in it, or perform any of a hundred operations. After the snowfall array is filled, it can be processed by checking to see how many months had snowfall greater than a foot:

```c
float monthly_snowfall[MONTHS]; /* snowfall in inches */
int over_a_foot = 0;
int count;

/* fill the array here */

/* process the array here */
for (count = 0; count < 12; count++)
{
    if (monthly_snowfall[count] >= 12.0)
        over_a_foot++;
}
```

Many programs that work with data use the following format:

1. Declare variables, including arrays.
2. If data is known in advance, fill the arrays.
3. If data is not known in advance, request it from the user.
4. Process the arrays.
5. Report the results to the user.

From reading this chapter, you know how to do steps 1, 2, and 4. In later chapters, you'll learn how to get user input from dialog boxes and how to report back to users by printing results in either dialog boxes or windows.
Structures

Arrays provide one way in which you can store large amounts of related data. Note that we say related data. That's because all the elements in an array must be of the same type. An array must be composed of all ints, all floats, or all of any other one data type. This is a very suitable system for many purposes. But there will be times when you'll want to keep track of different types of data. For this purpose C offers a very powerful and flexible data type called a struct. The struct type stands for structure. And like structures, structs can be designed and built in an infinite number of sizes and shapes.

Structure Template

A structure template describes a single type of structure. Here is one example:

```c
struct month_rec /* template for one month*/
{
    int month; /* number of this one month */
    float snowfall; /* snowfall in inches */
    float rainfall; /* rainfall in inches */
};
```

A structure template starts with the C keyword struct, followed by the template name, or tag—in this example, month_rec. A structure can be thought of as a record of information—thus the choice of the name month_rec. Enclose the body of the structure in braces. The body contains the structure members. A structure may contain as few or as many members as you wish. Each member is a declaration. The month_rec structure template has three members: the month of the year, stored as an integer; the month's snowfall in inches, which is a floating-point; and the month's rainfall, also a floating-point. Figure 10–7 illustrates this.

Declaring a Structure Variable

The definition of a structure template comes early in a program. It tells the compiler what the structure will look like, but it does not actually create a structure. Just as int and float are data types, not variables, the structure template is a type rather than a variable. What type is it? Whatever name you choose for the tag. This freedom to make a structure the type of your
own name and design makes a structure a *user-defined type*. You, the user, define it. Later in your program, when you declare your variables, you will declare any structs you need. In Figure 10–8 a month_rec struct variable and, for comparison, an int variable are declared.

**IMPORTANT**

You've just seen that the C keyword struct is used for two different purposes: to define a structure template and to declare a struct variable.

Once you have a structure template completed, you can declare as many variables of that type as you want:

```c
struct month_rec one_month;
struct month_rec first_month;
```


```c
struct month_rec
{
    int month;
    float snowfall;
    float rainfall;
};
```

After the above declaration, you would have four struct variables, each of month_rec type. That means each of the four variables would have a month, a snowfall, and a rainfall associated with it. Figure 10–9 shows how memory would look for one of these four variables.

### Accessing Structure Members

Earlier in this chapter you learned how to access the individual elements of an array by using a subscript:

```c
int auto_years[10];
auto_years[5] = 1991; /* 5 is the subscript */
```
Structures have their own way of allowing member access—you use a dot (.), the structure member operator. Let’s use the member operator on a variable of our month_rec struct type:

```c
struct month_rec
{
    int month;
    float snowfall;
    float rainfall;
};
struct month_rec one_month;

one_month.month = 1; /* month of January */
one_month.snowfall = 10.2; /* 10.2 inches of snow */
one_month.rainfall = 0.4; /* 0.4 inches of rain */
```
When you apply the member operator to a struct variable, as in the case of one_month.month, that struct and member together act like any variable of the member’s type. You can use one_month.month as you would any int variable. Similarly, you can use one_month.snowfall and one_month.rainfall as you would any float variable. Note that in the following example the snowfall and rainfall members are used on either side of the equal sign in assignment statements, just as any other float type would be used:

```c
struct month_rec
{
    int month;
    float snowfall;
    float rainfall;
};

struct month_rec one_month;
float  lot_of_rain = 15.0;
float    snow;

one_month.rainfall = lot_of_rain;
one_month.snowfall = 15.0;
snow = one_month.snowfall;
```

As you’ve seen, a struct variable can be a very useful data holder. The month_rec structure has three members, but you could create a structure template with 5, 10, or 100 members, each describing some aspect of a single month. To make a structure really useful, you might want to declare a month_rec variable for each of the 12 months:

```c
struct month_rec
{
    int month;
    float snowfall;
    float rainfall;
};

struct month_rec january_month;
struct month_rec february_month;
struct month_rec march_month;
```
struct month_rec december_month;

Useful, yes, but a bit awkward. And, luckily, there aren't 200 months in a year! The situation would improve greatly if it were possible to easily create a list of struct variables, all of the same type—perhaps a list of 12 month_rec variables, easily accessed through the use of a subscript. Can you think of a way to implement a list of structures?

**An Array of Structures**

An array is a list of variables, all of the same type. We complained earlier that although an array can be very useful, it has a limitation. All of its variables must be of the same type. This restriction could occasionally hamper us. As a matter of fact, that's how the whole subject of structures came about. Now, the array is back and about to redeem itself. Useful in its own right, when combined with structures it provides one of the most powerful of all programming tools—an array of structures.

You know about arrays, and you know about the month_rec structure. Here we combine the two to create an array of 12 month_rec variables—one variable for each month:

```c
struct month_rec all_months[12];
```

Is the requirement that all array elements be of the same type satisfied? Yes—each of the 12 elements is a month_rec struct. Now, when you want to refer to any one of the 12 struct variables, use an array subscript. In particular, your interest will be in individual members of a struct. Here's an example that assigns values to the three members of the first struct and the three members of the last struct:

```c
struct month_rec
{
    int month;
    float snowfall;
    float rainfall;
};
```

```c
struct month_rec all_months[12];
```

```c
all_months[0].month = 1; /* month of January */
all_months[0].snowfall = 10.2;
```
all_months[0].rainfall = 1.3;
all_months[11].month = 12;            /* month of December */
all_months[11].snowfall = 5.2;
all_months[11].rainfall = 0.4;

Notice where the subscript goes—after the array name, all_months. If you’re interested in the first struct, then you’re looking at all_months[0], regardless of which member you access. Don’t make the mistake of placing the subscript after the member name:

all_months.month[0] = 1;          /* NO! Wrong! */

Passing a Structure Member to a Function

If a structure member is a single variable, like an int or float, you can easily pass it to a function. Listing 10–8 passes a single element of a month_rec variable. Notice that the month_rec struct is declared before the definition of any functions. That makes the struct itself (but not month_rec variables) global and known to all functions.

LISTING 10–8  Passing a single struct element

struct month_rec    /* a global template */
{
    int month;
    float snowfall;
    float rainfall;
};

main()
{
    struct month_rec one_month;
    ....
    Check_Month(one_month.month);
    ....
}

Check_Month(int a_month)
{
    switch (a_month)
    {
        ....


Chapter I

Arrays and Structures

DrawString("\pKeep your New Year's resolution!");
break;

DrawString("\p'Tis the season to be jolly!");
break;

The month_rec struct member month is of type int, so the formal argument of Check_Month() must be an int—as argument a_month is. The preceding listing doesn't alter the struct variable member, so you needn't pass a pointer. If you want to make a lasting change to a member, pass a pointer to it. See Listing 10-9.

LISTING 10-9 Using a pointer to a struct member as a parameter

```c
struct month_rec /* a global template */
{
    int month;
    float snowfall;
    float rainfall;
};

main()
{
    struct month_rec one_month;
    one_month.rainfall = 5.0;

    Add_Inch(&one_month.rainfall);
}

Add_Inch(int *number)
{
    *number += 1.0;
}
```
If you have an array of structures, can you still pass one member of one structure? Yes. Just specify which array element the structure member belongs to, as shown in Listing 10–10.

LISTING 10–10  Using a pointer to an array struct member as a parameter

```c
struct month_rec /* a global template */
{
    int month;
    float snowfall;
    float rainfall;
};

main()
{
    struct month_rec one_month[12];

    one_month[0].rainfall = 5.0;
    Add_Inch(&one_month[0].rainfall);
}

Add_Inch(int *number)
{
    *number += 1.0;
}
```

**Passing a Structure to a Function**

Many of the new C compilers, including THINK C, allow you to pass an entire structure to a function. If you aren’t going to change any values, you need not pass a pointer. In the following fragment, an entire month_rec structure is passed to Check_Month(). The first of the 12 structures from an array of structures is passed:

```c
struct month_rec all_months[12];

....
....
Check_Month(all_months[0]); /* pass a structure */
....
```
Check_Month(struct month_rec a_month)
{
    if (a_month.month == 1)
    
        DrawString("\pOnly 11 months 'til Xmas!");

    
    Recall that an argument always includes a variable type and a variable
    name, as in this example:

    Add_One(int num)

    An argument that is a struct is no different. Notice that for the argu­
    ment of Check_Month() the full argument type is included—the keyword
    struct and the struct tag, month_rec.

    If you want to change one or more members of a struct variable, then,
    as always, pass a pointer. In Listing 10–11 a pointer to a single structure
    from an array of structures is passed.

    LISTING 10–11 Using a pointer to a struct as a parameter

    struct month_rec                /* a global template */
    {
        int month;
        float snowfall;
        float rainfall;
    
    main()
    {
        struct month_rec all_months[12];

        Set_To_December(&all_months[0]);

    
    Set_To_December(struct month_rec *a_month)
    {
Think THINK C

a_month->month = 12;
}

If you were napping, Listing 10–11 probably woke you up. Where did the -> come from, and what is its purpose? Follow along.

When you access a single member of a struct variable, you do so as follows:

```c
struct month_rec one_month; /* declare a struct */
one_month.snowfall = 2.0;
```

The form for accessing a single member of a struct pointer is slightly different. Here's how you achieve the same result, using a pointer:

```c
struct month_rec *month_ptr; /* a pointer to a struct */
month_ptr->snowfall = 2.0; /* accessing a member */
```

The -> operator is formed by typing a hyphen (-), followed by a greater than sign (>).

The main() function in Listing 10–11 did not declare a pointer to a structure. The address of a structure variable was passed to a function, where it was received as a pointer by that function. Listing 10–12 shows the declaration of a struct pointer, setting it to point to a struct variable and assigning values to the field of the struct variable through the use of the pointer to it.

**LISTING 10–12 Declaring a pointer to a struct**

```c
struct month_rec /* a global template */
{
    int month;
    float snowfall;
    float rainfall;
};

main()
{
    struct month_rec one_month; /* a struct variable */
    struct month_rec *one_month_ptr; /* a pointer to a struct */

    one_month_ptr = &one_month; /* point to the struct*/
    /* variable one_month */
```
one_month_ptr->month = 3;  /* assign values to the */
one_month_ptr->snowfall = 5.2;  /* the fields of the */
one_month_ptr->rainfall = 0.0;  /* struct by using the*/
                               /* pointer to the struct */

Chapter Summary

Computers are most needed when large amounts of data are involved—especially data that is similar, or related in some way, such as records. In order to keep track of related data, C offers two programming tools—the array and the structure.

An array is a group of data of the same data type. A single array variable can hold any number of variables, provided they’re all of the same type. To create an array variable, you specify the type of variable the array will hold, followed by the array name. The array name is followed by the number of elements in the array, enclosed in brackets. The following declares an array of five integers:

```c
int team_score[5];  /* an array of five integers */
```

An array is adequate for storing a collection of single values, such as 10 integers. But to hold a collection of values that are of different types, such as an informational record, the struct type is used.

A single struct variable can have members that are of different types. There are two steps to creating a variable of the struct type. First, you create the type itself by using the keyword struct, followed by the struct tag. The tag is the name of this new type you are creating. Then, you enclose the members of the struct in braces. The following is the declaration for a struct named month_rec. It has three members—one integer and two floating-points:

```c
struct month_rec  /* template for one month */
{
    int month;  /* month, 1 through 12 */
    float snowfall;  /* snowfall in inches*/
    float rainfall;  /* rainfall in inches*/
};
```
Once the struct type is defined, you can create variables of that type by using the struct keyword followed by the tag and then the variable name:

```c
struct month_rec one_month;
```

To take full advantage of the struct type, programmers often create an array of structs, which enables them to keep track of a series of structures.
Chapter 11

Introduction to QuickDraw

QuickDraw is the name given to a large group of graphic routines that resides in the Toolbox. The QuickDraw routines allow you to add impressive graphics to your programs. QuickDraw is what you use to create lines, rectangles, circles, pictures, and fancy text.

As you explore QuickDraw, you'll learn how to use Toolbox routines to create your first graphics.

Finally, to demonstrate how much you can accomplish in just one chapter, you will combine QuickDraw with a little ANSI C to create some simple animation effects.

The QuickDraw Coordinate System

The QuickDraw Toolbox routines are responsible for all of the graphics produced on the Macintosh. QuickDraw is used in icons, menus, windows,
patterns, fonts, and every other part of the Macintosh interface. We want to establish the importance of QuickDraw so you'll be patient as the details are explained. The details will allow you to fully understand and harness the power of QuickDraw.

There is a QuickDraw routine to help you draw in a window just about anything you want. When you decide to draw a shape in a window, you'll know just where you want the shape to be drawn. How does QuickDraw know? So it can draw to specific window locations, QuickDraw uses a coordinate system to refer to exact screen locations. Pixels form the basis of this system.

**Pixels**

The Macintosh screen is said to be *bit-mapped*. The screen, and thus a window that is on the screen, is divided into small squares. How small are these squares? There are 72 squares in one linear inch, and one square inch of your monitor screen contains about 5000 squares. A single square is a pixel. On a black-and-white screen, each pixel can be turned on or off. A pixel that appears black is considered on. A white pixel is off. On a color monitor, each pixel can take on any one of many colors. When you draw on the Macintosh screen, you're turning certain pixels on. To draw a line, you turn on (make black) the pixels that fall along the path of the line. Figure 11-1

![Figure 11-1](image)

**FIGURE 11-1** Turning pixels on to draw a line
illustrates this. Note that the pixels shown are enlarged and represent only a small portion of a real Macintosh screen.

Notice the stair-step appearance of the diagonal line shown in Figure 11-1. This effect is called the jaggies, and it is attributable to the size and square shape of screen pixels. Be aware that lines that are not exactly horizontal or vertical are subject to this slightly jagged look.

Pixels are a part of hardware—they're built into your monitor screen and never move. The way you refer to an individual pixel varies, however, as you'll discover in the following section.

The Coordinate System

The coordinate system is QuickDraw's way of locating a particular pixel on the screen. In this system, each pixel is defined by a horizontal and a vertical value. Pixel numbering for both the horizontal and the vertical starts at 0 and begins at the upper-left corner of the screen. The number of pixels on a screen depends on the size of the screen. The nine-inch screen found on the MacPlus and Mac Classic has 512 pixels from left to right and 342 pixels from top to bottom. Figure 11-2 shows an enlarged view of a portion of a nine-inch Macintosh screen.
Using this numbering system, you can describe any pixel by giving its horizontal and vertical values. Use the upper-left corner of the screen as the reference point. This is the origin, and its coordinate pair is (0,0). Figure 11–3 shows the coordinate system numbers of two arbitrarily chosen pixels. Note that the horizontal value is given first.

**Pixels and Windows**

We have been using the upper-left corner of the screen as the origin, but other origins can be used as well. Imagine a Macintosh program that places a window on the screen and draws a rectangle in it. As the window moves, QuickDraw must redraw the rectangle in the window. To make this possible, the window has its own private coordinate system, which allows it to redraw the rectangle in the same relative location in the window every time the window moves. The system is the same as the system used for the screen. The only difference is that QuickDraw uses the origin in the upper-left corner of the window rather than the fixed origin in the upper-left corner of the screen. See Figure 11–4.

When the Simulator software executes a QuickDraw routine, it opens a window and carries out the command. Whatever is drawn will be drawn in a window with the upper-left corner of the window as the origin.
Chapter 11  Introduction to QuickDraw

The elements that are unique to the Macintosh, including windows, menus, and a multitude of graphic effects, are all created using Toolbox routines. Chapter 2 introduced you to the Toolbox. There it was defined as a collection of small programs, or routines, that you can access through a programming language such as C. We also discussed Toolbox routines in Chapter 4.

To make use of the Toolbox, you must know the names of the routines that are in it. Apple's series of reference books *Inside Macintosh; Volumes I-VI*, contains the definitive listing of Toolbox routines. If you don't have *Inside Macintosh*, don't worry. In the remainder of this book, and throughout the Simulator software, we'll introduce, define, and give examples of the use of the most commonly used routines.
Access to the Toolbox

You can access a Toolbox routine just as you access any other C routine—you simply call, or invoke, it. Throughout this book we will refer to QuickDraw functions as either functions or routines—the words are interchangeable. To make use of a QuickDraw operation, type the routine name, then follow it with the proper number of parameters between parentheses. Consider the Toolbox routine PaintRect(), which paints, or shades, the contents of a rectangle. It has one parameter—a pointer to the rectangle it is to shade. Here's an example of its use in a C program:

PaintRect(&the_rect); /* invoke the routine */

Before using PaintRect() or any other QuickDraw routine, you must initialize some of QuickDraw's own variables. QuickDraw has global variables and internal data that require a one-time initialization. To do this, make a call to the routine InitGraf(). You give InitGraf() one parameter—a pointer to the global variable thePort. This is a system global variable, so you need not declare it in your program:

InitGraf(&thePort);

Other initialization routines must also be called before you begin a Macintosh program. The Note below summarizes this initialization.

NOTE

Before you write the bulk of your Macintosh C language program, one of the very first things you must do is initialize QuickDraw and the other parts of the Macintosh interface. There are several Toolbox routines that do this for you. The following is an initialization routine that we'll use in future example programs. We suggest that you incorporate this routine into your programs. You don't have to know anything more about it now—we'll discuss it again in Chapter 19 when we write our first complete program.

Initialize_Toolbox()
{
    InitGraf(&thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(OL);
FlushEvents(everyEvent, O);
InitCursor();
}

Please note that the order in which the initialization routines are called is important. Call them in the sequence presented above.

Using the Toolbox to Draw a Line

Your first experiment with graphics will be a simple one—draw a line across part of a window. You'll use two QuickDraw Toolbox routines to accomplish this. First, you want to establish the starting point of the line. You do this with the QuickDraw MoveTo() routine. MoveTo() moves an invisible "pen" to the location specified by the two parameters passed to MoveTo(). We'll have more to say on this mystical pen later in this chapter. The two MoveTo() arguments represent the horizontal and vertical pixel locations to move to. The next thing you want to do is to draw the line. The LineTo() routine also makes use of two arguments—the horizontal and vertical pixel locations to draw the line to. Here's an example:

MoveTo(75, 50);        /* starting location of the line */
LineTo(250, 150);      /* draw line to the end location */

The line created by these two lines of code is drawn from window location (75,50) to window location (250,150). It appears in the window shown in Figure 11-5.

What would happen if you used out of range values like the following as arguments in these routines:

LineTo(1000, 1000);    /* We're past the edge of the window! */

QuickDraw is smart enough to know where the edges of the windows are and where the edges of the screen are. The window being drawn to is called the active window. The line will be drawn to the edge of the active window and then stop. This action is called clipping.

LineTo() has a companion routine called Line(). While the LineTo() routine draws a line to a specific endpoint, the Line() routine draws a line a specific distance. The Line() routine accepts two arguments—the horizontal and vertical pixel distances of the line. The next page shows an example:
FIGURE 11-5  The result of calling MoveTo() and LineTo()

MoveTo(100, 50);  /* starting location */  
LineTo(150, 150); /* draw line to the end location */  
MoveTo(100, 50);  /* move back to starting location */  
Line(150, 150);   /* draw a second line */

The lines created by the Line() and the LineTo() routines are of different lengths, even though their arguments are the same. Line(150,150) will always draw a line 150 pixels in the horizontal direction and 150 pixels in the vertical direction. The line resulting from LineTo(150,150) will vary in length. Its length depends on the starting point of the line. In the LineTo() routine, the two arguments represent the endpoint of the line, not its length. A line's starting point is determined by a MoveTo() routine or by the endpoint of a previously drawn line. The result of the above code appears in the window shown in Figure 11–6.

Where's the C?

With built-in routines like MoveTo() and LineTo() doing all the drawing, where does regular ANSI C fit in? If you want to use these powerful Quick-Draw routines, you must use ANSI C statements. Imagine that you want to
draw not just one line but 10 horizontal lines. The code fragment below provides one solution:

```c
int v;
int count;

v = 50;                        /* vertical coordinate of first line */
MoveTo(20, v);                 /* starting location of the first line */

for (count = 0; count < 10; count++) /* loop ten times */
{
    LineTo(175, v);          /* draw a line */
    v += 10;                  /* increment vertical by 10 pixels */
    MoveTo (20, v);           /* starting location for next line */
}
```

The lines created by this code appear in the window shown in Figure 11-7.

This example shows a combination of ANSI C (the for loop and assignment statements) and Macintosh-specific C (the calls to the Toolbox routines). The Macintosh programs you create will have this same format. Note how adding just a few Macintosh routine calls to your program can
create screen-filling graphics. Macintosh programs thrive on graphics—this is why ANSI C by itself is not enough. Macintosh C picks up where ANSI C leaves off and adds the graphics capabilities you need.

## The Graphics Pen

The name QuickDraw implies drawing. To extend the analogy, Apple refers to a *graphics pen*. The pen never shows up on your screen—it's invisible. So why is it even defined? To aid you in setting the characteristics of the lines that QuickDraw will draw. In most cases the thickness of a line is only one pixel. If you want to draw a line that is thicker than that, you can. The QuickDraw routine PenSize() allows you to vary the pen size. PenSize() accepts two arguments—the horizontal \((h)\) length of the line in pixels and the vertical \((v)\) thickness of the line in pixels. The format of the pen sizing routine is:

\[
\text{PenSize}(h, v);
\]

The following PenSize() call sets the pen so it draws a line five pixels across and eight pixels high:

\[
\text{PenSize}(5, 8);
\]
When you draw a line with the LineTo() or the Line() routine, QuickDraw draws it with a vertical thickness specified by the second PenSize() argument. The horizontal length of the line will be the distance specified in the Line() routine plus the width specified by the first PenSize() argument. Figure 11-8 shows the result of a Line(200,0) call using three different pen sizes.

Notice in Figure 11-8 that the horizontal lengths of the lines vary even though each line is created with the same Line(200,0) command. This is because the line is drawn 200 pixels in length and then the horizontal width of the pen is added.

You manipulated the graphics pen earlier, although we made no mention of it—it really is invisible. When you drew a line, you established the starting point of the line by using the QuickDraw MoveTo() routine.

What MoveTo() does is move the graphics pen to the location specified by the MoveTo() arguments, readying the pen for drawing. The LineTo() routine starts drawing a line from the current location of the graphics pen.

Rectangles

In the previous section you saw how QuickDraw is used to draw lines. You're now ready to move into the second dimension—shapes. The Macin-
tosh makes it easy for you to draw a variety of shapes, including rectangles, ovals, and round rectangles.

We'll cover rectangles first; they're the simplest QuickDraw shape. QuickDraw has several routines for creating and operating on rectangles. The ones we will cover in this section are SetRect(), FrameRect(), and EraseRect(). Figure 11–9 illustrates the way the Toolbox defines a rectangle by specifying the coordinates of its top-left corner and its bottom-right corner.

The Rect Data Type

To hold the coordinates that make up a rectangle, you need a new data type. The Rect data structure is specific to Macintosh programming; you won't find it in ANSI C. Apple has defined a Rect as a structure that holds the upper-left and lower-right corners of a rectangle. Recall from Chapter 10 that a structure is a data type that allows you to group members together. The members of a Rect structure are the corners of the rectangle. The way you declare Rects is similar to the way you declare other data types. Here's how you would declare a rectangle named the_rect:

```c
Rect the_rect; /* data type followed by variable name */
```

Simple enough. But the rectangle has no initial boundaries. You must define its size. Below, four integers are declared to hold the coordinates of
the rectangle. As each is declared, it is given an initial value. These values will create a rectangle with the coordinates of the rectangle displayed in Figure 11-9:

```c
int left = 100; /* declare and initialize your */
int top  = 50;  /* rectangle's coordinates */
int right = 250;
int bottom = 120;
```

Now you're ready to define your rectangle. To do this, use the Toolbox routine `SetRect()`. The `SetRect()` routine demands five arguments—a pointer to the Rect to define, followed by the coordinates of the rectangle. The coordinates are all int values. You must give them in the order shown below:

```c
SetRect(&the_rect, left, top, right, bottom);
```

Recall from Chapter 9 that a pointer is an address. `SetRect()` asks for a pointer to a Rect, while the declaration is that of a Rect. You've seen in previous chapters that placing the & operator before a variable provides the address of that variable. Therefore using &the_rect satisfies the `SetRect()` pointer requirement.

Notice that variables, rather than the values, have been used for the last four parameters. The following would give the same result:

```c
SetRect(&the_rect, 100, 50, 250, 120);
```

Why choose the first method? It lets someone looking at your code know just what is going on. Using distinctive names helps you avoid mixing up the coordinates, too. Whenever possible, make it a habit to use variables or #defines instead of numbers.

As you saw with `SetRect()`, the QuickDraw routines that work with rectangles require a pointer to a rectangle, not the rectangle itself. There are two ways to do this: use the & operator to pass the address of a Rect variable or declare a pointer to a Rect and then pass the pointer. Here's an example of the first method:

```c
int left = 100; /* declare and initialize the */
int top  = 50;  /* rectangle's coordinates */
int right = 250;
int bottom = 120;
Rect the_rect;  /* Rect variable */
SetRect(&the_rect, left, top, right, bottom);
```
Now, an example of the second method:

```c
int left = 100;           /* declare and initialize the */
int top  = 50;            /* rectangle's coordinates */
int right = 250;          
int bottom = 120;         
Rect the_rect;            /* Rect variable */
Rect *rect_ptr;           /* pointer to a Rect */
rect_ptr = &the_rect;     /* point to the_rect */

SetRect(rect_ptr, left, top, right, bottom);
```

### Displaying a Rect

Although you've defined your rectangle with `SetRect()`, you still won't be able to see it in a window. The `SetRect()` routine defines only the boundaries of a `Rect`; it doesn't display the `Rect` itself. To do so, you need one more QuickDraw routine—`FrameRect()`. The `FrameRect()` routine needs just one argument—a pointer to the rectangle you want to display:

```c
FrameRect(&the_rect);
```

Let's put it all together now to create a code fragment that draws a rectangle:

```c
int left = 100;           
int top  = 50;            
int right = 250;          
int bottom = 120;         
Rect the_rect;

SetRect(&the_rect, left, top, right, bottom);
FrameRect(&the_rect);
```

Execution of these lines would draw the frame of a rectangle, as shown in Figure 11-10.

### Hiding a Rect

Now that you can draw a rectangle, can you get rid of it just as easily? The `EraseRect()` routine will do just that. Supply a rectangle as the argument, and `EraseRect()` will obscure it. If you add an `EraseRect()` call to the above
example, a rectangle will be drawn and then immediately erased, resulting in a blank window:

```c
int left = 100;
int top = 50;
int right = 250;
int bottom = 120;
Rect the_rect;
SetRect(&the_rect, left, top, right, bottom);
FrameRect(&the_rect); /* outline the rect */
EraseRect(&the_rect); /* erase the rect */
```

The rectangles that you have just drawn have taught you the basics of shape drawing. You'll learn about the remaining shapes and other QuickDraw graphic concepts in the next chapter. You've covered a lot of ground in this chapter, so it seems like a good time to take a break. It's time to have a little fun with some of the concepts that you've learned.

# Animation, Already?

Graphics are interesting, but the Mac can do so much more. From ANSI C, you know how to create a loop. And you've just learned how to use QuickDraw Toolbox routines to create graphic shapes. Let's apply these two concepts and create a simple case of animation—a rectangle that moves rapidly across a window. The code fragment on the next page will do just that:
FIGURE 11-11 Growing rectangles

```c
int left = 50;
int top = 50;
int right = 100;
int bottom = 100;
int count;
Rect the_rect;

for (count = 0; count < 300; count++) /* loop 300 times */
{
    EraseRect(&the_rect);
    SetRect(&the_rect, ++left, top, ++right, bottom);
    FrameRect(&the_rect);
}
```

If you're following along in the Simulator software, you see a rectangle that starts on the left side of a window and travels across to the right side. Granted, we've all seen smoother animation, but you get the point. With just a simple ANSI C for loop and a couple of QuickDraw routines, we've hinted at the power within your grasp.

In future chapters you'll explore ways to create more polished effects than those used in the preceding example. For now, we end this chapter with one more example that contains some animation. Again a for loop is used. This time we increment and decrement rectangle coordinates so that each pass through the loop draws a rectangle two pixels larger in each direction. Figure 11–11 shows that this gives the effect of a growing rectangle:
```c
int left = 250;
int top = 130;
int right = 250;
int bottom = 130;
Rect the_rect;
int count;

for (count = 0; count < 50; count++) /* loop 50 times */
{
    left -= 2; /* subtract 2 */
    top  -= 2;
    right += 2; /* add 2*/
    bottom += 2;
    SetRect(&the_rect, left, top, right, bottom);
    FrameRect(&the_rect);
}
```

**Chapter Summary**

QuickDraw is the name given to a large group of graphic routines that resides in the Toolbox. You can use a QuickDraw routine in your own program if you know the routine's name and the arguments it needs.

The Macintosh screen is composed of a grid of pixels. You reference individual pixels by the coordinate system that defines the location of each pixel.

To form a line, you turn on, or darken, pixels. The Line() and LineTo() routines are used to draw lines. They draw with what Apple refers to as the *graphics pen*. To move this invisible graphics pen to a particular screen location, you use the Toolbox routine MoveTo(). Then you draw your line. You can vary the thickness of a line by adjusting the size of the graphics pen with a call to the QuickDraw routine PenSize().

You define a rectangle with the Rect data type. The Rect data type is unique to Macintosh C—it is not a part of ANSI C. The SetRect() routine defines the size of a rectangle but doesn't show the rectangle. To display the rectangle, you use FrameRect(). You can erase the same rectangle with the EraseRect() routine.

You learned that the QuickDraw routines that operate on rectangles require a pointer to a Rect as an argument. Recall that a pointer is nothing more than an address. To pass a pointer to a rectangle, first declare a Rect
variable, then place the & operator in front of the Rect variable name when passing it as a parameter. A second way to pass a pointer to a rectangle is to declare a variable that is a pointer to a Rect.

In this chapter you've been introduced to several of the QuickDraw routines. When you consider that there are more than 150 QuickDraw routines in the Toolbox, the power of QuickDraw becomes very apparent. The names of the QuickDraw routines and all of the other Toolbox routines are found in Apple's series of reference books *Inside Macintosh, Volumes I-VI.*
In the last chapter you were introduced to the basics of QuickDraw. In this chapter you will dig deeper into QuickDraw. QuickDraw is an integral part of the Macintosh philosophy, so it is essential that you have a thorough understanding of it before moving on to other topics.

One of the chores handled by QuickDraw is the placement of text on the Macintosh screen and setting the look of that text. In this chapter you’ll learn how to display text in any style that you want.

In this chapter you’ll pick up where the previous chapter left off—with rectangles. You’ll see how rectangles are the basis of other shapes, including ovals and round rectangles.

Here you’ll discover how to use patterns to enhance the many shapes you draw.

Finally, the discussion on shapes will lead you into an introduction to graphic ports, which allow the Macintosh to draw to the correct window when there is more than one window on the screen.
Patterns

The previous chapter showed you how to define and frame rectangles. Before we continue with the topic of shapes, we’ll show you how you can spice up the rectangle by filling it with different patterns.

In the previous chapter we introduced the concept of the graphics pen, and showed that the pen is responsible for drawing lines to a window. In that chapter you varied a characteristic of the pen using the Toolbox routine PenSize() to change the dimensions of the pen point. Now you’ll see how you can alter another characteristic of the pen—its pattern. A pattern is an eight-pixel by eight-pixel square bit image that, when repeated, gives an area a uniform look. This square bit image can be “stamped” repeatedly to fill an entire area with the pattern currently in use.

As it fills the area you specify, QuickDraw is clever enough to align the square bit pattern pieces so there is no overlap or mismatch, but rather, a smooth continuous pattern. See Figure 12–1.

Macintosh C has a special data type associated with patterns—the Pattern type. Five standard, predefined Patterns are available for your use—
white, ltGray, gray, dkGray, and black. Use the Toolbox routine PenPat() to change the pattern of the pen. Pass one of the Patterns as the sole parameter. Below is the call to change the pen pattern to light gray:

PenPat(ltGray);

From this point on, QuickDraw will use this pattern, rather than its old standby, solid black. Keep in mind that not until you change the pen pattern with another call to PenPat() will the drawing pattern change.

The System File that resides in your System Folder contains more patterns than the five standard ones just mentioned. It contains a pattern list that holds many of the patterns you see on the palettes in popular Macintosh paint and draw programs. To access them, you need to know more about resources, which we cover in the next chapter. So for now, be content with the five standard patterns and with the knowledge that you’re always just a Toolbox call away from changing the look of the pen.

Now that you can change the pattern of the pen, what can you do with it? You’ll make use of what you’ve just learned about patterns in the very next section.

**Shapes: Rectangles Revisited**

Your introduction to rectangles came in the previous chapter. Though you learned a lot, you did not fully exploit the QuickDraw routines that operate on rectangles. With that said, you’re ready for...

**More Operations on Rectangles**

Chapter 11 showed you how to define a rectangle using SetRect(), how to frame it with FrameRect(), and how to erase it using EraseRect(). Now that you know something about patterns, you’re ready to examine two routines that allow you to create rectangles filled with patterns—rectangles that are much more impressive than the simple outlined ones you drew in Chapter 11.

Before you use either of the routines we’re about to discuss, you must define the rectangle to which you want to add a pattern. Recall that the SetRect() routine does just that—it defines a rectangle without drawing anything to the screen. SetRect() pairs screen coordinates for a rectangle with the name of a rectangle. After defining the rectangle, use the PaintRect() routine to draw a rectangle filled with the current pen pattern. The fol-
lowing draws a rectangle shaded light gray. The result of the code is shown in Figure 12–2:

Rect pattern_rect;

SetRect(&pattern_rect, 50, 50, 200, 200);
PenPat(ltGray);
PaintRect(&pattern_rect);

A second Toolbox routine, FillRect(), gives the same result as PaintRect() except that you can specify the rectangle’s pattern as a parameter rather than using a separate PenPat() call. Using FillRect() changes the pen pattern for just one call. After the call to FillRect(), the pen pattern returns to its previous state. The following code fragment will draw two patterned rectangles, as shown in Figure 12–3:

Rect first_rect;
Rect second_rect;

SetRect(&first_rect, 50, 50, 200, 200);
SetRect(&second_rect, 250, 50, 450, 200);

PenPat(black);
FillRect(&first_rect, ltGray);
PaintRect(&second_rect);
Notice in Figure 12–3 that the second rectangle, second_rect, is drawn in black, even though the rectangle drawn just before it was drawn in light gray. The FillRect(&first_rect, ltGray) call set the pen to light gray while drawing the rectangle, then set the pen back to its previous state, black, from the PenPat(black) call.

The final rectangle routine we'll cover is one that examines the state of the drawing area before it operates on a rectangle. InvertRect() inverts every pixel within a specified rectangle. All black pixels within the rectangle boundaries become white, and all white pixels turn black. Here's an example:

Rect the_rect;
SetRect(&the_rect, 20, 20, 400, 250);
FillRect(&the_rect, ltGray);
SetRect(&the_rect, 100, 50, 250, 200);
InvertRect(&the_rect);

Figure 12–4 shows the result of the above code.

This example uses SetRect() calls to define two rectangles. The coordinates supplied to the second SetRect() call define a rectangle whose bound-
InvertRect() inverts the pixels within its boundaries.

Notice that this example uses the Rect variable the rect in two SetRect() calls. Like any other variable, the value of the rect can be changed. The second call to SetRect() changes the coordinates of the rect, and the coordinates from the first call are lost.

We've devoted several pages of this chapter and the previous one to rectangles. We had good reason for this thorough coverage: QuickDraw uses the rectangle as the basis for several different shapes, not just for the rectangle itself. Before you can perform graphic operations on rectangles, squares, ovals, circles, and round rectangles, you must define a rectangle. Now that you have a thorough understanding of the mighty rectangle, we can cover the remaining shapes in a briefer fashion.

**Ovals**

QuickDraw uses a very simple and elegant concept to define an oval. You first set up a rectangle with a call to SetRect(). Then you use the Toolbox routine FrameOval(), passing the defined rectangle as the only parameter.
SetRect() sets up the invisible rectangle that will serve as the oval’s boundaries.

FrameOval() inscribes an oval within these boundaries.

QuickDraw inscribes within the defined rectangle the largest oval that will fit in that rectangle, as shown in Figure 12-5.

The following code fragment sets up the boundaries of an oval and frames it:

Rect the_rect;
SetRect(&the_rect, 20, 20, 300, 250);
FrameOval(&the_rect);

QuickDraw has five routines you can use to operate on ovals. Analogous to the routines used for working with rectangles, they are FrameOval(), PaintOval(), FillOval(), InvertOval(), and EraseOval(). Each of these routines, with the exception of FillOval(), has just one parameter—a pointer to the rectangle in which the oval is to be inscribed. FillOval() also requires the pen pattern with which the oval is to be filled. The following are examples of calls to these routines:
Rect the_rect;
Pattern the_pat;

the_pat = ltGray;
FrameOval(&the_rect);
PaintOval(&the_rect);
FillOval(&the_rect, the_pat);
InvertOval(&the_rect);
EraseOval(&the_rect);

Round Rectangles

The last shape we'll cover is the round rectangle—a rectangle with rounded corners. You'll notice that the Macintosh uses this shape when it draws buttons.

To create a round rectangle you first set up a boundary rectangle using a SetRect() call. The QuickDraw routine FrameRoundRect() frames the round rectangle. Here's an example:

```c
int corner_width = 40;
int corner_height = 50;
Rect the_rect;

SetRect(&the_rect, 50, 100, 200, 150);
FrameRoundRect(&the_rect, corner_width, corner_height);
```

The FrameRoundRect() routine uses the second and third of its three parameters to determine the degree of rounding it should apply to the corners of the rectangle. These two values form the boundaries of an imaginary oval set in each of the corners of the SetRect() rectangle. Figure 12-6 illustrates this.

As you may have already guessed, QuickDraw has a set of routines devoted to round rectangles—FrameRoundRect(), PaintRoundRect(), FillRoundRect(), InvertRoundRect(), and EraseRoundRect(). Each of these routines, with the exception of FillRoundRect(), requires three parameters—a pointer to the rectangle that is the basis of the round rectangle and the width and height of the imaginary oval that determines the degree of corner-rounding. FillRoundRect() also requires the pen pattern with which the rectangle is to be filled. The following are examples of calls to the round rectangle routines:
SetRect() sets up the invisible rectangle that will serve as the round rectangle's boundaries.

FrameRoundRect() frames a rectangle within SetRect()'s boundaries, rounding the corners according to the parameters passed to FrameRoundRect().

FIGURE 12-6 A rectangle with rounded corners

Rect the_rect;
Pattern the_pat;
int corner_width;
int corner_height;

FrameRoundRect(&the_rect, corner_width, corner_height);
PaintRoundRect(&the_rect, corner_width, corner_height);
FillRoundRect(&the_rect, corner_width, corner_height, the_pat);
InvertRoundRect(&the_rect, corner_width, corner_height);
EraseRoundRect(&the_rect, corner_width, corner_height);

Shape Summary

The code fragment below creates a circle, a square, and a round rectangle. Notice that a circle is simply a special case of an oval—an oval inscribed within a square. Likewise, a square is a special case of a rectangle; the SetRect() routine defines a rectangle with equal sides. Also notice that to create any of these shapes, you must first define a bordering rectangle:
Rect circle_rect;
Rect square_rect;
Rect round_rect;
int corner_width;
int corner_height;

SetRect(&circle_rect, 20, 20, 100, 100);
FrameOval(&circle_rect);
SetRect(&square_rect, 130, 20, 210, 100);
FrameRect(&square_rect);
SetRect(&round_rect, 210, 150, 360, 200);
corner_width = 40;
corner_height = 40;
FrameRoundRect(&round_rect, corner_width, corner_height);

The resulting shapes appear in Figure 12–7.
In Chapter 11 we said that the Macintosh screen is bitmapped. The Mac’s screen is a collection of pixels, and you create graphics on the screen by turning on selected pixels. Similarly, when your Macintosh puts text on the screen it simply turns on certain pixels to form the text characters. To the Macintosh, the process of drawing a text character to the screen is very similar to the process of drawing a shape. Figure 12–8 shows the pixel composition of two characters and two small shapes. The inset shows what you would see on the screen.

Text and graphics are normally thought of as two very different entities. Why does the Macintosh make so little distinction between the two? Here are three reasons:

- Because it is thought of in terms of graphic pixels, text can be drawn to very specific coordinate locations.
The look of text can easily be modified by turning on extra pixels, as in drawing boldfaced words.

Text and graphics can be mixed—words can be drawn over and through graphic figures.

**Drawing a String**

Now that you know something about how the Macintosh draws text, how do you tell the Mac what text to draw? If you guessed that a Toolbox call is involved, congratulate yourself now. The routine is DrawString(), and as its name implies, it draws a string to the screen. A string is a collection of characters that is treated like one variable. The string is prefaced with \p and must be in double quotes. To draw the string Testing 123, you would do the following:

```c
DrawString("\pTesting 123");
```

Why the cryptic \p notation? The Macintosh Toolbox was written with the intention of using Pascal as the programming language to access its routines. The \p tells your C compiler to convert the string to Pascal format.

All of Apple's Toolbox software was originally written in Pascal and assembly language. In Pascal, the DrawString() routine calling convention is:

```pascal
DrawString(s: Str255);
```

where Str255 is an Apple-defined Pascal data type that holds a string of up to 255 characters. The Toolbox is expecting a string in this Pascal format. If you were programming in Pascal, you could do the following:

```pascal
DrawString('The Toolbox knows this is a string!');
```

The Pascal compiler would be smart enough to know that the text that lies between the single quotes should be placed in Str255 format for the Toolbox. When programming in C, however, you must explicitly tell your compiler to convert your string to this Pascal format. The \p notation does this. In C, the above example would become:

```c
DrawString("\pThe Toolbox knows this is a string!");
```

Enough about the \p—back to drawing on the screen. If you don't specify where to draw a string, the string will start at the current location of
the graphics pen. To place a string at a specific window location, precede the DrawString() call with a MoveTo() call:

```
MoveTo(100, 50);
DrawString("\pHello, World!");
```

Figure 12-9 shows the result of the above code. Note that the string starts 100 pixels from the left edge of the window, not the screen. The same holds true for the starting vertical coordinate, 50.

To draw two lines of text, one below the other, you would use a MoveTo() call to move the graphics pen down, then use a second DrawString() call to draw your second line of text. If you are familiar with the ANSI C printf() function, the following Note might interest you.

**NOTE**

To move down a line, ANSI C uses the newline (\n) escape sequence:

```
printf("Writing in ANSI C\n");
```

Does Macintosh C use this as well? The answer is no. Recall that ANSI C is text-based. Macintosh C is graphics-based. To move down a line in Mac C, you move the graphics pen to a different pixel location. Because you can move the graphics pen in single-pixel increments, this method allows for much greater
precision in the placement of text. The following code would write two lines of text, the second line 15 pixels below the first:

```c
MoveTo(100, 50);
DrawString("\pEvery programming book must include:");
MoveTo(100, 65);
DrawString("\pHello, World!");
```

The result:
Every programming book must include: Hello, World!

---

**The Str255 Type**

You've seen how to use `DrawString()` by placing text between double quotes:

```c
DrawString("\pThis is a string of text.");
```

The `DrawString()` routine will also accept a variable of the Macintosh Str255 type. A variable of this type holds a single string of up to 255 characters and is especially useful for creating strings with `#define` statements and for initializing string variables:

```c
#define congrats_str "\pGood Job!"
Str255 test_str = "\Testing 1 2 3";
```

```c
DrawString(test_str);
DrawString(congrats_str);
```

Another important use for the Str255 type is in the definition of structure members. Records of people, whether students, employees, or any other group, are often maintained through the use of structures. Here, we create a simple structure that holds a student's name, ID number, and expected graduation year. Then we create an array of 300 of those structures:

```c
struct student_rec
{
    Str255 name;
    int id_num;
    int grad_year;
};
```

```c
struct student_rec all_students[300];
```
The Str255 is a very important type for getting information from a program user; it is used when working with dialog boxes. We'll offer several more examples of the use of the Str255 type in our example programs in Chapters 19 and 20.

**Drawing a Character**

You use the DrawString() function to draw lines of text. What if you want to draw just a single character? DrawString() will work for this purpose too:

```haskell
DrawString("\pX"); /* draw the letter X */
```

Although DrawString() works for one character, there is a Toolbox function—DrawChar()—which exists specifically for drawing one character. DrawString() works with a Str255; DrawChar() works with a single character. A single character occupies less memory, so DrawChar() is a more efficient manager of memory when drawing a single character.

Below are two ways of using DrawChar(). Note that while a string lies between double quotes, you always enclose a single character in single quotes:

```haskell
char letter; /* declare a char variable */
letter = 'Y'; /* set the variable equal to Y */
DrawChar('X'); /* draw the letter X */
DrawChar(letter); /* draw the letter Y */
```

**Text Characteristics**

In the previous section we mentioned that one of the advantages of treating text like graphics is that you gain the ability to alter its characteristics easily. The Toolbox provides three routines that allow you to do this—TextFont(), TextSize(), and TextFace(). We'll cover each in turn.

**Fonts**

Characters are letters of the alphabet, numbers, and symbols. Every character is defined by the pixels that make up its size and shape. A font contains the pixel definition for each character. Fonts reside in the System File of your Macintosh. Each font is identified by an integer value. Apple has reserved the numbers 0 through 127 for its own fonts. So you don't have to
memorize these identifying numbers, Apple has predefined some font constants, a few of which are listed below:

```c
#define chicago 0
#define newYork 2
#define geneva 3
#define monaco 4
#define venice 5
#define sanFran 8
#define cairo 11
#define losAngeles 12
#define times 20
```

You can use the Toolbox routine `TextFont()` to change the font that is used to display the text that you draw to a window. Pass the font identifier when using this routine. The following call would change the font to geneva:

```c
TextFont(geneva);
```

Because geneva is a constant defined as 3, the following call would have the same effect:

```c
TextFont(3);
```

Once you call `TextFont()`, all text drawn with future `DrawString()` calls will be affected. Text that was already on the screen before you called `TextFont()` will not be changed. Figure 12–10 shows the result of executing the following code:

```c
TextFont(geneva);
DrawString("\pGeneva Font ");
TextFont(chicago );
DrawString("\pChicago Font ");
TextFont(venice );
DrawString("\pVenice Font");
```

**Text Size**

You can alter the way a font looks on the screen by changing its size. The display size is given in *points*, a unit of measure used by professional typographers. There are approximately 72 points in an inch. You can use the Toolbox routine `TextSize()` to change the size of the current font. Pass the
point size when using this routine. The following call would change the current font to 12 points:

TextSize(12);

You may have noticed that programs like word processors that use Style menus offer only certain font sizes. The sizes 9, 10, 12, 14, 18, and 24 are typically available. Why are you limited to these sizes? The answer lies in what TextSize() does. It does not simply scale the current font, as you might have imagined. Instead, TextSize() uses the font size with the font identifier to select a font from the System File. The System File contains separate pixel definitions for geneva 9 point and geneva 12 point, for example. If you do attempt to access a font that doesn’t exist (such as geneva 15), the Toolbox will attempt to scale an existing geneva font but will have only limited success, so the result will be text that has a jaggie look.

**Text Face**

Another method of altering the way a font looks on the screen is to change the face, or style, of the current font. Apple has a C type just for this purpose. It’s called, not surprisingly, Style. This is a data type unique to Macintosh C—it is not a part of ANSI C. Apple has defined seven text face styles—bold, italic, underline, outline, shadow, condense, and extend.

To change the face of the current font, use the Toolbox routine TextFace(). This routine accepts a single parameter of type Style. The following would change to a bold font:

TextFace(bold);
FIGURE 12-11 Altering text characteristics—font, size, and face

You can combine as many or as few Styles as you wish by using the + sign between parameters. If you don't include any parameters, the text face will return to plain. The following code fragment is a brief example of changing the characteristics of some text. Figure 12–11 shows what the screen output would look like.

```c
TextFont(geneva);
TextSize(12);
TextFace(italic);
DrawString("\pItalic text.");
TextFace(bold + underline);
DrawString("\pBold and Underlined.");
TextFace( );
DrawString("\pBack to plain.");
```

**Centering a String**

You've seen how you can vary the look of text that you draw in a window. A potential drawback to this is that the length, the number of pixels across, that a string has will vary depending on the characteristics you give it. If you want to center a string in a window, this can present a problem. Consider the following code:

```c
TextFont(geneva);
TextSize(12);
TextFace( );
/* 12 point plain geneva text */
MoveTo(30, 40);
```
FIGURE 12-12  Text characteristics affect string length.

```c
DrawString("\pTesting 123");
TextSize(18);
TextFace(bold); /* 18 point bold geneva text */
MoveTo(30, 60);
DrawString("\pTesting 123");
```

This code draws the same string to a window twice. The graphics pen is what draws text. Because the characteristics of the graphics pen were changed before drawing the string for the second time, the pixel lengths of the strings differ. This is shown in Figure 12-12.

For situations like this the Toolbox provides a useful routine called StringWidth(). You pass StringWidth() a string, either directly between quotes or as a variable of type Str255. You do not have to specify the settings of the graphics pen. As a matter of fact, that's the power of this routine—even you do not have to have any idea what the current settings are. The code below demonstrates the procedure for centering a string in a window. We'll assume that we have created a window with a pixel width of 400:

```c
#define WINDOW_WIDTH 400
#define TEST_STR "\pTesting 123"

int string_pixel_length;
int horiz_location;

string_pixel_length = StringWidth(TEST_STR);
```
A window 400 pixels across was created using ResEdit.

StringWidth() determines that the "Testing 123" string is 100 pixels across.

To center the string, leave 150 pixels on either side of it:

\[
\frac{400 - 100}{2} = 150
\]

FIGURE 12–13 Centering text using StringWidth()

```
horiz_location = (WINDOW_WIDTH - string_pixel_length) / 2;
MoveTo (horiz_location, 40);
DrawString(TEST_STR);
```

Figure 12–13 further illustrates how the above code centers a string in a window.

**Graphics Ports**

So far all of our examples have used just one Macintosh window on a screen. But what if there were two windows, or more? Have you wondered just how QuickDraw knows which window it is supposed to draw to? We've discussed the coordinate system, but we've made no mention of how QuickDraw knows which window it should choose as its reference point when you issue a command like MoveTo(20, 40). The answer lies in a concept called *graphics ports*.

A graphics port defines where drawing is to take place. It holds graphical information about one drawing area. A graphics port defines the drawing environment of an area of the screen and the current "state of
affairs” of that area. The size, pattern, and location of the pen and the size and type of text to use when drawing are just a few of the items contained within the graphics port.

To hold all of this information, C defines a structure called a GrafPort. A GrafPort has more than two dozen fields. Keeping track of the name and purpose of each of those fields could be confusing. Fortunately for us, the C language allows these fields to be accessed and altered indirectly through Toolbox calls. The Toolbox routines know which fields to work with. For example, the GrafPort field that keeps track of the text size to display in a particular graphics port is the txSize field. When you want to change the size of the text QuickDraw displays, you use the Toolbox routine TextSize() rather than directly manipulating the value of a GrafPort’s txSize field. This makes the exact inner structure of a GrafPort less important to you.

Every window has a graphics port associated with it. When you want QuickDraw operations to apply to a particular window, you make that window’s graphics port active. To do this, you use the Toolbox routine SetPort(), passing a pointer to the graphics port associated with the window of interest. This pointer is of the Macintosh C data type GrafPtr. From that point on, all QuickDraw operations will take place within this window.

This section has only been an introduction to graphics ports. Later, we’ll discuss them in greater detail. We’ll let you know how to determine what GrafPort pointer is associated with what window, and just how and when to use the SetPort() routine.

By now you should be convinced that GrafPorts are handy things to have around. Figure 12-14 shows what Macintosh life would be like without graphics ports = disaster!
without them. Without a reference as to what window to draw to, Quick-
Draw would draw text and shapes across windows and across the screen
without regard for the boundaries of windows.

Graphics ports are an important concept in the Macintosh windowed
drawing environment, so we’ll have more to say about them when we dis-
cuss the concept of windows in Chapter 16.

**Chapter Summary**

This chapter introduced you to many of the QuickDraw graphic routines
that reside in the Toolbox. You can use a QuickDraw routine in your own
program by knowing the routine’s name and the arguments it requires.

You discovered the power of the graphics pen, which is responsible for
what you see on the screen, when we described Apple’s Macintosh C type
Pattern. Passing one of five predefined patterns to PenPat() causes all subse-
quent drawing to be done in this new pattern. The five patterns are white,
LtGray, gray, dkGray, and black.

In the last chapter you learned how to define a rectangle using Set-
Rect(), how to frame it with FrameRect(), and how to erase it using
EraseRect(). In this chapter you learned about other operations you can
perform on a rectangle. PaintRect() paints a predefined rectangle with the
current pen pattern. FillRect() also paints a rectangle, but uses a pen pattern
that is passed as a parameter. The InvertRect() routine inverts all of the
pixels in a given rectangle.

This chapter introduced you to the oval. The SetRect() routine is first
used to define a rectangle that will form the boundaries for an oval. If the
SetRect() call defines a square, then the oval that is drawn will be a circle.
Five routines—FrameOval(), PaintOval(), FillOval(), InvertOval(), and
EraseOval()—which are analogous to the similarly named rectangle rou-
tines, are devoted to the oval.

The last shape covered was the round rectangle. Again, use the Set-
Rect() routine to define a boundary rectangle. The five routines devoted to
the round rectangle are FrameRoundRect(), PaintRoundRect(), FillRoun-
dRect(), InvertRoundRect(), and EraseRoundRect().

You’ve learned not only how to draw text to the screen with Draw-
String() but also how to alter the font, size, and style of text using Text-
Font(), TextSize(), and TextFace(). You saw that Apple has created the
special data type Style to make it convenient to pass the TextFace() routine
one or more of the text styles—bold, italic, underline, outline, shadow, condense, or extend.

Finally, graphics ports were introduced. Graphics ports tell QuickDraw exactly where drawing operations should take place on the Macintosh screen. Each window on the screen has its own graphics port. A graphics port holds all of the graphical information associated with a window, such as the pattern of the pen and the size and font of text drawn in the window. This allows each window to have its own drawing environment.
Chapter 13

Resources and ResEdit

Throughout this book we have mentioned resources and their role in Macintosh programs. In this chapter we discuss them in greater depth to help you learn why resources are so important in creating a program with the Macintosh look and feel.

Apple encourages Macintosh developers like you to adhere to its graphical interface standards. That way users of Macintosh programs will never run into unexpected surprises when they run a program for the first time. So it was in Apple's best interest to create a tool to help developers create the interface portions of their programs. ResEdit is this tool. In this chapter we'll describe ResEdit in detail and provide plenty of examples of its use.

Code + Resources = Program

We've stated that QuickDraw has a part in everything you see in a Macintosh program. The same holds true for resources. Resources play a part in just about everything you see when you look at your Macintosh screen.
Menus, dialog boxes, and windows all make use of resources. Even the things that appear within these interface components, such as buttons, menu items, and scroll bars, use resources. Thus, their importance and the recommendation that you fully understand how to use them cannot be overstated.

To create a Macintosh program, you must do more than write your C source code. You must also create the resources that your program will use. This extra step may at first sound like more work for you. But when you use a graphical tool like ResEdit to create the resources, your program development time will actually be shorter than it would be if you had to create menus and dialog boxes within your code. This is because ResEdit allows you to see, right on your screen, exactly how menus, dialog boxes, and windows will appear in your program. ResEdit takes the guess work out of creating those parts of your program.

Resources exist on disk in a separate physical file from that of your source code. This resource file is a collection of individual resources. You must eventually join these two files to form your final program. The Macintosh C compiler that you use will probably perform two steps at once when you compile your source code. When you select your compiler's Compile or Run menu option, it first compiles your C language source code file into an object file. It then takes the resource file you created with ResEdit and links, or merges, it with the object file. The result of the link is an executable file—a Macintosh application. Figure 13-1 illustrates this compile/link process.

![Diagram of compile/link process](image)

**FIGURE 13-1** Source code and resources make a program.
Because you use resources to create the graphical components of your program, you can alter your program's appearance without modifying any source code. The items that appear in a menu are examples of this. You can change the item names without modifying any source code. You use a resource editor like ResEdit to simply rename a menu item.

You can use a resource editor to make an addition, such as a button in an existing dialog box, to a program you're creating, but your work doesn't stop there. You must also write source code so that your program will respond to a mouse-click in the button. Without the additional source code, the button will appear in your dialog box, but no action will be taken when the user clicks in it.

This chapter will show you how to use ResEdit to create such things as menus and buttons. Future chapters will show you how to write the source code to make these things usable. In this chapter you'll create a resource file that contains a WIND resource, which is a window template. In Chapter 16 you'll learn how to write Macintosh C code that takes this WIND resource from a resource file and loads it into memory.

Before moving on, let's quickly summarize the purpose of resources:

- **What resources do**: They describe the visual characteristics of your program.
- **What resources do not do**: They do not alter or define the way in which your program behaves.

### Introduction to ResEdit

In the last illustration, Figure 13–1, you may have noticed that the picture that represents the resource file resembles the picture that represents the object file. The object file and the resource file are both readable by a computer, but not by a human being. The C language source code is the form of the object file that is human-readable. But how do you read the resource file? You use a software tool that opens the resource file in a graphical form that is easy for us to understand. No surprises here—the tool is ResEdit. ResEdit allows you to view and edit the individual resources that make up a resource file.

ResEdit is distributed free of charge by Apple. It comes with the THINK C compiler. Another source for ResEdit is on-line information services. If you or someone you know subscribes to an information service, such as CompuServe or America Online, you can download a copy using a modem.
The next several pages will take you on a short tour of ResEdit. Our mission will be to do the following:

1. Create a new resource file.
2. Create a WIND window resource.
3. Modify some of the features of our new window resource.

Accomplishing these three tasks will give you good insight into the workings of ResEdit. There are three ways to learn about ResEdit:

- If you are following along in the Simulator software, the next few software screens will step you through the process.
- If you aren't running the Simulator software but have a copy of ResEdit, you can run ResEdit and follow the steps we present on the following pages.
- If you don't have your Mac running, just read the next few pages so that you know what to expect when you do actually run ResEdit.

You can get to the first task by running the ResEdit program. From your Macintosh desktop, double-click on the ResEdit icon. Figure 13–2 shows the dialog box that will appear if you’re using ResEdit version 2.1. Click anywhere in the dialog box to continue.

The next dialog box gives you the opportunity to select a resource file to edit by clicking the filename and then the Open button. Because this is
the first time you’re using ResEdit, you’ll want to click on the New button to create a new resource file.

When you click on the New button, another dialog will appear. This one allows you to enter a name for your resource file. Although some compilers allow you to give it any name, others require you to give it a name that associates the file with your program. In Figure 13–3, we have a C program source code file called MyProgram.c, so we’ve given our resource file the name MyProgram.rsrc. After typing in a name, click on the New button.

When you click on the New button, an empty window with your filename displayed at the top, as shown in Figure 13–4, opens. You have now completed the first of your three tasks. You have created and named a
resource file. Next, you will add some substance to it by adding a resource to the empty resource file.

Select Create New Resource from the Resource menu, as shown in Figure 13–5. This will bring up a dialog box, shown in Figure 13–6, that lists the different types of resources. We will describe some of these in just a moment. For now, scroll down to the WIND type and click once on it. Then click the OK button.

After clicking the OK button, you will see the window that is shown in Figure 13–7. This window allows you to modify the characteristics of the window resource template you are creating. At this point, you have accom-
phished your second task—you've created a WIND resource. Now you can modify some of the features of your window.

Figure 13-7 shows what ResEdit calls a MiniScreen. This gives you an idea of what your window will look like when your final program runs. As you make changes to your window with ResEdit, the MiniScreen window reflects the changes. To see how it works, double-click the mouse in the edit box that follows the word Width. Type in the number 450. What happens as you type? The MiniScreen window widens, as shown in Figure 13-8.

Another feature you can change is the type of window your WIND resource represents. Note the row of icons across the top of Figure 13-9. Clicking on any one of those changes the type of the window. In Figure 13-9 we've clicked on the fifth icon from the left. The window in the MiniScreen has changed to match this type.

We have finished experimenting for now, so click on the first window type icon so that your window again looks like the one shown in Figure 13-8.

Now that you've changed the window's size and learned how to change its type, you've accomplished your third and final task. Click in the close box, as shown in Figure 13-10.
FIGURE 13-8  Altering the size of the window

FIGURE 13-9  Altering the type of the window
After you close the window, your screen should look similar to Figure 13–11. Your WIND has an ID of 128, the number ResEdit gave it. ResEdit allows you to change this number, but for now leave it as is. When you write C code to go with this window, you’ll use this WIND ID. Whenever you want your program to put a window on the screen, you’ll tell it to look in your resource file for a WIND with an ID of 128. We’ll discuss this process in detail in Chapter 16.

In Figure 13–11 there are two windows open. One of them lists all of the WIND resources in your resource file. So far, you have just one, WIND
128. If you had more, they would be listed in this window. Click in the close box of this window now. The window that remains open on your screen has one icon, labeled WIND, in it. Double-clicking on this icon reopens the list of WINDs that you just closed. Later, when you create other types of resources, more icons will appear in this window.

Now that your WIND resource is complete, select Save from the File menu. If you have finished experimenting with ResEdit, select Quit from the File menu to quit ResEdit.

There are other features of the WIND that we could have altered. And there are numerous other types of resources you can create. We'll go into more detail in future chapters as we need more resources.

## Types of Resources

We've briefly examined the WIND resource type. There are dozens of other types as well. In this book, we'll cover only about a half dozen. These will be sufficient to give you a solid understanding of resources and to get a full-featured Macintosh program running. Other types we'll be covering in future chapters include the following:

- **MENU** for creating menus
- **DLOG** for creating dialog boxes
- **DITL** for adding dialog items (such as buttons) to dialog boxes created using **DLOG**

## Chapter Summary

Like QuickDraw, resources play a large part in everything the Macintosh does. Resources make menus, windows, and dialog boxes possible.

Resources and C language code work together. Source code uses resources to load predefined menu, window, and dialog box templates into memory for use by your program.

ResEdit is the name of a program released by Apple that allows developers to create and edit resources. It lets the programmer view resources in a graphical way, making it easy to edit resources.
Resources are created by first making a resource file using ResEdit. ResEdit contains a Resource menu option that allows you to create new resources in your resource file.

The WIND resource is just one of several dozen resource types. ResEdit gives you the ability to define what a window will look like—its size, location on the screen, and window type. After you create and edit a WIND resource, you normally write in your source code file the C code that loads the window into memory.
One of the features that have made the Macintosh computer so popular is its user-friendliness. The Macintosh empowers the user to easily control what action will take place and when it will take place. The menus, buttons, and scroll bars on the screen give the user the option of choosing what to do next. Invisible to the user—but very important to the Macintosh and to you, the programmer—is the mechanism that locks on to these user actions. This mechanism is called an *event*. An event is a description of the user-action, such as a click of the mouse.

This chapter will show you how to capture events. Capturing them is not enough, though. You want to be able to process events—you want to make the flow of your program dependent on the events that occur. This chapter will show you just how to do this.

---

**Events**

Way back in Chapter 2, we said that Macintosh programs were event-driven. There, we provided a figure that we repeat here as Figure 14–1.
When a Macintosh program is running, the user is not aware that the actions shown in Figure 14–1 are taking place. The user never realizes that the program is continuously cycling, waiting for him or her to take some action—to cause an event to occur. Once an event occurs, the program grabs that event and processes it.

What constitutes an event? The following are some, but not all, of the events the Macintosh is capable of responding to:

- Mouse events occur when the user presses the mouse button (mouse down event) or releases an already pressed button (mouse up event).
- Key events occur when the user presses (key down event) or releases a key (key up event) on the computer keyboard.
• Disk events occur when a disk is inserted into the Macintosh disk drive.
• Activate events occur when an inactive window becomes active, or vice versa.
• Update events occur when the contents of a window must be drawn. This occurs when an obscured window comes to the forefront or when a window partially off the screen is dragged back on screen.
• Null events occur when you ask for an event and there is none to report.

How does a program process an event? As the programmer, that is up to you. You have the power to ignore certain events and respond to others. If a program you write makes no use of files, you may choose to ignore the fact that a program user inserts a disk into the disk drive. On the other hand, if your program is capable of opening other files, as a word processor does, you'll want to be aware of disk-insertion events.

To see how a very simple event-driven program works, we're going to take a look at a simple program we've named Event. Also, if you have THINK C, you'll have an opportunity to practice creating a project and adding files to it.

The Simplest of Events

As far as Macintosh programs go, our Event program is as simple as they get. It consists of a resource file with just one resource and a source code file with fewer than 20 lines of code. The Event program offers few of the features of a true Macintosh program. But it serves to demonstrate the importance of events in Macintosh programming.

The Resource File

The software that comes with this book contains a folder called THINK C Examples. In this folder you'll find other folders, one of which contains the source code, resource file, project file, and ready-to-run application for a Macintosh program called Event. The folder is called Event Folder.

You can run the Event program that's in the Event Folder to see just what you'll be creating in this chapter. If you have THINK C, we recommend that you create your own Event folder, source code, project file, and resource file. Then you can compare your files to the versions we've supplied. If you don't have THINK C, don't worry. This chapter and the Simulator will show you just how these files would be created and how the Event program operates. Then you'll be ready when you do get a compiler.
If you're using THINK C, you might have a folder setup similar to that shown in Figure 14–2. We have the Development folder on our hard drive. Begin by creating a folder called MyPrograms Folder in your THINK C Development Folder. Then, in the MyPrograms Folder create another folder. Call this one MyEvent Folder. You'll keep your THINK C project, source code, and resource file in this folder when you create them.

Begin by creating the resource file. Run ResEdit by double-clicking on its icon. Follow the four steps below to create a resource file:

1. Click on the Introductory Dialog.
2. Make sure you're in the MyEvent Folder you just created.
3. Click on the New button, as shown in Figure 14–3.
4. Type in the name Event.pi.rsrc and click on the New button. Figure 14–4 illustrates this. The π symbol is made by pressing and holding the Option key while you type the letter p. The name you give the resource file is important. We'll offer an explanation in a few pages.

Your new resource file starts out empty. All you have to do is add a WIND resource. Create your WIND resource by following these steps:

2. Choose the WIND type, then click the OK button.

A window will open. For this simple program you can leave all the settings as they are. Save this resource file and exit ResEdit.
FIGURE 14-3 Creating the Event.rsrc resource file using ResEdit

FIGURE 14-4 Naming the Event.rsrc resource file using ResEdit
The THINK C Project

If you have THINK C, follow the steps in this section to create a THINK C project. First, run THINK C by double-clicking on the THINK Project Manager icon. You’ll be presented with a dialog box. Change folders until you are in the MyEvent Folder that you created in the last section. You want to create a new project, so click on the New button. Another dialog box will open. Type in the name of your project, Event.π.

Traditionally, a THINK C project has the π symbol after its name so that you know it’s a project. You don’t have to include the π symbol in your project names, but we recommend it. One thing you must do, however, is give a program’s resource file the exact same name as the project, with the extension .rsrc added. Recall that we asked you to name the resource file Event.π.rsrc, which is the project name (Event.π) with the correct extension (.rsrc) added.

Next, click on the Create button. Again, be sure you are in the MyEvent Folder.

You’ll be presented with the Event.π project window. Because it’s new, there are no files listed in the project. Select New from the File menu. Then type in the source code. The entire listing for Event.c is shown in Listing 14–1.

LISTING 14–1 Event program source code listing

```c
main()
{
    WindowPtr the_window;

    InitGraf(&thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(0L);
    FlushEvents(every_event, 0);
    InitCursor();

    the_window = GetNewWindow(128, 0L, (WindowPtr)-1L);

    SetPort(the_window);
}```
MoveTo(20, 30);

DrawString("\pClick the mouse.");

while (!Button())
{
}

When you're done, select Save As from the File menu. Name the file Event.c. You've created a source code file, but the Event.rt project isn't aware of it yet. You must now add it to the project. Do so by selecting Add from the Source menu.

You will see a dialog box listing all of the files in the current folder. You're in the MyEvent Folder, which has just one C file in it—the Event.c file you just created. Click once on the file name to highlight it, then click on the Add button.

The Event program will use Macintosh C types, which are already defined in a file called MacTraps, which is included with your THINK C compiler. MacTraps is in the Mac Libraries folder. Change to that folder now. Click on MacTraps, then click on the Add button.

IMPORTANT

Every program you write for the Macintosh will use the MacTraps file. Add it to all of your projects.

MacTraps and Event.c are the two files you need to run the Event.rt project. Click on the Done button to add them. To see the source code for this program, double-click on the file titled Event.c in the Event.rt project window. The source code file will open on the screen, as it has in Figure 14–5.

You'll be able to examine the code in a moment. But first, let's get the project ready to run. Choose Bring Up To Date from the Project menu of THINK C. This will tell THINK C to compile your program. If THINK C displays an error window, like the one shown in Figure 14–6, you'll have to make a slight change to the THINK C environment.

THINK C projects should include a header file called MacHeaders. A header file contains #define statements and structure definitions—things the compiler uses before compiling the rest of your source code. THINK C
FIGURE 14-5 Opening the source code file from the Event.π project window

FIGURE 14-6 THINK C alerts you to an error.

has the ability to compile, use, and reuse a header file in any project. MacHeaders is just such a file.

The error message shown in Figure 14-6 results when MacHeaders is not included in a project. You can be sure that MacHeaders is in your project by selecting Options from the Edit menu of THINK C. In the upper-left corner of the dialog box that appears is a pop-up menu. Click on it and select the Prefix menu choice. Figure 14-7 shows this dialog box and the line of text that is responsible for including the MacHeaders header file
in your project. You can select this dialog box in either THINK C 5.0 or THINK C 6.0.

If you're in THINK C, run the program by selecting Run from the Project menu. If you're using the Simulator, you can click on the GO button on the page that tells you to do so. This will run the Event program from within the Simulator.

Running the Event program will display on the screen a window with the words “Click the mouse.” written in it. This is illustrated in Figure 14–8. Click the mouse to end the program. Now that you've seen the Event program running, you can look at the source code you used to create the program.

The Source Code File

The source code used for the Event program is repeated in Listing 14–2. This time, to improve clarity, we've added several comments to the source code. There is just one function, main(), and one variable, a WindowPtr named the_window. The first eight lines of code are all calls to initialization routines—you must include these calls in every Macintosh program you write. You'll see them again, grouped together in a function, in the second program that appears in this chapter.
The first line of code of interest is the line that brings a window into memory and onto the Mac's screen:

```
the_window = GetNewWindow(128, 0L, (WindowPtr)-1L);
```

This line calls up WIND 128, the window resource you created earlier in this chapter. When you created WIND 128, the Initially Visible check box was checked. That causes the window to be displayed when GetNewWindow() is called.

Next, the code sets the port to the newly created window. This is a necessary step that ensures that the actions taken by subsequent calls to QuickDraw drawing routines will take place in the window. Recall from Chapter 12 that a graphics port holds information about the drawing area of a window. We'll cover graphics ports thoroughly in Chapter 16.

After setting the port, the graphics pen is moved by calling MoveTo(). Then a short string is drawn to the window with DrawString().

Finally, a while loop is entered. The body of the while loop consists of just a semicolon. No action is taken in the loop. The loop simply repeats over and over until you press the mouse button. The Toolbox routine Button() checks the state of the mouse button to see if it is pressed. The line

```
while (!Button())
```
reads “while not button, perform the loop.” Thus, until you press the mouse button, the loop will continue to be repeated. The body of the loop consists of nothing more than a semicolon—a do-nothing line. Once the mouse button is pressed, the test fails, and the loop and the program end.

**LISTING 14-2**  
Event program source code

```c
main()
{
    WindowPtr the_window; /* pointer to a window */
    InitGraf(&thePort);  /* initialization routines */
    InitFonts();        /* */
    InitWindows();      /* */
    InitMenus();        /* */
    TIEinit();          /* */
    InitDialogs(OL);    /* */
    FlushEvents(every_event, 0); /* */
    InitCursor();       /* v */
    the_window = GetNewWindow(128, OL, (WindowPtr)-1L);
    /* bring resource WIND 128 */
    /* into memory */
    SetPort(the_window); /* set port to our window */
    MoveTo(20, 30);      /* move the pen*/
    DrawString("\pClick the mouse.");   /* draw to window */
    while (!Button())    /* while the button NOT clicked... */
    {                    /* ...do nothing */
        ;
    }
}
```

**But Where Are the Events and the Event Loop?**

We'll be the first to admit that the Event program isn't much of a program at all. There is no menu bar, and the window that comes to the screen cannot be moved. Worst of all, a single click of the mouse button ends the program. We can overlook all of this if there is a lesson to be learned. And we claim there is—how events work. So where are the events and the event loop? Our Event program acknowledges just one kind of event—a mouse
event. And the mighty event loop we've been speaking so highly of appears here as just two lines of code:

```c
while (!Button())
```

Let's see if the Event program shares the structure we claimed all Macintosh event-driven programs should have. We repeat Figure 14–1 on the next page, this time with the Event code superimposed onto it. Look closely at Figure 14–9.

**Start Program**

The main() function is always the start of any C language program. The opening brace of main() signals the start of the action.

**Initialization**

The program has plenty of initialization code. Eight routines are called to perform the required Toolbox initializations.

**Retrieve Event**

When the Toolbox routine Button() is called, a request is being made to the Toolbox. Button() asks the Toolbox to examine the state of the mouse to retrieve the status of the mouse button. Function Button() returns an event to the program, which now knows whether the button is pressed (TRUE) or not (FALSE). So the Button() part of the while loop serves to retrieve an event.

**Process Event**

Including the ! in the while loop has the effect of processing the event. An event is retrieved by calling Button(). Processing of this event simply involves checking the state of the button. Button() retrieves an event, the ! processes it by saying, “if the button is not down, enter the loop for further action.” If the button *is* down, the type of event is determined; in this case, it is a mouse down event. If the button is *not* down, the body of the while loop is performed. Further processing of events, whatever their kind, could occur within the body of the loop. This event-processing loop simply has a semicolon—a do-nothing statement. The usefulness of the program could be greatly increased by including more statements in the loop body.
Repeat

A program would not be much good if it retrieved and processed only one event. So the Event program uses a while loop to retrieve and process the state of the mouse button repeatedly.
**End Program**

The closing brace of the main() function ends the program. Your Macintosh is smart enough to know it's time to wrap it up and return to where it started—the THINK C environment, the Simulator, or the Finder—the Macintosh Desktop.

As simple as it is, the Event program has the structure of an event-driven program. But with events, as with most programming concepts, things are never quite as simple as they first appear. The Event program looked for, and could deal with, only one type of event—the mouse down event. Any program worthy of being called a Macintosh program must be capable of much more than this. But the mighty Event program served its purpose by providing a basic understanding of events and event-driven programming. Now you're ready to study further how the Macintosh deals with events. Then, it will be time to expand the main event loop into something more useful and powerful.

**Event Records**

Events are at the very heart of the Macintosh. Whenever an event takes place, your computer knows about it—long before your program is aware of it. The hardware passes event information on to system software where it is saved for future reference. The information passed is a complete description of the event that occurred. It includes such information as the type of action—a mouse click, for example—and associated information, such as where on the screen the cursor was at the time of the mouse click.

Because your program might not respond to events right away, the Macintosh stores events in a holding area—an event queue. Several events can be stored, awaiting processing by your program. Your program, through Toolbox function calls, can retrieve those events from the queue and process them as it sees fit.

The information about a single event is contained in an event record. To make accessing these events easier, Apple has defined the C structure EventRecord:

```c
struct EventRecord
{
    short what;       /* type of event */
    long message;     /* additional information */
    long when;        /* time the event occurred*/
};
```
Of most importance to you is the what field. This field tells you what
type of event occurred. You can’t begin to deal with an event until you
know what type of event it is. Macintosh C has predefined constants that
you can use in determining the event type:

```c
#define nullEvent 0
#define mouseDown 1
#define mouseUp 2
#define keyDown 3
#define keyUp 4
#define autoKey 5
#define updateEvt 6
#define diskEvt 7
#define activateEvt 8
#define everyEvent -1
```

We’ll describe some of these event types later in this chapter as we use
them in our example code. Table 14–1 summarizes the above event types.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nullEvent</td>
<td>No event</td>
</tr>
<tr>
<td>mouseDown</td>
<td>Mouse button clicked</td>
</tr>
<tr>
<td>mouseUp</td>
<td>Mouse button released</td>
</tr>
<tr>
<td>keyDown</td>
<td>Key pressed down</td>
</tr>
<tr>
<td>keyUp</td>
<td>Key released</td>
</tr>
<tr>
<td>autoKey</td>
<td>Key pressed and held down</td>
</tr>
<tr>
<td>updateEvt</td>
<td>Update (redraw) a window</td>
</tr>
<tr>
<td>diskEvt</td>
<td>Disk was inserted</td>
</tr>
<tr>
<td>activateEvt</td>
<td>Window was activated/deactivated</td>
</tr>
<tr>
<td>everyEvent</td>
<td>Event of any kind occurred</td>
</tr>
</tbody>
</table>

**TABLE 14-1** Event types and their meanings
IMPORTANT

We've said it before, and we'll say it again—remember, C is case-sensitive. When you use the event type constants, use upper- and lowercase just as we've done here.

After the what field, the other fields have more or less importance depending on the type of event. For example, if an event's what field tells you that the event is a mouseDown event, then the where field becomes important. It tells where on the screen the cursor was situated when the mouse button was pressed. If you had a dialog box open on the screen, you'd want to know if the cursor was over a dialog button or check box when the mouse button was pressed.

Retrieving an Event

You must use a Toolbox routine to determine the type of event that has occurred. You pass to the Toolbox routine a pointer to a variable of EventRecord type and let the Toolbox fill the fields of this event record variable. Within your program, you can declare a variable of EventRecord type:

```c
EventRecord The_Event;
```

We chose to make it a global variable so its scope encompasses the entire source code. The Toolbox routine you use to retrieve an event is GetNextEvent(). Here is an example of a call to GetNextEvent():

```c
GetNextEvent(everyEvent, &The_Event);
```

From the above you can see that a call to the GetNextEvent() function requires two parameters:

- **Event mask** The first parameter is an integer used as a *mask*. The purpose of a mask is to hide, or mask, certain things. In the case of GetNextEvent(), you might want to mask some of the many event types we listed above. The event mask lets you tell GetNextEvent() what types of events you're interested in.

- **Event record** The second parameter is the address of an event record. You pass to GetNextEvent() the address of a variable of EventRecord type, and GetNextEvent() will fill the fields of the variable with information about the event.
As with event types, the Macintosh has a set of constants defined for event masks. The following are some of the event masks available:

```c
#define mDownMask 0x2
#define mUpMask 0x4
#define keyDownMask 0x8
#define keyUpMask 0x10
#define updateMask 0x40
#define diskMask 0x80
#define everyEvent 0xFFFF
```

Apple makes the suggestion, and we strongly support it, that you pass the constant everyEvent as the event mask. This signals the Toolbox to be on the look out for events of all types. It's a way of playing it safe. Let the Toolbox report an event of any type to you, then you be the one to decide which events you want your program to handle.

GetNextEvent() is an important Toolbox function. We'll use it in the upcoming version of our event loop.

### Expanding the Main Event Loop

The Event program that you ran at the start of this chapter used a while loop as the basis of its main event loop. For simplicity, the program checked only to see if the mouse button was pressed by using the Button() routine. Here, in its entirety, is the main event loop of the Event program:

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

In this section, the event loop will be expanded to make it applicable to any program and any type of event. We'll make several passes, each time filling in details so we eventually end up with a Macintosh C fragment that can be used in any program.

### Improved Event Loop Pass 1: Pseudo Code

In Chapter 2 we first made mention of pseudo code. There we said that pseudo code is the combination of a programming language with informal English statements. Its purpose is to convey a general programming idea without going deeply into the specifics of the programming language
1. Get the most recent event.

2. Check to see what the event was related to
   (Was a click of the mouse involved? A press on a key?)

3. Call another function that will handle the particular event.

4. Repeat Steps 1 through 3 until done.

FIGURE 14–10  Steps in an event loop

needed to execute this idea. The event loop is a perfect candidate for pseudo code. Figure 14–10 summarizes the four main steps an event loop should perform.

Below we turn the four points from Figure 14–10 into pseudo code:

while (Not Done With The Program)
{
    Get The Next Event
    switch The Event
    {
        case Event Type 1:
            Call Routine To Handle Event Type 1
            break

        case Event Type 2:
            Call Routine To Handle Event Type 2
            break
        etc.
    }
}

Let's take a closer look at the pseudo code event loop. Figure 14–10 shows that the first step in creating an event loop is to get the most recent event. The pseudo code satisfies that requirement with:
Get The Next Event

Notice that no details regarding how the event will be retrieved were provided. Pseudo code does not have to include details.

The second step is to check to see what the event is related to. Here, a C language switch statement suffices. The intent is to examine the most recent event to see what kind it is. Then, based on the type of event, a separate routine is called from within the body of the switch. For now, the events have been generically called Type 1, Type 2, etc. The body of the switch statement satisfies the third requirement of calling a separate routine to handle the particular event encountered.

The fourth and final step is to repeat the previous three steps. This is accomplished by nesting the pseudo code in a while statement:

```c
while ( Not Done With The Program )
{
    /* the rest of our pseudo code in here */
}
```

Like the Event program, this version of an event loop also uses a while loop. But instead of checking for a press of the mouse button in the while loop, we use the more generic phrase “Not Done With The Program.” And, the body of the loop has grown. We retrieve the event from within the while loop and then do the processing of the event using a switch statement. We look for two as yet unnamed types of events. By using a switch statement, we allow our program to handle any number of event types by adding more cases.

**Improved Event Loop Pass 2: Adding GetNextEvent()**

Although our event loop is in pseudo code, we’ve mixed a little real C code in with it so you can visualize how the event loop might actually work. We’re ready to take it a little farther by substituting some more real C language code for the pseudo code:

```c
Boolean All_Done = FALSE
EventRecord The_Event

while (All_Done == FALSE)
{
    GetNextEvent(everyEvent, &The_Event)
}
switch (The_Event.what)
{
    case mouseDown:
        Call Handle Mouse Down Routine
        break

    case updateEvt:
        Call Handle Update Event Routine
        break
}

The event loop is shifting from pseudo code to C language code. We’ve introduced a variable called All.Done in the while test expression and a call to the Toolbox routine GetNextEvent(). Within the switch statement, we’ve changed the two case labels to actual Macintosh C constants: mouseDown and updateEvt. These are two event types that almost every Mac program looks for and responds to.

- **mouseDown**  Events of type mouseDown occur whenever the mouse button is pressed. The mouse button does not have to be released for the event to occur.
- **updateEvt**  This type of event occurs whenever a window must be redrawn, as in the case of a background window coming to the front after the mouse is clicked on it.

We now have a more specific purpose for each case—to call a routine that handles properly the particular event that occurred. Each program you write will handle events in a different way, so it will be up to you to write a function for each event you are interested in.

**Improved Event Loop Pass 3:**

**Macintosh C Event Loop**

Now, the transition from pseudo code to C code is about to be completed. Below we present a functioning event loop written in Macintosh C:

```c
Boolean All.Done = FALSE;
EventRecord The_Event;
```
while (All_Done == FALSE)
{
    GetNextEvent(everyEvent, &The_Event);

    switch (The_Event.what)
    {
    case mouseDown:
        Handle_Mouse_Down();
        break;

    case updateEvt:
        Handle_Update_Event();
        break;
    }
}

Here we've made the lines immediately below the two case statements function calls. To complete things, we've added semicolons in the appropriate places.

Our event loop is not meant to be a complete program. It is intended to be included as a part of a program. One thing we've omitted is an assignment statement that sets All.Done to TRUE. Without this statement, our event loop would never end, nor would any program of which it was a part. Typically, a program has a menu option called Quit. When this option is selected, the All.Done variable is set to TRUE.

**Improved Event Loop Pass 4: MultiFinder Aware**

Many Macintosh users run MultiFinder on their computers. MultiFinder allows a Macintosh user to have more than one program running at a time. Before the advent of MultiFinder, events could always be retrieved using a call to GetNextEvent(), as we've demonstrated. With MultiFinder, you'll want to use a different Toolbox call—WaitNextEvent(). This routine handles things like background processing—the ability of your Mac to juggle two or more programs at once. Here is a typical call to WaitNextEvent():

```c
#define SLEEP_TICKS OL
#define MOUSE_REGION OL

WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION);
```
The \texttt{WaitNextEvent()} routine starts exactly like \texttt{GetNextEvent()}; they both have the same first two parameters. However, \texttt{WaitNextEvent()} goes on to add two more parameters—we've added two \#defines for a little clarity. \texttt{SLEEP\_TICKS} is used for background processing and \texttt{MOUSE\_REGION} is used for cursor tracking. Our programs won't make use of these features, so we won't discuss them in any more detail. For our purposes, and probably for yours as well, they can both be 0L. A 0 followed by an \textit{L} tells the compiler to store the 0 as a long integer type. \texttt{WaitNextEvent()} is expecting the last two parameters to be longs.

We've just stated that there are two routines that do essentially the same thing—retrieve an event. Your code should provide for one of the two routines. To do this, check to see if \texttt{MultiFinder} is currently active. Declare a global Boolean variable, \texttt{MultiFinder\_Active}, and call a Toolbox routine that will set \texttt{MultiFinder\_Active} to \texttt{TRUE} if \texttt{MultiFinder} is running, \texttt{FALSE} if it's not. This is a little tricky—you might want to just take our word for it when we tell you that what we're about to do is a valid check. If you want more detail, read the \textbf{Note} that follows the code:

\begin{verbatim}
#define WNE\_TRAP\_NUM 0x60
#define UNIMPL\_TRAP\_NUM 0x9F

Boolean MultiFinder\_Active;

MultiFinder\_Active = (NGetTrapAddress(WNE\_TRAP\_NUM, ToolTrap) !=
NGetTrapAddress(UNIMPL\_TRAP\_NUM, ToolTrap));
\end{verbatim}

\textbf{NOTE}

When you use the Toolbox routine \texttt{NGetTrapAddress()}, you are looking directly into the system to see if a certain Toolbox routine, \texttt{WaitNextEvent()}, has been loaded into memory. If it's there, you know the system has MultiFinder.

You may be wondering why we expend effort determining which of the two event-retrieving calls to use—\texttt{GetNextEvent()} or \texttt{WaitNextEvent()}. Why not just always use one or the other? As Mac programmers, we're obligated to support as many Mac users as we can. Owners of older model Macintoshses could not use our programs if they called \texttt{WaitNextEvent()}—their computers don't support this Toolbox routine. And we want to please
owners of new Macs by using a Toolbox call that supports the extended features that have been added to new Macintosh computers.

We’ve covered a lot of material in this chapter. Now, it’s time to tie all of the event-related concepts together and unveil our final product—an event loop that you can use and reuse in just about any program you write. We’ve bundled the body of our while event loop into a function of its own. Our while body now consists of just one line—a call to a function we’ve named Handle_One_Event(). Look over the code, then read the text that follows for a full explanation:

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
#define WNE_TRAP_NUM 0x60
#define UNIMPL_TRAP_NUM 0x9F
#define SLEEP_TICKS 0L
#define MOUSE_REGION 0L
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

Boolean All_Done;
Boolean Multifinder_Active;
EventRecord The_Event;

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
Main_Event_Loop()
{
    All_Done = FALSE;
    Multifinder_Active = (NGetTrapAddress(WNE_TRAP_NUM, ToolTrap)
    != NGetTrapAddress(UNIMPL_TRAP_NUM, ToolTrap));

    while (All_Done == FALSE)
        Handle_One_Event();
}
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
Handle_One_Event()
{
    if (Multifinder_Active == TRUE)
        WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS, MOUSE_REGION);
    else
    {
        SystemTask();
        GetNextEvent(everyEvent, &The_Event);
    }

    switch (The_Event.what)
    {
    case mouseDown:
        Handle_Mouse_Down();
        break;

    case updateEvt:
        Handle_Update_Event();
        break;
    }
}

We've given our main event loop the very appropriate name Main_Event_Loop(). Here we first set All_DONE to FALSE. It will be up to the program in which our main event loop appears to change the value of All_DONE to end the program. Next, we determine if the user's machine has MultiFinder and set our global variable Multifinder_Active based on the result. This first of our two functions, Main_Event_Loop() will be called only once during the running of a program, so these two assignment statements will be performed only once.

Next comes the main event loop. Our now-familiar while statement is the driving force. We've gathered the code that was formerly the body of the while loop and made it its own function—Handle_One_Event(). Once we reach the while statement, we'll repeatedly call Handle_One_Event() until the program ends.

In Handle_One_Event() we examine variable Multifinder_Active to determine which of the event-retrieving routines should be called. We've made one addition here. If MultiFinder is not present, we call GetNextEvent() as expected. But we also make a call to a Toolbox routine called
System Task(). This function is for system windows—namely, desk accessories. System Task() will give any open desk accessory some of the processor time so that it can perform updates of its own, if necessary. That is a task delegated to the Macintosh operating system.

The remainder of Handle_One_Event() is the switch statement that we developed earlier. Our event loop handles only two types of events. To handle other event types, simply add more cases to the switch.

**Event2: A Real Event Loop**

You started this chapter by running a simple program called Event. Now, you're ready for a more sophisticated example—Event2. In the THINK C Examples folder is a folder called Event2 Folder. In it we've included everything you need to run this program using THINK C—the resource file, the source code, and the THINK C project. Copy this folder into the Development Folder on your hard disk.

If you're using THINK C, double-click on the Event2.pi project, then select Run from the Project menu to run the Event2 program. Otherwise, use the Simulator program to run Event2. When you are running Event2, your screen should look like Figure 14–11.

![FIGURE 14-11 Result of running the Event2 demo](image-url)
The program demonstrates the handling of two types of events—mouseDown and keyDown. The Event2 program responds to either of these events by beeping the Macintosh's built-in speaker. Below is the source code, in its entirety, for the Event2 program. Most of it should look familiar to you, so we'll discuss only the new code as we come to it.

The listing starts with a number of #define statements. You've encountered most of them at one time or another in past listings. The first five were described in this chapter. NIL_PTR and IN_FRONT are used when the window is placed on the screen. We'll discuss them in Chapter 16. REMOVE_EVENTS is used in the initialization of the Toolbox:

```c
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
#define WIND_ID 128
#define WNE_TRAP_NUM 0x60
#define UNIMPL_TRAP_NUM 0x9F
#define SLEEP TICKS 0L
#define MOUSE_REGION 0L
#define NIL_PTR 0L
#define IN_FRONT (WindowPtr)-1L
#define REMOVE_EVENTS 0
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
```

Next, we list our global variables. You've had exposure to all of them in this chapter:

```c
Boolean All_Done;
Boolean Multifinder_Active;
WindowPtr The_Window;
EventRecord The_Event;
```

The first function is the only function required by all C programs and the starting point of all C programs—main(). The main() function simply calls three of our own functions:

- Initialize_Toolbox() is called near the beginning of the program to ensure that everything will run smoothly and system crashes will be avoided.
- Put_Up_Window() loads your WIND resource and displays the window on the screen.
Main_Event_Loop() is the main event loop, exactly as we designed it in this chapter.

```c
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
main()
{
    Initialize_Toolbox();
    Put_Up_Window();
    Main_Event_Loop();
}

The Initialize_Toolbox() routine will become very familiar to you as you program. Copy it and paste it into your source code file every time you write a new program:

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Initialize_Toolbox()
{
    InitGraf(&thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEinit();
    InitDialogs(NIL_PTR);
    FlushEvents(everyEvent, REMOVE_EVENTS);
    InitCursor();
}

Put_Up_Window() loads the window using a call to the Toolbox routine GetNewWindow(). It then makes sure the window is visible by calling ShowWindow(). A call to SetPort() sets the port to the loaded window. Several MoveTo() and DrawString() calls are then used to write instructions for the user:

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Put_Up_Window()
{
    The_Window = GetNewWindow(WIND_ID, NIL_PTR, IN_FRONT);
```
ShowWindow(The_Window);
SetPort(The_Window);

MoveTo(20, 30);
DrawString("\pClick mouse for two beeps.");
MoveTo(20, 45);
DrawString("\pType a key for one beep.");
MoveTo(20, 60);
DrawString("\pType 'q' key to Quit.");
}

The main event loop follows. This is the same code that appears in the Improved Event Loop Pass 4: MultiFinder Aware section:

/***************************************************************************/

Main_Event_Loop()
{
    All.Done = FALSE;

    Multifinder_Active = (NGetTrapAddress(WNE_TRAP_NUM, ToolTrap)
                      != NGetTrapAddress(UNIMPL_TRAP_NUM, ToolTrap));

    while (All.Done == FALSE)
    {
        Handle_One_Event();
    }
}

The main event loop calls Handle_One_Event() repeatedly. Here, Handle_One_Event() is almost identical to the one created earlier in this chapter. The only difference is in the types of events handled. The previous example handled mouseDown and updateEvt types. Here the event loop handles mouseDown and keyDown types.

The switch is where you can add the ability to handle other event types. To handle other types of events, simply change or add case labels in the switch. Then create appropriate functions to handle the events as you see fit:

/***************************************************************************/

Handle_One_Event()
{

if (Multifinder_Active == TRUE)  
    WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS,  

    MOUSE_REGION);
else
{
    SystemTask();
    GetNextEvent(everyEvent, &The_Event);
}

switch (The_Event.what)
{
    case mouseDown:
        Handle_Mouse_Down();
        break;

    case keyDown:
        Handle_Key_Down();
        break;
}

Each case label in the switch of Handle_One_Event() calls one function. If the event type is a mouseDown type, Handle_Mouse_Down() is called. This function actually carries out the handling of the mouseDown event. We chose to beep the Macintosh speaker twice to let you know the event was retrieved and this function was called. Hopefully, you’ll handle events in more interesting ways. The ShapeMaker program in Chapters 18 and 19 shows you how to respond to mouse clicks in dialog check boxes and how to pull down menus from the main menu bar.

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

Handle_Mouse_Down()
{
    SysBeep(5);
    SysBeep(5);
}

A keyDown event is handled by the Handle_Key_Down() function. Here we show you how to determine what key the user has pressed, based on the event record The_Event. Remember, when an event occurs, we cap-
ture its attributes in the global EventRecord variable The_Event. In the case of a keyDown event, the message field of The_Event holds the value of the key that was pressed. But it is not presented to us in the form we need. The event holds the keyboard location of the key that was typed, not the character value. We use the Toolbox routine BitAnd() and the Apple constant charCodeMask to extract this information from the message field. We then assign it to the int variable ch_code. Then we convert this integer to a character. Remember, to the C compiler a character is an integer.

Finally, we test to see if the character was the letter q. If it was, we set the global All_Done variable to TRUE. It is vital that at some point in your program you set All_Done to TRUE. If you don't, your program will hang—that is, it will run until the user turns the computer off.

If the pressed key was not a q, we beep once to let the user know a key-Down event occurred:

```c
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
Handle_Key_Down()
{
    int ch_code;
    char ch;

    ch_code = BitAnd(The_Event.message, charCodeMask);
    ch = ch_code;

    if (ch == 'q')
        All_Done = TRUE;
    else
        SysBeep(5);
}
```

There you have it—your first complete Macintosh program that is capable of handling different types of events. Later, in Chapters 18 and 19, we will expand on what we've learned here to develop a true Mac application—one with menus, dialog boxes, and alerts.
Chapter Summary

A Macintosh program is event-driven. The program waits for the user to take some action—that is, to cause an event to occur. Once the program recognizes the occurrence of an event, it performs. As a programmer, you are the one who decides what action the program should take.

An EventRecord is a Macintosh C structure that describes one event. We first declare a variable to be of EventRecord type. We then use the powerful Toolbox routine GetNextEvent() to grab and latch on to an event. The GetNextEvent() routine stores the information regarding this one event in the EventRecord variable. That gives us an opportunity to process the event at our leisure. When we process an event, we are examining it to determine its type—a mouse click or a key stroke, for example. Processing further involves taking appropriate action, such as pulling down a menu.

We include the GetNextEvent() function within a loop. That way we can check over and over again to see if and when the user performs some action. This is done in what we refer to as an event loop.
Macintosh programs are known for their ease of use. Pull-down menus are an important part of this reputation. Menus allow the user to see what choices are available without having to memorize either the options or the commands for choosing them. They give the user control of what is going on in the program and when the action should take place. All Macintosh programs have the familiar menu bar that runs across the top of the screen. In this chapter you'll learn how to create and control menus so that when the time comes to write a complete program you'll be able to include a menu bar.

You will learn about the parts of a menu, and you'll examine the Menu Manager and menu records and handles.

You'll learn how to create individual menus using MENU resources. Then you will see how to combine these menus into one menu bar by creating an MBAR resource. You’ll see how to incorporate these menu resources into your C language code.

Finally, you will see how evens and menus work together.
The Menu Manager

The Menu Manager, like QuickDraw, is a collection of routines that resides in ROM. Their responsibilities are to handle menu-specific chores such as dropping down a pull-down menu and highlighting a selected menu option. The Menu Manager relays information about a selected menu item to your program so that your program can respond properly.

Before it can use the Menu Manager, your program must first initialize it. This is done by a call to the routine InitMenus():

\[\text{InitMenus();}\]

Recall that this was part of the complete initialization routine that we discussed in the previous chapter. Most Macintosh features, like menus, require some initialization. Initialization routines, and the order in which your program should call them, are covered in detail in Chapter 19.

Description of a Menu

Before you learn how to create and use menus, you should become familiar with the elements of which they are composed.

Figure 15–1 shows the components of a menu. The menu itself,
which extends across the top of the Macintosh screen, is called the *menu bar*. Each of a program's menus has a name that appears in the menu bar.

When a user clicks on a menu name in the menu bar, a menu drops down. Within this menu are the *menu items*. A menu item has several properties, which are discussed below.

A menu item may be *enabled* or *disabled*. A menu item that is disabled appears dimmed. Releasing the mouse button over a disabled menu item has no effect. Conditions within a program cause a menu item to be enabled or disabled.

A menu item may optionally have a *key equivalent*. This is a combination of the ⌘ key and some other key. When these keys are pressed simultaneously, the menu action is performed as if a click of the mouse button had selected it.

Some menu items are followed by an ellipsis (...). This is a visual indication to the program user that more information will be requested when this option is chosen. Often a dialog box opens, as when Print... or Save As... is chosen from most programs.

A menu item will *invert*—that is, turn black with white letters—when you slide the mouse over it. This highlighting tells the user which menu item is about to be selected. When the user releases the mouse button over a menu item, the item will flash a few times, signifying that it has been selected.

In this chapter you'll learn how to use ResEdit to add and edit the items that appear in a menu.

---

**Menus and Resources**

In Chapter 13 we discussed the creation of a WIND resource type using ResEdit. In this chapter you'll create two MENU resources and an MBAR resource using ResEdit. Later in this chapter you'll see the C code that uses these types of resources. If you skipped Chapter 13, you might want to go back and read it now.

**The MENU Resource**

If you're running ResEdit now, you can create your MENU resource from the information on the next few pages. If you're running the Simulator C software, the Simulator will show you how to create a MENU resource. If you aren't running either of these programs, follow along so that you'll know what to do when the time comes.
In ResEdit, open the MyProgram.rsrc resource file that you created in Chapter 13. If you created the WIND resource in Chapter 13, a window like the one shown in Figure 15–2 will appear.

Select Create New Resource from the Resource menu. This will bring up the dialog box that allows you to select the resource type. Scroll down this list of resource types until you come to MENU. Click once on it; then click the OK button.

When you click OK, a window will open, as shown in Figure 15–3. This window allows you to give your menu a name and add menu items to it. ResEdit has assigned our first menu an ID of 128. You could change it to a different value, but 128 is fine.

This is the first of our two MENU resources and will serve as the Apple Desk Accessory menu, denoted by the symbol. This menu appears in the menu bar of all Macintosh programs, so of course you'll want to include it in any programs that you write. ResEdit makes it very simple to turn a MENU into the menu—just click in the radio button labeled (Apple menu). Figure 15–4 illustrates this.

Because it's the menu, there is no title to give it. You will, however, want to add a single item to it—the About… item that will eventually put up a dialog box describing your program. Select Create New Item from the Resource menu, then type About…, as shown in Figure 15–5.

You may have noticed that the menu has a dashed line within it that separates the About… item from the list of desk accessories. You can add a
FIGURE 15-3  The MENU dialog box

dashed line by again selecting Create New Item from the Resource menu. Then click on the radio button titled (separator line). This is shown in Figure 15-6.
You’re done with this MENU resource, so click in the close box. Your screen should now have two windows, similar to the ones shown in Figure 15–7.

At this point you are ready to create a second MENU resource. As you did when you created the first MENU resource, choose Create New
Resource from the Resource menu. A new MENU window, like the one shown in Figure 15–8, will open. ResEdit was smart enough to realize that the last menu you created had an ID of 128, so it gave the new menu an ID of 129. Again, you can give it any ID you want, but leave it at 129 for now. Type in a menu title. We've called it MyMenu, as shown in Figure 15–9.
Note in Figure 15–9 that the name of the menu, MyMenu, appears as the last menu item in the ResEdit menu bar. This menu is for demonstration purposes only. It lets you see how your menu will appear in the menu bar of a program you write.

Notice the lower-right corner of the menu window. If you write a program that takes advantage of color, ResEdit allows you to modify some of the color attributes of a menu. For simplicity, stick with black and white here.

You haven't yet written a program to make use of this menu, but New and Quit are found in almost every program, so create these two items. Remember, with ResEdit you can always go back and change the names of menu items.

Choose Create New Item from the Resource menu. This highlights a bar under your menu title, as shown in Figure 15–10.

Type in the name of the first menu item, New. You can change this name at any time—even after you create a program using this resource. That's one of the powers of resources—the ability to alter their features easily. Your screen should look similar to the one in Figure 15–11.

When you chose Create New Item from the Resource menu, some of the features in the MENU window changed. We won't be using them, but we illustrate them in Figure 15–12.
Now add a second menu item to the menu. Again, choose Create New Item from the Resource menu. Type in the name of a second menu item, in this case Quit, as shown in Figure 15–13.

To see how your menu would appear in an actual program, click on MyMenu at the end of ResEdit’s menu bar. The menu you have created will
Used for hierarchical menus - menus that have a second, or submenu

Turns a menu item into a dashed divider line in the menu

(separator line)

Defines a command key equivalent for a menu item

Places a mark, such as a checkmark, next to a menu item to indicate it controls an on-or-off setting

FIGURE 15-12 Some MENU item features can be modified.

FIGURE 15-13 The MENU dialog box with the second item name in place
FIGURE 15-14 How your menu will look in a program

drop down. It is only an example; it is not functional. Figure 15–14 shows this new menu.

Now you have your MENU resource with two menu items. Click in the close box. The two menus, with IDs of 128 and 129, are displayed in a window. You will use these when you create an MBAR resource. Click in the close box of this window now. The window that remains open on your screen contains two icons—one that lists your previously created WIND and the one that lists your two MENU resources. If you didn’t create a WIND in Chapter 13, you’ll have only the MENU icon.

The MBAR Resource

A Macintosh program’s menu bar contains menus, such as the standard File and Edit menus. These individual menus are created in ResEdit as MENU resources, as you just learned. You then tie the individual menus together by yet another resource type—the MBAR. To create an MBAR resource, select Create New Resource from ResEdit’s Resource menu and then choose the MBAR type. Click the OK button. An MBAR window will open, as shown in Figure 15–15.

You’ll add your two MENU resources to the MBAR. The first step in adding a MENU is to click on the row of asterisks. The row will be enclosed in a rectangle. This serves to enable the Insert New Field(s) menu
item in ResEdit's Resource menu. Next, add the ID of the first MENU resource you want in your MBAR. To add a menu ID, select Insert New Field(s) from the Resource menu. An edit box titled Menu res ID will appear. Type in 128, the ID of the MENU resource. Figure 15–16 illustrates this.

FIGURE 15–16 Typing in the menu ID
Now add a second MENU resource, the one you called MyMenu. Repeat the steps you took in creating the first menu ID:

1. Click on the row of asterisks by the 2.
2. Select Insert New Field(s) from the Resource menu.
3. Type the number of the menu ID of the MyMenu menu, 129, in the edit box that appears.

Click in the close box of the MBAR window. We’re now done with the MBAR resource. Close the window that lists the MBAR. Your screen should contain a window that looks like the one shown in Figure 15–17.

Now that your MBAR resources are complete, select Save from the File menu. If you have finished experimenting with ResEdit, select Quit from the File menu. Now it’s time to look at the source code needed to make use of menu resources.

**Creating a Menu in C**

To create a menu in your program code, you must:

1. Have MENU and MBAR resources in your resource file.
2. Bring the MBAR resource into memory.
3. Bring the MENU resources into memory.
4. Add desk accessories, if needed.
5. Display the menu bar on the screen.

You took the first step when you created the two MENU resources and put them in an MBAR resource. Now you will write the C language code needed to bring them all into memory. The Toolbox routine GetNewMBar() loads an MBAR resource into memory. First declare a handle, then pass the ID of the MBAR resource to GetNewMBar(). Recall from the section on creating an MBAR resource that the ID of your MBAR was 128:

```c
Handle main_menu_bar;
main_menu_bar = GetNewMBar(128);
```

Some Macintosh programs change the main menu bar at different points in the program, so the Macintosh has to know which menu bar it should be using. The Toolbox routine SetMenuBar() will let your program know which menu bar is the current one. Call SetMenuBar() right after GetNewMBar(), passing it a handle to your menu:

```c
main_menu_bar = GetNewMBar(128);
SetMenuBar(main_menu_bar);
```

Next, load the individual menus into memory. Macintosh C has a data type called MenuHandle used specifically for dealing with menus. Declare two variables of this type—one for each of your two menus. Then use the Toolbox routine GetMHandle() twice to load the two menus into memory and assign handles to them. Should you have to work with these individual menus later in a program, the menu handles will allow you to access them:

```c
MenuHandle Apple_Menu;
MenuHandle MyMenu_Menu;

Apple_Menu = GetMHandle(128);
MyMenu_Menu = GetMHandle(129);
```

Here is what is happening when you load a MENU resource into memory using the Toolbox function GetMHandle(). As you read, keep in mind that resources and Toolbox routines reside in different locations:

- Resources exist on disk in a resource file.
- Toolbox routines exist in ROM.
You will eventually compile your source code and link it to your resource file. The result is one executable program. Within that one program are both program code and resources. When you run a program, the code and a *resource map*—not the resources themselves—are loaded into RAM memory. They remain on disk. The resource map holds the disk location of all of the program's resources. When your program calls the Toolbox routine GetMHandle() the Toolbox will look to the resource map to determine where on disk the MENU resource is located. Once found, the resource will be copied from disk and loaded into free RAM memory.

Because your program is in RAM, and all Toolbox routines reside in ROM, RAM will be communicating with ROM. We show this in Figure 15–18. When you purchased your Macintosh, it came with all of the

---

**FIGURE 15–18** Communication between RAM, ROM, and disk
Toolbox routines programmed into ROM. If you were to open your Mac and remove the ROM, your program would not run.

The relationship between program source code, Toolbox routines, and resources is an important one. It's summarized in a slightly different manner in Figure 15–19.

Now that you have an understanding of what's going on when GetMHandle() is called, we can return to the source code. The call to GetM-
Handle() returned a MenuHandle for your program's use. Recall that a handle is a pointer to a pointer. When you want to take some action involving this menu, you will use the menu's handle.

When a GetMHandle() call is made with the ID of the Apple menu as the parameter, a special menu is created. Under it will be housed the desk accessories that are contained in the System Folder of the user's Macintosh. When you created the Apple MENU resource, you did not include any desk accessories as menu items. That's because you don't know what desk accessories users will have on their computers. With a little background information, this dilemma can easily be solved.

Each Macintosh running with System 7 has its desk accessories stored in its System Folder. On Macintosh computers using a System prior to System 7, desk accessories are stored in the System File itself. Recall that each Macintosh has a system resource file of its own. In Chapter 12 we noted that this is where the system stores QuickDraw patterns. Desk accessories are also resources that reside in the system resource file of every Macintosh.

A desk accessory is a type of driver, so its resource type is DRVR. A driver is a type of program that controls the transfer of data between another program and a device. A printer is one example of a device. You will use the Toolbox call AddResMenu() to track down the desk accessory DRVR resources and add them to the Apple menu. Pass AddResMenu() the handle to the menu to which you'll be adding the resources and the type of resource—DRVR—you'll be adding. Enclose the resource type in single quotes. Below is a call to AddResMenu():

AddResMenu(Apple_Menu, 'DRVR');

With the Apple menu in memory and the desk accessories resources added to it, you are ready to show the main menu bar on the Mac's screen:

DrawMenuBar();

Let's combine the individual menu calls now. We also use a few #defines to add clarity to the passed parameters:

```
#define MAIN_MENU_BAR_ID 128
#define APPLE_MENU_ID 128
#define MYMENU_MENU_ID 129

Handle main_menu_bar;
MenuHandle Apple_Menu;
MenuHandle MyMenu_Menu;
```
main_menu_bar = GetNewMBar(MAIN_MENU_BAR_ID);
SetMenuBar(main_menu_bar);

Apple_Menu = GetMHandle(APPLE_MENU_ID);
MyMenu_Menu = GetMHandle(MYMENU_MENU_ID);

AddResMenu(Apple_Menu, 'DRVR');

DrawMenuBar();

In Chapter 19 the above calls are combined in a function and used to set up the menu for that chapter's example program, ShapeMaker.

Events and Menus

In the previous chapter we showed you an event loop that responded to a mouseDown event by calling a function we named Handle_Mouse_Down(). In this section we'll look at the Handle_Mouse_Down() routine, as well as the routines that it calls.

Responding to a Mouse Click

To handle a mouse click, we've elected to use a programming technique called divide-and-conquer. This means that we break one large task into several smaller ones, so we can concentrate on one problem at a time.

Our initial problem is that of handling the occurrence of a single event. The Handle_One_Event() function developed in the previous chapter begins this process. It identifies the type of event that occurred, then calls the appropriate function to handle that event. In the case of a mouseDown event, that function is Handle_Mouse_Down().

Handle_Mouse_Down() is the starting point of a sequence of functions that handle the occurrence of a mouse click. Figure 15–20 gives an overview of the routines that would be called for a mouse click that occurred on the first menu item usually found under a program's ⎯ menu—the About... item. This menu choice usually brings up a dialog box describing the program being run.

In Figure 15–20, an arrow that points to the right tells you that an action other than the one you're concerned with will be dealt with by a different function.
On the following pages we'll cover the routines shown in Figure 15–20, starting with a recap of `Handle_One_Event()`. As you saw in the previous chapter, `GetNextEvent()` captures the most recent event. You extract the type of event from the what field of an `EventRecord` variable. The code for `Handle_One_Event()` appears on the next page.
EventRecord The_Event;

Handle_One_Event()
{
    if (Multifinder_Active == TRUE)
        WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS,
                      MOUSE_REGION);
    else
    {
        SystemTask();
        GetNextEvent(everyEvent, &The_Event);
    }

    switch (The_Event.what)
    {
    case mouseDown:
        Handle_Mouse_Down();
        break;

    case updateEvt:
        Handle_Update_Event();
        break;
    }
}

In this chapter you will be concerned only with mouse clicks. Update events deal with windows—we’ll cover them in the next chapter.

Once you know that an event is a mouseDown event, you’ll call a separate function to handle it. The above code tell us that it is appropriately named Handle_Mouse_Down(). In that routine you must determine if the mouse click occurred on a screen object. Did the click occur on the main menu bar? On a window? The Toolbox routine FindWindow() aids you in this task. You pass FindWindow() the screen coordinates of the mouse click, which are held in the where field of the EventRecord. In return, FindWindow() returns a short that tells you just what the mouse click landed on. We’ll list all of Handle_Mouse_Down() in a moment, but for now, look at this example of the use of FindWindow():

WindowPtr which_window;
short the_part;
the_part = FindWindow(The_Event.where, &which_window);

In addition to returning just what the mouse was clicked on, FindWindow() uses a variable of type WindowPtr, which you use as the second parameter, to return a pointer to the window that was clicked on (if the mouse was indeed clicked while over a window). We'll have more to say about this use of FindWindow() in Chapter 16.

We said that FindWindow() returned the screen part that the mouse was clicked on. We hold this value in a variable named the_part. We'll use this variable a little later in a switch statement to further determine what action we need to take.

**Responding to a Menu Click**

The Toolbox includes a very powerful menu-handling routine—MenuSelect(). Once it has been determined that the mouse was clicked in the menu bar, call MenuSelect(). Once called, MenuSelect() stays in effect for as long as the user holds the mouse button down. The MenuSelect() routine does all of the following:

- Tracks the mouse, showing and hiding pull-down menus
- Highlights individual menu items as the user moves the mouse over them
- Flashes the selected menu item a few times when the mouse button is released and then hides the pull-down menu
- Leaves the menu title highlighted in the menu bar to let you know an item has been selected and is being handled
- Tells your program the menu ID and the menu item selected so that you can process the choice

With your knowledge of the FindWindow() and MenuSelect() routines, you're ready to see what a typical Handle_Mouse_Down() routine might look like:

```c
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
Handle_Mouse_Down()
{
    WindowPtr which_window;
    short the_part;
}*/
```
long menu_choice;

the_part = FindWindow(The_Event.where, &which_window);

switch (the_part)
{
    case inMenuBar:
        menu_choice = MenuSelect(The_Event.where);
        Handle_Menu_Choice(menu_choice);
        break;

    case inSysWindow:
        break;

    case inDrag:
        break;

    case inGoAway:
        break;

    case inContent:
        break;
}

The case labels—inMenuBar and inSysWindow, for example—are Macintosh C constants that are predefined for you. Naturally, in this chapter's version of Handle_Mouse_Down(), we're interested only in menu clicks. We'll discuss the other constants in Chapter 16.

When FindWindow() returns a part with a value of inMenuBar, we call the powerful MenuSelect() function. This takes care of all of the menu highlighting the user sees as the mouse is moved over the menu bar. When the user finally makes a menu choice by releasing the mouse button on a menu item, a number representing the menu and the menu item selection is returned by MenuSelect() to your program. There you save it as a long integer variable.

After a menu selection is made, a separate routine is called. We've named this routine Handle_Menu_Choice(). We pass to this routine the menu-identifying number that MenuSelect() gave us in the previous line of code. This is part of the divide-and-conquer technique mentioned at the
beginning of this section. After identifying an event as a mouseDown event, we call a routine—Handle_Mouse_Down()—that handles only mouseDown events. Further, once we’ve determined that the mouseDown occurred in a menu, we call a routine—Handle_Menu_Choice()—that handles only mouseDown events in menus. Here’s a look at our Handle_Menu_Choice(). A description follows:

```c
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
Handle_Menu_Choice(long menu_choice)
{
    int the_menu;
    int the_menu_item;

    if (menu_choice != 0)
    {
        the_menu = HiWord(menu_choice);
        the_menu_item = LoWord(menu_choice);

        switch (the_menu)
        {
            case APPLE_MENU_ID :
                Handle_Apple_Choice(the_menu_item);
                break;

            case FILE_MENU_ID :
                Handle_File_Choice(the_menu_item);
                break;
        }
        HiliteMenu(0);
    }
}
```

The purpose of Handle_Menu_Choice() is to extract the menu and the menu item from the passed parameter menu_choice. We do this through the use of two Toolbox routines—HiWord() and LoWord(). Notice that menu_choice is declared to be a variable of type long. That means its size is 32 bits. When the MenuSelect() routine called by Han-
The call to MenuSelect() placed the resource ID of the menu in the upper 16 bits of menu_choice and placed the menu item number in the lower 16 bits. It treated the 32-bit variable as two separate variables merged together. It's up to you to break menu_choice down into its two components. If we think of the leftmost 16 bits as the upper bits, then those bits contain the resource ID number of the menu that was clicked on. The rightmost 16 bits then contain the item number of the menu item selected. The first item in each menu is item number one; the second item is item number two, and so forth.

The HiWord() routine returns the value in the upper 16 bits of a long; the LoWord() routine returns the value in the lower 16 bits. Using these two routines, we can determine both the menu and the menu item from the one variable, menu_choice. Figure 15–21 illustrates this.

Earlier in this chapter you created an MENU resource with an ID of 128 and made the first item in it the About... item. If this menu was in a program and the user selected the About... item, we would expect the outcome to be that shown in Figure 15–21.
Now that you have determined just which menu item the user clicked on, use a switch statement to call a routine to handle that choice. Here we show you how we handle two menus. This method can be extended to handle any number of menus:

```c
switch (the_menu)
{
    case APPLE_MENU_ID:
        Handle_Apple_Choice(the_menu_item);
        break;

    case FILE_MENU_ID:
        Handle_File_Choice(the_menu_item);
        break;
}
```

We've finally made it to the last routine in our series of menu-handling functions—Handle_Apple_Choice(). Pass the menu item to this routine. At this point we've covered the basics of menu handling. The final functions, like Handle_Apple_Choice() and Handle_File_Choice(), carry out the actions associated with a menu choice. In Chapters 19 and 20 we provide specific examples that will help you write this type of routine.

### Chapter Summary

Macintosh programs make extensive use of pull-down menus to give program users control of what takes place on the screen. A menu bar holds individual menus. A menu drops down to expose items within it.

Two types of resources are used to create menus. Individual menus are stored in MENU resources. A menu bar is defined by an MBAR resource. An MBAR is a collection of MENU resources. MENU resources are created and edited using the resource editor program ResEdit.

Your C programs will use the MENU resources you create. The Macintosh Toolbox contains routines that aid you in making use of your MENU resources. GetNewMBar() loads an MBAR resource into memory and returns a handle to it. SetMenuBar() gives your program information about the menu bar. Individual MENU resources are loaded into memory with the Toolbox routine GetMHandle().
The menu contains desk accessories. The Toolbox call AddResMenu() locates the desk accessories and adds them to the menu. After the menu bar is loaded into memory and the desk accessories added, use DrawMenuBar() to display the menu bar.

A click of the mouse by a program user is interpreted by your program as an event. Use GetNextEvent() to determine if an event is a mouseDown event. If it is, use the FindWindow() routine to determine where on the screen the mouse click took place.

If an event is a mouse click that occurs in the menu bar, call MenuSelect() to track the mouse as the user holds the button down and moves the mouse. MenuSelect() will tell you the menu and menu item that contained the mouse click. There is one catch though—MenuSelect() gives you these two pieces of information in the form of a single variable. Use HiWord() and LoWord() to extract the two values from the one variable.

Once you know what menu item was selected by the user, it is up to you to act accordingly. In future chapters you will write C code that handles these individual menu selections.
Windows form the foundation of the Macintosh graphical user interface. Any true Macintosh program has at least one window. In this chapter you'll see how QuickDraw creates the illusion of windows as overlapping objects.

You will learn about the elements that make up a window and the different types of Macintosh windows. You'll look at the Window Manager and window records and pointers.

You'll learn how to write C language code that makes use of the WIND resource we created in Chapter 13. Finally, you will see how events and windows work together.

The Window Manager

The Toolbox is responsible for the drawing of windows. The routines that do this and several other window management tasks are grouped together and are collectively called the Window Manager. The Window Manager, like QuickDraw and the Menu Manager, is a collection of routines that
FIGURE 16-1  Windows are drawn to the screen by QuickDraw.

resides in ROM. Their responsibilities are to handle window-specific chores such as resizing and moving windows.

Before you can use the Window Manager, your program must first initialize it. This is done with a call to the routine InitWindows():

InitWindows();

We cover initialization routines and the order in which they should be called in Chapter 19.

Sorry to Disillusion You

As we stressed earlier, QuickDraw is responsible for all the drawing that takes place on your Macintosh screen. A Macintosh window is not a physical object. Like everything else on your computer screen, a window is a group of pixels whose on/off values are defined in such a way that they form a recognizable shape. In that sense, windows are just an illusion! The right side of Figure 16-1 shows a window. The left side of the figure shows an enlargement of a section of that window. When viewed at this level, the
only difference between the window and the desktop is the number of pixels that are turned on.

Fortunately for us, when we create a window the Toolbox takes care of all the drawing needed to frame the window and draw the title bar. When the user moves a window on-screen the Toolbox again takes charge of redrawing the window.

**Description of a Window**

Before we delve into the process of creating a window, we will discuss the basic elements of a window and the different types of windows.

**Window Elements**

Figure 16-2 shows the components of a window. Across the top of a window you see the close box, the title bar, and the zoom box. When the user clicks in it, the close box closes the window. The title bar is also called the *drag region*. The user can drag, or move, the window by clicking and holding the mouse button down in this area. The user expands the window to a size that fills the screen by clicking in the zoom box. The title bar, close box, and zoom box are all drawn and redrawn by the Window Manager.

Windows may contain scroll bars, which allow the user to bring the unexposed contents of a window into view.
The size box allows the window to be resized. Clicking and holding the mouse button while in the lower-right corner of the window lets the user change the dimensions of the window.

The portion of the window below the title bar is called the content region. This includes the area where drawing takes place, the scroll bars, and the size box. The close box, zoom box, and title bar lie outside the content region.

When the user clicks on a window, it becomes the active window—it is activated. This causes its components, including the title bar and scroll bar, to be appropriately highlighted. All other windows on the screen become inactive. The components of these windows are dimmed.

**Window Types**

A window can be any of several window types. We show the six standard window types in Figure 16–3. Macintosh C gives a name to each of the types. The documentProc and the noGrowDocProc are the two most common window types. Desk accessories are usually of type rDocProc. The three types in the second column of Figure 16–3, dBoxProc, plainDBox, and altDBoxProc, can be used for windows but are normally dialog box types rather than windows.

In Chapter 13 we demonstrated how to change the type of a window by editing a WIND resource. We'll touch on this topic again later in this chapter.

**The Window Structure**

In Chapter 12 we first described graphics ports. We said that a graphics port holds graphical information about one drawing area. Macintosh C defines a structure, called a GrafPort, to hold this information. Graphics ports and the C structure GrafPort are very important to the concept of windows.

Imagine that there exists one graphics port and that you call SetPort() to let QuickDraw know that this is to be the current port. You can then call SetRect() to define a box that completely fills the graphics port and then FillRect() to fill it with a white pattern. A call to FrameRect() frames the graphics port. Further imagine that you then call MoveTo() to move the invisible graphics pen in the graphics port and then use DrawString() to draw a line of text to the graphics port. The result on your screen might look something like Figure 16–4.
FIGURE 16-3  The six standard window types

FIGURE 16-4  A poor man’s window
Although the object in Figure 16–4 doesn’t look exactly like a window, it does bear a resemblance to one. And, like a window, it allows you to draw graphics and text to it. On the other hand, it lacks many of the properties that are expected of a Macintosh window. A user of your program can’t move or resize this window. And although you could create a second graphics port that overlapped this one, the user couldn’t click the mouse on one of the windows and get the effect of activating and deactivating windows.

From this experiment two observations can be made. First, graphics ports are very important to the concept of windows. And second, graphics ports alone cannot create the full effect of windows. The idea of merging a graphics port with additional window information is implemented in a Macintosh C structure called a WindowRecord. The first and most important field of a WindowRecord is the port field, which is of type GrafPort. Following the port field come about a dozen fields that describe window-specific properties. The format of a WindowRecord structure is shown below. For simplicity we show only some of the structure’s fields:

```c
struct WindowRecord
{
    GrafPort     port;
    int          WindowKind;
    char         visible;
    char         hilited;
    char         goAwayFlag;
    char         spareFlag;
    ...
    ...
    ...
}
```

Their names may help you to surmise the purpose of some of these fields. Don’t worry about the specifics. Recall that a GrafPort is a structure. We said that its fields are manipulated by Toolbox routines. You don’t have to change the field values directly. The same applies to the WindowRecord structure. You’ll use Toolbox routines to change the value of WindowRecord fields. Because of this, it is more important to become familiar with Toolbox window routines than with the fields of the WindowRecord structure.

We said that the most important field of the WindowRecord structure is the port field, which is a GrafPort. To a Macintosh program, a window is essentially a GrafPort to which it can draw using any QuickDraw routine. A window is a graphics port with some added properties. Because C relies on
pointers, it has a pointer type—the GrafPtr—that points to a GrafPort. We first encountered the GrafPtr in Chapter 12. Macintosh C defines another pointer type, the WindowPtr. Here is the C definition of a WindowPtr:

typedef GrafPtr WindowPtr;

A typedef is C's way of creating a new data type from an existing type. After the keyword typedef comes the existing data type, followed by the name of a new type. The two types, the old and the new, are logically the same. Why this double definition? Simply a naming convention. When we're dealing with windows, we want to think in terms of windows. So when we want to access a WindowRecord port field, we do so with a WindowPtr. If this explanation satisfies you, skip the rather long Note that follows. If you're even a little bit confused, read it over.

**NOTE**

If a WindowPtr is equivalent to a GrafPtr, then what makes a window different from a graphics port? What about the other fields of a WindowRecord that we just mentioned? A window actually has two different pointers associated with it—a WindowPtr and something called a WindowPeek. The WindowPtr is used when dealing with a window as a graphics port. The WindowPeek exists for the times when it is necessary to modify the other fields of a WindowRecord. Here are the definitions of the two:

```c
typedef GrafPtr WindowPtr;
typedef struct WindowRecord *WindowPeek;
```

The C typedef keyword creates new data type names. So the above statements create two new data types—the WindowPtr type, which is equivalent to a GrafPtr, and the WindowPeek type, which is a pointer to a WindowRecord type.

Both types of pointers point to the beginning of a window's WindowRecord. But a WindowPtr allows you to access only the first field of the WindowRecord, the port field, while the WindowPeek allows you to access all of the WindowRecord fields. The illustration at the top of the next page may help. Why have two separate pointers when obviously just the WindowPeek would do? Because a window is usually used by the programmer as just a graphics port. The simple WindowPtr is all you'll usually need. All Toolbox routines that require a GrafPtr as a parameter will work with a WindowPtr as well. The WindowPeek, on the other hand, is available for the few times when you have to access WindowRecord fields other than the graphics port field. WindowPtrs are convenient; WindowPeeks are necessary.
Windows and Resources

In Chapter 13 you created a WIND resource type. In this chapter you will actually load a window into memory. Figure 16–5 shows a WIND resource, as created in ResEdit. Of most importance to you in this chapter will be the WIND ID. Recall that you gave your WIND resource an ID of 128.

![FIGURE 16–5  A WIND resource, as viewed in ResEdit](image-url)
Creating a Window in C

To create a window in your program code, you must

1. Create a WIND resource.
2. Bring that resource into memory.

You met the first requirement in Chapter 13. The Toolbox routine GetNewWindow() will load a window into memory for use by your program. The routine returns a pointer to the window you are creating, so it is necessary to declare a variable of type WindowPtr. As you have done in past chapters, use #defines to make your code clear. The code below demonstrates how simple it is to create a new window:

```c
#define NIL_PTR OL /* zero L */
#define IN_FRONT (WindowPtr)-1 /* typecasting*/
#define MY_WIND_ID 128 /* WIND resource ID */

WindowPtr My_Window;

My_Window = GetNewWindow(MY_WIND_ID, NIL_PTR, IN_FRONT);
```

The routine GetNewWindow() expects three parameters. Let’s take a look at each in turn.

The first parameter is the resource ID of the WIND resource template that was previously created in a resource file.

The second parameter is a pointer that specifies where in memory your window should be placed. A value of OL (that’s the number zero followed by the letter L) is a nil, or invalid, pointer and tells the Window Manager to allocate the memory for you. For most applications this will do just fine.

The third parameter specifies whether this new window should appear behind or in front of any other windows that may be on the screen. A value of 0 places the window behind all others; a value of -1 places the window in front of all others. There is one catch though—this parameter must be a WindowPtr. So we must cast our value of 0 or -1 to a WindowPtr.

The parameters passed to GetNewWindow() in the above example will result in a window that uses the WIND resource template you created (first parameter = 128). The Window Manager will decide where in memory the window will reside (second parameter = 0L). Finally, if any
When in ResEdit, checking this box of a WIND resource means that the window will be visible on screen when created with GetNewWindow().

![Diagram showing properties of a window](image)

**FIGURE 16-6** Setting a visible WIND

other windows were created before this one, this new window will be in front of all of them (third parameter = (WindowPtr)-1).

Recall that a value or variable preceded by a type in parentheses casts, or converts, the value or variable to that of the type given in parentheses. So in this example value -1 is being cast to a WindowPtr. In essence the value -1 is being forced to occupy the same amount of memory as a WindowPtr would. The Toolbox is expecting this third parameter to be a WindowPtr, and you don't want to let the Toolbox down! Chapter 3 provides more information on casting.

In the code fragment above, did the call to GetNewWindow() display the window on the Mac screen? We don't mean to sound evasive, but the answer is: That depends. When you created the WIND template in ResEdit, one of the options you had was whether the window should be initially visible, as shown in Figure 16–6. If you check this box, the window will be displayed upon calling GetNewWindow(). If not, the window will be in memory but will not appear on the screen.

The Initially Visible box was checked when you created the WIND resource in ResEdit—you probably left it that way. But just to play it safe, call a Toolbox routine—ShowWindow()—that displays, or shows, a window.
To use ShowWindow(), simply pass a pointer to the window you want to show:

ShowWindow(My_Window); /* show the window */

The parameter is a pointer to the window we want to show. Our call to GetNewWindow() returned a window pointer, so the parameter is all taken care of. We've repeated our code fragment below with the addition of a call to ShowWindow():

WindowPtr My_Window;

My_Window = GetNewWindow(MY_WIND_ID, NIL_PTR, IN_FRONT);

ShowWindow(My_Window);       /* let's see it */

Using a Window’s Graphics Port

Before learning more about window routines, it might be helpful to verify that at this point you have created a window that can serve some purpose. To do this, draw a line of text to it with a call to DrawString(). In Chapter 12 we mentioned the routine SetPort(). We said this routine was necessary to let QuickDraw know what window was being referred to when a QuickDraw routine was used. Here’s an application of SetPort():

WindowPtr My_Window;

My_Window = GetNewWindow(MY_WIND_ID, NIL_PTR, IN_FRONT);
ShowWindow(My_Window);

SetPort(My_Window);
MoveTo(20, 50);
DrawString("Our very first window");

Recall that SetPort() requires a GrafPtr to be passed to it. As we mentioned earlier, a WindowPtr is identical to a GrafPtr, so using My_Window as the parameter in SetPort() is valid. Figure 16-7 shows the window.

Throughout this book we have used a function we’ve called Make_A_Window(). This function enabled us to include a Macintosh window in our example listings. It’s time to end the mystique surrounding
this function. To create this function, we simply group our window code between braces and give it a name. Listing 16–1 shows the Make_A­ _Window() function.

**LISTING 16–1 The Make_A_Window() function**

```c
#define NIL_PTR OL
#define IN_FRONT (WindowPtr)-1
#define MY_WIND_ID 128

WindowPtr My_Window;

Make_A_Window()
{
    My_Window = GetNewWindow(MY_WIND_ID, NIL_PTR, IN_FRONT);
    ShowWindow(My_Window);
    SetPort(My_Window);
    MoveTo(20,50);
}
```

Because we've used this function throughout the book, it deserves a closer look—we'll look at each part individually now.
#define Statements
The listing begins with three global definitions. These would be placed at the start of your program, before the main() function, so they can be used anywhere in your program.

Global Variable
Following the #define statements, one global variable—My_Window—is declared. Note the use of uppercase letters in the variable name. This lets anyone reading your source code know that My_Window is a global variable and that it can be used anywhere in your program.

The Function
In the Make_A_Window() function, Toolbox routine GetNewWindow() is called to load your WIND resource into memory. A call to ShowWindow() ensures that the window is visible on the screen. The port is then set to this window, and the graphics pen is moved to a point near the upper-left corner of the window.

Seeing the code for this routine may clear up a question you’ve had. In previous examples, the Make_A_Window() routine was called and immediately followed with a call to DrawString():

```
Make_A_Window();
DrawString("\pThat was easy!");
```

Because the port is set to the new window and the graphics pen is moved within the Make_A_Window() function, you now know why the text that is written to the window by DrawString() always starts in the upper-left corner of the window.

Closing a Window

When you’re all finished with a window and want it off the screen, close it. When the user clicks in a window’s close box, your program will close the window by removing it from the screen and from memory. The Toolbox routine DisposeWindow() serves this purpose. DisposeWindow() frees all storage associated with the window. Below is a code fragment that opens a window, shows it, then immediately closes it. Not very practical, but it demonstrates the use of DisposeWindow():
WindowPtr My_Window;

My_Window = GetNewWindow(MY_WIND_ID, NIL_PTR, IN_FRONT);
ShowWindow(My_Window);
DisposeWindow(My_Window);

If you want to make a window invisible without actually disposing of it, use the routine HideWindow(). This routine works in the same manner as ShowWindow()—by passing a pointer to the window we are dealing with. Here's an example:

WindowPtr My_Window;

My_Window = GetNewWindow(MY_WIND_ID, NIL_PTR, IN_FRONT);

ShowWindow(My_Window); /* make the window visible */
SetPort(My_Window); /* set port to this window */

MoveTo(20, 30);
DrawString('Click the mouse to hide the window');

while (!Button()) /* loop here until user clicks the mouse button */
    HideWindow(My_Window); /* hide the window */

Events and Windows

As we did in our Chapter 15 discussion of events and menus, we'll refresh your memory with a recap of the event handling routine developed in Chapter 14:

EventRecord The_Event;

Handle_One_Event()
{
    if Multifinder_Active == TRUE)
        WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS,
            MOUSE_REGION);
An event involving a window can be either a mouse event or an update event. Both the Handle_Mouse_Down() routine and the Handle_Update_Event() routine will contain code that responds to events in a window.

### Mouse Events in a Window

In last chapter's version of Handle_Mouse_Down() we examined the case of a mouse click in the menu bar. Here, we'll look at several more possibilities. Below is an expanded Handle_Mouse_Down() routine:

```c
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
Handle_Mouse_Down()
{
    WindowPtr which_window;
    int the_part;
    long menu_choice;

    the_part = FindWindow(The_Event.where, &which_window);

    switch (the_part)
    {
        case inMenuBar:
            menu_choice = MenuSelect(The_Event.where);
            break;
    }
```
Handle_Menu_Choice(menu_choice);
break;

case inSysWindow:
    SystemClick(&The_Event, which_window);
    break;

case inDrag:
    DragWindow(which_window, The_Event.where, &Drag_Rect);
    break;

case inGoAway:
    if (TrackGoAway(which_window, The_Event.where))
        DisposeWindow(which_window);
    break;

case inContent:
    SelectWindow(which_window);
    break;

The new Handle_Mouse_Down() routine has a switch statement that handles five part code types. Part codes tell at what point on (on what part of) the screen a mouse click took place. The first, inMenuBar, was covered in the previous chapter. The other four—inSysWindow, inDrag, inGoAway, and inContent—deal with events in a window. Let's look at the ways in which these part code types are handled.

**Click on a System Window**

Windows you create in your Macintosh programs are called *application windows*. A window that holds a desk accessory is called a *system window*. When the user clicks the mouse on an open desk accessory, FindWindow() responds by assigning variable the_part a value of inSysWindow and provides you with a WindowPtr to the desk accessory window in variable which_window. You handle this case by calling SystemClick(), which passes the mouseDown event on to the desk accessory, where it is handled as the particular desk accessory sees fit. Here's how inSysWindow is handled:
case inSysWindow:
    SystemClick(&The_Event, which_window);
    break;

**Click on a Window Drag Bar**

The title bar that runs across the top of most windows is also called the drag bar. The user clicks and holds the mouse button down on it, then moves the mouse to move the window. The FindWindow() routine responds to a click in the drag bar of a window by assigning variable the_part a value of inDrag and returning a WindowPtr to the clicked-on window in variable which_window. In response to a the_part value of inDrag, call DragWindow(). This Toolbox function does all of the work for you. Once called, it takes control until the user releases the mouse button. DragWindow() moves the window about the screen as long as the mouse button is held down. inDrag can be handled like this:

```c
case inDrag:
    DragWindow(which_window, The_Event.where, &Drag_Rect);
    break;
```

Have you noticed that in Macintosh programs windows can't be dragged completely off the screen? They usually stop a few pixels from any edge so that they don't disappear off screen. The last parameter passed to DragWindow() is a Rect variable that gives the boundaries within which you want to allow a window to be dragged.

Because Macintosh computers have different screen sizes, the boundaries of this rectangle should vary from computer to computer. The question that arises is: How do you know in advance what size monitor a user will have? The answer: You don't have to! When a user starts up his or her Macintosh, the Mac examines the hardware that is attached to it and sets some of its own global variables with values that hold this information. One such variable is screenBits. The bounds field of this structure is a Rect that holds the size of the screen.

To set up your dragging rectangle, first declare and set a global variable of your own, Drag_Rect, equal to this Macintosh global variable. Then modify the rectangle a little by giving it an offset of a few pixels:

```c
#define DRAG_EDGE 5

Rect Drag_Rect;
```
Drag_Rect = screenBits.bounds;
Drag_Rect.left += DRAG_EDGE;
Drag_Rect.right -= DRAG_EDGE;
Drag_Rect.bottom -= DRAG_EDGE;

**Click on a Window GoAway Box**

When a program user clicks on the close box of a window in your program, FindWindow() gives variable the_part a value of inGoAway. It also places a pointer to the clicked-on window in variable which_window. Use the Toolbox routine TrackGoAway() to see if the user releases the mouse button while still in the close box. If the user does, TrackGoAway() will be TRUE, and you should close the window. inGoAway can be handled like this:

```c
case inGoAway:
    if (TrackGoAway(which_window, The_Event.where))
        Close_Window(which_window);
    break;
```

The routine Close_Window() is one that you will write. Its job is to hide the window, then dispose of the memory it occupies:

```c
Close_Window(WindowPtr which_window)
{
    HideWindow(which_window);
    DisposeWindow(which_window);
}
```

**Click in the Contents of a Window**

When a program user clicks in a window in your program, variable the_part receives a value of inContent from the FindWindow() routine. Call Toolbox routine SelectWindow() to activate the clicked-on window:

```c
case inContent:
    SelectWindow(which_window);
```

**Update Events in a Window**

As we mentioned above, the other kind of window event that can occur is an update event. When a window is partially or completely obscured and then brought back into view, it must be updated—that is, the previously obscured part of the window must be redrawn. Your Handle_One_Event()
routine, repeated below, should respond to an update event by calling a routine that does the updating:

EventRecord The_Event;

Handle_One_Event()
{
    if (Multifinder_Active == TRUE)
        WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS,
                      MOUSE_REGION);
    else
    {
        SystemTask();
        GetNextEvent(everyEvent, &The_Event);
    }

    switch (The_Event.what)
    {
        case mouseDown:
            Handle_Mouse_Down();
            break;

        case updateEvt:
            Handle_Update_Event();
            break;
    }
}

If a window update is in order, you must know which window needs updating—some programs have more than one window open on the screen at one time. The message field of your global The_Event variable holds this information. You must use typecasting to turn this information into a WindowPtr:

WindowPtr the_window;

the_window = (WindowPtr)The_Event.message;

Once you know which window needs updating, you call a routine that takes care of all of the necessary drawing to the window. Place the call to
this routine between the Toolbox routines BeginUpdate() and EndUpdate():

/*+++++++++++++++++++++++++++++++*/
Handle_Update_Event()
{
    WindowPtr the_window;

    the_window = (WindowPtr)The_Event.message;

    BeginUpdate(the_window);
    Update_Window(the_window);
    EndUpdate(the_window);
}

Your window updating routine should draw the entire contents of the window. By nesting the call to this routine between BeginUpdate() and EndUpdate(), you ensure that only the portion of the window that was formerly obscured will be redrawn—not the entire window. The Macintosh is smart enough to know what part of your window updating routine to execute.

What will your window updating routine look like? That depends entirely on what is to be drawn to the window. If your window had just one line of text drawn in it, your routine would look like the following:

/*+++++++++++++++++++++++++++++++*/
Update_Window(WindowPtr the_window)
{
    SetPort(the_window);

    MoveTo(20,30);
    DrawString('\pTesting 1 2 3.');
}

Whatever you draw to your window when you create it should be in the updating routine. And, so you don't repeat code unnecessarily, you should use your one updating routine for two purposes: to draw to a window when you create it and to update that same window. Figure 16–8 illustrates this.
Create a window:
My_Win= GetNewWindow(MY_WIND_ID, NIL_PTR, IN_FRONT);

Call your update routine to initially draw to the window:
Update_Window(My_Win);  

The window becomes obscured...

...then exposed.

Call your update routine to redraw the window contents:
Update_Window(My_Win);

---

FIGURE 16-8 Call your update routine whenever you need to refresh your window.

---

Chapter Summary

QuickDraw draws windows to the screen. As windows are created and moved about the screen, the Window Manager handles the events that take place.

Windows come in several different types and can have several different components, including title bar, close box, zoom box, scroll bars, and size box. The components of a window depend on its type. A window's content region is one component that windows of all types have.

A WindowRecord is a Macintosh C structure that defines the properties of a window. The port field is a GrafPort and is one of the most impor-
tant fields of a WindowRecord structure. The WindowPtr type points to a window's port field.

You use the WIND resource type as a template for a window. The Toolbox routine GetNewWindow() makes use of the WIND resource type in creating a window. The GetNewWindow() routine returns a WindowPtr that can be used by other routines in your program to draw to and move a window.

The Toolbox routine ShowWindow() allows you to display a window to the screen. The routine HideWindow() does just the opposite—it allows you to make a window invisible. The window remains in memory. Call ShowWindow() to see the window again.

After creating a window, you can draw text or graphics to it. First, you must use SetPort() to let QuickDraw know which window to direct its work to. The SetPort() routine allows QuickDraw to draw to the GrafPort of the correct window. Once the correct port is set, use DrawString() or any QuickDraw drawing routine, such as FrameRect(), to draw to the window.

When a mouseDown event occurs, it is your responsibility to handle it. The FindWindow() routine will tell you where a mouse click occurred and in what window. A click in a desk accessory is responded to by a call to SystemClick(). A click in a window's drag bar requires a call to DragWindow(). If a user clicks in a window's close box, call TrackGoAway() to determine if the user released the mouse button in the box. If the mouse click occurred in the contents of a window, call SelectWindow() to activate that window.

When an updateEvt event occurs, you should call a routine that handles the updating. Nest this call between calls to the Toolbox routines BeginUpdate() and EndUpdate(). Write your update routine such that it does all the drawing that is necessary for the window you have on screen.
Chapter 17

Dialog Boxes and Alerts

Dialog boxes are the user's link to a Macintosh program. Like Macintosh menus, dialog boxes offer a user-friendly way to make program choices.

In this chapter you'll learn how dialog boxes are similar to windows and how they differ. You will also look at the Dialog Manager and dialog records and pointers.

ResEdit will be used to create DLOG and DITL resources—essential templates for bringing dialog boxes to the screen. You'll then learn how to write C code that uses these resources.

Dialog boxes contain items—buttons, check boxes, and text fields. Here, you'll learn how to respond when a user clicks the mouse over one of these items.

In this chapter you will also examine alerts, which are simplified dialog boxes. You'll use ResEdit to create an ALRT template, then write the code necessary to put an alert on the screen.

Finally, you will revisit the event loop to see how events, dialog boxes, and alerts work together.
The Dialog Manager

Like all of the managers, the Dialog Manager is a set of Toolbox routines that resides in the Macintosh ROM. The routines that make up the Dialog Manager create and manage dialog boxes and alerts. The Dialog Manager operates at a higher level than most of the other managers; it calls several routines that are part of other managers, including the Event Manager and the Window Manager.

Before it can use the Dialog Manager, your program must initialize it. This is done with a call to the routine InitDialogs():

```c
InitDialogs(0L);
```

As we've mentioned in the past, the order in which initialization routines are called is important. An example is given in Chapter 19.

Description of a Dialog Box

When the Macintosh program needs information from the user, it relies on dialog boxes. A dialog box usually appears on the screen as the result of a menu selection. For example, if you want to alter the settings on your ImageWriter printer, you select the Page Setup... option from the File menu of your word processor. The result is a dialog box similar to the one shown in Figure 17-1.

![ImageWriter v2.7 dialog box](image)

**FIGURE 17-1** This typical dialog box helps the user set up the page.
There are several ways in which dialog boxes allow the user to communicate with the Macintosh. Figure 17–1 shows radio buttons, check boxes, icons, and push buttons, for example. You’ll become familiar with these and edit boxes later in this chapter.

**Types of Dialog Boxes**

Dialog boxes come in two types—*modal* and *modeless*. Modal dialog boxes are fixed on the screen and cannot be moved—only dismissed. Modeless dialog boxes, in contrast, look and act much like windows. You can drag a modeless dialog box across the screen, and it may have a close box. Figure 17–2 shows examples of both types.

Most of the dialog boxes you encounter are of the modal type. In this chapter you’ll learn about the resources and code needed to create modal dialogs.
A dialog box can have the appearance of any of the six window varieties discussed in Chapter 16. Of the six standard window varieties presented in that chapter, the three most commonly used for dialog boxes are dBoxProc, plainDBox, and altDBoxProc as shown in Figure 17–3. You are probably most familiar with the dBoxProc type.

In this chapter you will learn how to change the look of a dialog box by editing a DLOG resource using ResEdit.

**Dialog Items**

A dialog box can hold one or more items. Dialog items allow the user to interact with the Macintosh. Figure 17–4 shows a dialog box with its items labeled. There are several types of items.
FIGURE 17-4 Items in a dialog box allow the user to interact with the computer.

Control Items

Controls are the most common and most important types of dialog items. There are four types of controls—push buttons, check boxes, radio buttons, and dials.

- **Push buttons** cause an action to be performed when the mouse is clicked on one. The OK and Cancel buttons found in many dialog boxes are push buttons.

- **Check boxes** are used to turn options on and off. Clicking the mouse in a check box has a *toggling* effect, checking and unchecking the box. Action will take place at a later time, usually after the user dismisses the dialog box.

- **Radio buttons**, like check boxes, set options. You'll find radio buttons in groups. Only one radio button in a group can be on at any given time. Clicking the mouse in one radio button turns it on and turns off the button that was previously on.

- **Dials** give the user a range of choices, rather than the two on/off choices of buttons and check boxes. A dial has some kind of indicator that tells the user what the current setting is. The scroll bar is a dial.
Edit Text Items

Edit text items, or edit text boxes, allow the user to type in text from the keyboard. Text boxes appear as framed rectangles. The user may use the mouse or the Tab key to move the insertion point to a text box.

Static Text Items

Dialog text that the user cannot alter is static text. Examples of static text items are the title of a dialog box and any instructions that might appear in it. Static text items are usually, but not always, inactive. By that we mean that they are for visual effect only, and mouse clicks on them have no effect.

Icons and Pictures

Dialog boxes may contain icons and pictures. Icons are 32-pixel squares, while pictures can be of any size. Icons and pictures, like static text, are usually, but not always, inactive.

User Items

The item types we've covered up to this point are predefined and follow certain rules. User items, on the other hand, are free-form. There is no preset definition for user items. You select the area of a dialog box that will contain a user item and then write C code that both draws and displays that item.

Dialog Boxes and Resources

By now you should be very familiar with ResEdit. In previous chapters you created WIND and MENU resource types. Here, you will create a Dialog (DLOG) resource and a Dialog Item List (DITL) resource, which will give you the resources you need to load a dialog box into memory. You'll want to gain experience in creating the four main DITL types—radio buttons, check boxes, push buttons, and edit boxes. For that reason we present a simplistic project: create a DLOG and a DITL for a program that allows the user to set a new speed limit. Your program should ask the user if the old limit should be saved, if the limit should be raised or lowered, and what the value of the new limit will be.

Start out by creating the DLOG resource. You can do this by running ResEdit or by following along with the Simulator software. If you're run-
If you have previously created both MENU and WIND resources, your screen should look similar to the one shown in Figure 17-5.

Select Create New Resource from the Resource menu of ResEdit. The Select New Type dialog box will then open. Click on the DLOG type, then click on the OK button. A window, similar to the one in Figure 17-6, will open. This window allows you to edit some of the features of the dialog box.

If this DLOG window looks familiar to you, your memory serves you correctly. It is almost identical to ResEdit’s WIND editing window. There is one key difference, however. Figure 17-7 shows this difference.

The only difference between a WIND resource and a DLOG resource is that a DLOG has a DITL resource associated with it. The DLOG resource defines the general look of your dialog box, including its size and type. The DITL is the Dialog Item List, a listing of all of the items, such as buttons and check boxes, that will be displayed in the dialog box. The DITL resource defines what is displayed in your DLOG.

If you were to create more than one DLOG, you would also have more than one DITL. You are responsible for pairing a dialog box with its
FIGURE 17-6  Creating a DLOG resource in ResEdit

FIGURE 17-7  DLOGs have DITLs.
dialog item list. You do this by entering the proper DITL number in the DITL ID edit box.

Notice that your dialog box has an ID of 128. This is displayed in the DLOG window’s title bar. This value of 128 is the default ID given to the first DLOG you create. ResEdit assigned this ID for you. As always, ResEdit allows you to change the ID, but, for this example, leave it at 128. Now look at the DITL ID edit box in the DLOG window. In it appears the number 128. Again, this is a value assigned by ResEdit. Because it has given the DLOG a value of 128, it assumes that you will create a DITL with the same number. This is good practice; give both the DLOG and the DITL that goes along with it the same ID number.

As you did in the WIND resource, you can edit some of the features of your DLOG. Make your DLOG look like the one in Figure 17–8 by doing the following:

1. Change the DLOG Width to 350.
2. Change the DLOG type by clicking on the fourth icon from the right in the row of icons at the top of the DLOG window.
3. Click on the close box check box so that the DLOG has no close box.

![Figure 17–8 Altered DLOG](image-url)
Now that your DLOG is complete, you're ready to create the accompanying DITL resource. Click on the close box of the DLOG window. You'll be left with a window that displays the ID of each DLOG you have. At this time you should have just one. Click on the close box of this window. Now that your DLOG resource is complete, select Save from the File menu to preserve your changes.

**DITL Resource**

As you've done for all your resource types, select Create New Resource from the Resource menu of ResEdit. The Select New Type dialog will then open. Click on the DITL type, then click the OK button. A window that allows you to add items to the DITL will open, as shown in Figure 17–9.

Now select Show Item Numbers from the DITL menu. Each item in a DITL has an item number. Choosing this menu option causes ResEdit to display the number of each item.

You'll become familiar with the DITL resource by adding items to it, beginning with a button. Next to the DITL window is a floating palette that simplifies the process of adding items to the DITL. Click and hold the mouse button on the Button option. With the mouse button still down, drag over to the DITL window and release the mouse button. You've just created a button! Figure 17–10 illustrates this.
To edit the text that appears in the button, double-click on the button. An Item Edit window like the one shown in Figure 17-11 will open.

To change the text in the button, type in new text. Most dialog boxes have an OK or a Done button, so name your button Done. To resize or move the button, type new button dimensions. Notice that as you type in new numbers the button moves and changes size. This is shown in Figure 17-12.
One important observation you should make about the button's edit window is that the check box titled Enabled is checked. This is very important. In order for your program to respond to user clicks on the button, it must be enabled. This is shown in Figure 17–13.

Now that you have created your button, click in the close box of the button's edit window. Notice the small number 1 that appears in the upper-

![Figure 17–12 Changing the size and placement of a button](image)

**FIGURE 17–12** Changing the size and placement of a button

![Figure 17–13 Make sure a button's Enabled check box is checked.](image)

**FIGURE 17–13** Make sure a button's Enabled check box is checked.
right corner of the button, as shown in Figure 17-14. This is the item number that every item must have. When creating a DITL, keep a list of the item numbers that go with the items. Later, when you're writing source code that uses this DITL, you will want to refer to the DITL items by their item numbers.

Using the same technique that you used to create the button, you will now add a check box, two radio buttons, an edit text item, and a static text item to your DITL. Use the floating palette for each item you add. Once an item has been placed in the DITL window, you can double-click on the item to open the item edit window. There, you can change the text of the item and set its location. Figure 17-15 shows how the finished DITL will look. Use the figure as a reference for the placement of the items you add. ResEdit numbers items according to the order in which they are added to a DITL.
To add a check box, click on Check Box in the floating palette and drag the mouse over to your DITL. Release the mouse button, then double-click on the check box. Enter **Save Old Speed Limit?** as the new text for the edit box.

Next, add two radio buttons to your DITL. Add them one at a time, selecting the Radio Button option from the floating palette for each. Add text to each, as you've done for the other items. The first should say Raise Limit; the other should say Lower Limit.

The next item you will want to add is an edit box item. An item of this type allows the user to enter text into it—it allows for user interaction with a program. Because a program must know about typing that takes place in an edit box, be sure to enable edit box items when you create them.

The final item you can add is a static text item. Double-click on the item after adding it. This opens the window that allows you to enter the static item's text. Note that the Enabled check box is not checked. Static text is usually used for titles and labels, and as such, it does not have to be enabled. If a user clicks on a static text item, the program does not have to be aware of it, because no action will be taken. Enter **Enter New Limit:** for the static text item.

Note the item number of each item, and let's take a closer look at the DITL. Select Show Item Numbers from the DITL menu to hide the item numbers. Depending on where you placed your DITL items, your DITL should look similar to the one pictured in Figure 17–16.

It has been a while since you started working on the DITL, but hopefully you haven't forgotten its place in the scheme of things. Every DLOG has a corresponding DITL. The DLOG is the framework of a dialog; the
DITL is the contents. You want your DITL to be paired with the DLOG you created earlier in this chapter. Recall that your DLOG has DITL ID 128 paired with it—so you want your DITL to have an ID of 128. And that's the ID ResEdit gave it by default—the value it automatically gives a resource. So you're all set. If you do have to change the ID, you can use the same procedure you would use to change the ID of any resource—use the Get Resource Info option from the Resource menu and type in a new ID number.

Let's see if your effort to associate your DITL with your DLOG worked. Open the DLOG by double-clicking on the DLOG icon, then double-clicking on ID 128, as shown in Figure 17–17.

Figure 17–18 shows the DLOG window, and, lo and behold, the DLOG now has items in it—the items of DITL 128.

Now that your DLOG and DITL resources are complete, select Save from the File menu. Then select Quit from the File menu to quit ResEdit.
The Structure of a Dialog Box

A dialog box can be thought of as a glorified window—a window with a few extra features. In fact, as the next section will show you, the dialog box structure has a window embedded in it.

Dialog Box Pointers and Records

The structure that defines a dialog box is the DialogRecord. A dialog box is so similar to a window that it actually contains an entire window record within its definition. The first field of the dialog box structure is a WindowRecord:

```c
struct DialogRecord
{
    WindowRecord window;
    Handle items;
    THandle textH;
};
```
int editField;
int editOpen;
int aDefItem;
}

Now that you know that a dialog box is similar to a window, it should come as no surprise that the graphics port plays an important role in a dialog box. In the preceding chapter we said that a WindowPtr was the same as a GrafPtr. Now we introduce the DialogPtr, which is equivalent to a WindowPtr:

typedef GrafPtr WindowPtr; /* WindowPtr is a GrafPtr */
typedef WindowPtr DialogPtr; /* DialogPtr is a WindowPtr */

This essentially gives us three names for the very same data type—GrafPtr, WindowPtr, and DialogPtr. All three point to a GrafPort. Again, this multiple definition of types is for your convenience. When working with dialog boxes, you'll find yourself thinking in terms of dialog boxes, so a data structure named DialogPtr will fit right into your programming thoughts.

NOTE

If you read in the previous chapter the Note concerning WindowPtrs and WindowPeeks, you already have a pretty good idea of what a DialogPeek is.

Like a window, a dialog box has two different pointers associated with it. For a dialog box, these are the DialogPtr and the DialogPeek. The DialogPtr is used when working with the graphics port of the WindowRecord field of the DialogRecord. The DialogPeek is used to access the other fields of a DialogRecord. Here are the definitions of the two:

typedef WindowPtr DialogPtr;
typedef struct DialogRecord *DialogPeek;

Like the WindowPeek, which is used to access WindowRecord fields, the DialogPeek exists so that you can access DialogRecord fields other than the graphics port of the WindowRecord field. Because they're identical, all Toolbox routines that require a GrafPtr as a parameter will work with a DialogPtr as well. In this book we will be most concerned with the DialogPtr.
The items that are to appear in a dialog are defined in the resource DITL type. The second field of the DialogRecord, the items field, is a handle to this item list.

Creating a Dialog Box in C

Since you've read Chapter 16, you already understand the basic principles involved in creating a dialog box. That's because dialog boxes are very similar to windows. If you skipped Chapter 16, we strongly recommend that you go back and read it—the information presented there will apply here.

To create a dialog box in your program code, you follow the same two steps used in creating a window. Only the resource type is different:

1. Create a DLOG and a DITL resource.
2. Bring the DLOG resource into memory.

So far, most of this chapter has been devoted to the first step. The second step requires C language code. Use the Toolbox routine GetNewDialog() to load your dialog box into memory so it can be used by your program. The GetNewDialog() routine returns a pointer to the dialog box you are creating. That means that you must declare a variable of type DialogPtr. Here is a code fragment for creating a dialog box:

```c
#define NIL_PTR OL
#define IN_FRONT (WindowPtr)-1
#define MY_DLOG_ID 128

DialogPtr the_dialog;

the_dialog = GetNewDialog(MY_DLOG_ID, NIL_PTR, IN_FRONT);
```

The routine GetNewDialog() needs three parameters. The first is the resource ID of the DLOG resource template to be loaded into memory. The second and third parameters serve the same purpose as those of GetNewWindow(). The second parameter—OL—tells the Dialog Manager to allocate the necessary memory for you. The third parameter—(WindowPtr)-1—places the dialog box in front of all others.

To ensure that your dialog box is visible, you can use the Toolbox routine for showing a window—ShowWindow():

```c
ShowWindow(the_dialog);
```
ShowWindow() is not the only routine you can use on dialog boxes; you can use many other Toolbox routines devoted to windows as well. That's because a DialogPtr is identical to a WindowPtr. If a Toolbox routine requires a WindowPtr as a passed parameter, it can usually be used for dialog boxes as well. Figure 17–19 shows the dialog box that results from your call to GetNewDialog().

In Chapter 16 you learned how to draw to a window. Drawing to a dialog box is done in the same manner. Create the dialog box, set the port to that of the dialog box using SetPort(), then use a QuickDraw routine like DrawString() to draw to it. The code fragment on the next page is very similar to one used in Chapter 16. Figure 17–20 shows the result.

FIGURE 17–19 The dialog box resulting from GetNewDialog()
The purpose of a dialog box is to provide a way for the user to communicate with the program. Once you put the dialog box on the screen, it is your job to monitor the user’s actions and respond accordingly. You leave the dialog box on the screen until the user tells you to remove it, usually by clicking a button labeled Done or OK.

Because the user can click on more than one dialog item or on the same item over and over, it is best to handle a dialog box with a loop. At the start of the loop, determine which item the user clicked the mouse on. The remainder of the loop is devoted to carrying out an action in response to the user’s choice. To determine which dialog item was clicked on, use the Toolbox function ModalDialog():

```c
#define NIL_PTR 0L
#define IN_FRONT (WindowPtr)-1
#define MY_DLOG_ID 128

DialogPtr the_dialog;

the_dialog = GetNewDialog(MY_DLOG_ID, NIL_PTR, IN_FRONT);
ShowWindow(the_dialog);
SetPort(the_dialog);
MoveTo(50, 160);
DrawString('Drawing to a dialog.');
```

The ModalDialog() function intercepts mouse events and returns the number of the selected, or clicked on, dialog item. Earlier in this chapter we discussed using ResEdit to enable items. Some items, such as buttons, need to be enabled so that ModalDialog() will recognize a mouse click that occurs on them. The format to use for ModalDialog() is to put it at the top of the body of a loop, then compare the returned value to your DITL items to see which item was clicked on. Here’s a pseudo code version:
while (Not Done With Dialog)
{
  ModalDialog(NIL_PTR, &the_item)
  Compare the_item To DITL Items
  Take Action Based On the_item
}

The number returned in variable the_item is the item number of the item, as defined in the DITL. When you were creating your DITL earlier in this chapter, we advised you to make a list of the item numbers of the DITL items. This is one time when you need them. Figure 17–21 looks back at your DITL and displays the number of each item.

Notice that we declared the_item as a short. Recall from Chapter 3 that a short is like an integer with a smaller range of values. Several of the Toolbox functions that we'll be using later on will be expecting an argument of type short. We'll be using the_item as the argument, so we'll declare it as a short.

Create a #define for each of the five enabled DITL items. Then, when ModalDialog() returns an item number in its passed parameter variable the_item, you'll have something to compare it to:

#define DONE_BUTTON_DITL_ITEM 1
#define CHECK_BOX_DITL_ITEM 2
#define RADIO_1_DITL_ITEM 3

FIGURE 17-21 Recalling the DITL item numbers
Notice that there is no #define for item number 5. Item 5 is a static text item that is not enabled—you don’t care about, nor will you respond to, mouse clicks on this text item.

A call to ModalDialog() tells you which dialog item was clicked on—it assigns the dialog item number of the clicked-on item to the variable the_item:

```c
ModalDialog(NIL_PTR, &the_item); /* the_item gets a value */
```

Compare this item number to your #defined values and take action. Use a switch statement for the comparison. Below is a function that can be used to handle a dialog:

```c
Handle_Dialog()
{
    DialogPtr the_dialog;
    short the_item;
    Boolean done_with_dialog = FALSE;

    the_dialog = GetNewDialog(MY_DLOG_ID, NIL_PTR, IN_FRONT);
    ShowWindow(the_dialog);

    while (done_with_dialog == FALSE)
    {
        ModalDialog(NIL_PTR, &the_item);
        switch (the_item)
        {
            case DONE_BUTTON_DITL_ITEM:
                Get_Dialog_Info(the_dialog);
                done_with_dialog = TRUE;
                break;
            case CHECK_BOX_DITL_ITEM:
                Set_Check_Box(the_dialog, the_item);
                break;
            case RADIO_1_DITL_ITEM:
                Set_Radio_Buttons (the_dialog, the_item);
                break;
            case RADIO_2_DITL_ITEM:
```
Set_Radio_Buttons (the_dialog, the_item);
break;
}
}
HideWindow(the_dialog);
DisposDialog(the_dialog);

Notice that the while loop in this code will not terminate until DONE_BUTTON_DITL_ITEM is clicked on. At that time, call one of your own functions—Get_Dialog_Info()—to help you glean what information you can from the items in the dialog box. Then set done_with_dialog to TRUE. That will end the loop. Once the loop has ended, hide the dialog box and dismiss it with a call to the Toolbox routine DisposDialog(). A call to DisposDialog() will free the memory occupied by the dialog box.

If the check box is clicked on, call your own Set_Check_Box() function to check or uncheck the check box. If either radio button is clicked on, call Set_Radio_Buttons(). Set_Radio_Buttons() will be written so that it turns the clicked-on radio button on and turns the previously selected radio button off. Later in this chapter you’ll see the code that makes up these two functions.

Notice that we make no effort to handle a mouse click in the edit text item. Fortunately, the Macintosh handles that for you. If the user clicks in an edit text item, the cursor is automatically moved into the edit box. When the user types, the keystrokes are reflected in the edit box without any help from you. Your job is to determine what the user typed in the edit box. That will be covered a little later in this chapter.

Now you know the basic ways to handle a dialog box. But three functions are key to getting information from the dialog box and reflecting the user’s actions. These functions are Get_Dialog_Info(), Set_Check_Box(), and Set_Radio_Buttons().

### Accessing Dialog Items

The preceding section demonstrated how to bring a dialog box onto the screen and how, through the use of a loop and ModalDialog(), to monitor the user’s mouse actions in that dialog box. Once you discover that an enabled item has been clicked on, you must access it—to determine its current value, set it to a new value, or perform some other action. This section is devoted to the very important topic of accessing the items in a dialog box.
Getting Push Button Items

Push buttons are the easiest items to work with. When the user clicks on one, its DITL item number is returned by ModalDialog() in the variable the_item.

Include the item number of the button as a case label in HandleDialog(). Under the case, take any action appropriate for that button. For a Done button, you'll want to set the flag that ends the while loop—done_with_dialog in our example:

```
ModalDialog(NIL_PTR, &the_item);
switch (the_item)
{
    case DONE_BUTTON_DITL_ITEM:
        done_with_dialog = TRUE;
        break;
    ...
    ...
}
```

Getting and Setting Check Box Items

A check box is an independent item whose state is toggled. A dialog box can have a single check box, as opposed to radio buttons, which always appear in groups of at least two. We'll have more to say about radio buttons in the next section. When we say that a check box is toggled, we mean that clicking the mouse in it alternately checks and unchecks it—changes it to the value opposite from the value it had before the mouse click.

When a mouse click occurs in a check box, you first get the current value, or state, of the check box item. You then set the check box value to the opposite state. Telling you that a check box has just two settings should clue you in to the fact that a check box item can have only one of two values—a 0, for off, or a 1, for on. Before you can get the current value of a check box or set a check box to a new value, you need a handle to the check box item.

Recall from Chapter 9 that a handle is a pointer to a pointer. This is a bit confusing in theory, but in application there is not much to it. The handle tells your program just where in memory the DITL item resides. The Toolbox routine GetDltem() provides you with a way to get a handle to a dialog item—any dialog item. The GetDltem() routine requires five arguments. Before calling the routine, you set the first two—a pointer to the dialog box you're working with and the item number of the item you're
working with. Upon return, GetDlgItem() sets the remaining three items for you. Of these three remaining arguments, only one is of interest to you—the argument that is a handle to the item.

Below we list declarations of the five required arguments, followed by a call to GetDlgItem():

```c
DialogPtr the_dialog; /* pointer to dialog with item */
short the_item; /* DITL Item Number of the item */
short item_type; /* type of item */
Handle item_handle; /* handle to the item */
Rect item_rect; /* rectangle enclosing the item */
```

GetDlgItem(the_dialog, the_item, &item_type, &item_handle, &item_rect);

Notice that the last three parameters to GetDlgItem() are all pointers. When the call to GetDlgItem() is complete, these three pointers will contain the address of the item's type, the address of a handle to the item, and the address of the rectangle that encloses the item. You're interested in the item's handle. Once the call to GetDlgItem() is complete, &item_handle contains the address of a handle to the item, so item_handle is the handle to the item.

The dialog handling code introduced in the last section provides the pointer to the dialog box and the item number of the clicked-on item. Figure 17-22 summarizes the use of GetDlgItem() and the arguments it needs.

Once you've received a handle to the dialog item, you can get, or examine, the item's value, or you can set the item's value. To accomplish these tasks, use two related and very useful Toolbox routines—GetCtlValue() and SetCtlValue(). The Ctl in these functions is an abbreviation for control. Check boxes are control items. Each of these routines requires a handle to the dialog item you are going to work with. If you're going to change the value of a check box, you must determine its current value—after the change, this will be the old value. Here is another look at the GetDlgItem() code, with the addition of the declaration of old_value and a call to GetCtlValue():

```c
DialogPtr the_dialog;
short the_item;
short item_type;
Handle item_handle;
```
FIGURE 17-22 The arguments of GetDItem()

Rect item_rect;
int old_value;

GetDItem(the_dialog, the_item, &item_type, &item_handle, &item_rect);

old_value = GetCtlValue((ControlHandle)item_handle);

The only argument that GetCtlValue() requires is a handle to the item being investigated. But the handle can't be just any handle—it must be the Macintosh C type ControlHandle. We obtain a handle to the item using GetDItem(), but we need a variable of the more specific ControlHandle type as an argument for GetCtlValue().

To make this transition from Handle to ControlHandle, we rely on a trick you learned back in Chapter 3—casting. There we said that to force a variable of one type to become a different type, we precede the variable
name with the new type name in parentheses. Here's an example that turns
a variable of type Handle into one of type ControlHandle:

Handle item_handle;
ControlHandle cntrl_handle;

cntrl_handle = (ControlHandle)item_handle;

GetCtlValue() returns the current state of the item. Save this value in
the variable old_value.

To set a check box value, use SetCtlValue(). This function requires two
arguments—a handle to the item to set and the value to set the item to. As
with GetCtlValue(), the handle must be a ControlHandle.

We said that a check box is on when it has a value of 1, off when it has
a value of 0. So the two calls to SetCtlValue() that you'll be making for
check boxes are:

#define CONTROL_OFF 0
#define CONTROL_ON 1

SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);
SetCtlValue((ControlHandle)item_handle, CONTROL_ON);

Earlier we said that when a check box is clicked on by the user, you
toggle its value. That means you must get its current value, using
GetCtlValue(). Then, based on that value, you set the check box to the
opposite value using SetCtlValue(). Once you know the value, you can use
an if-else statement to determine whether to turn the check box on or off:

if (old_value == CONTROL_ON)
    SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);
else
    SetCtlValue((ControlHandle)item_handle, CONTROL_ON);

The if-else says that if the check box is on, turn it off. If it is off, turn it
on. Below we've combined all of our code fragments in a function called
Set_Check_Box():

Set_Check_Box(DialogPtr the_dialog, short the_item)
{
    Handle item_handle;
    short item_type;
}
Rect item_rect;
int old_value;

GetDlgItem(the_dialog, the_item, &item_type, &item_handle, &item_rect);

old_value = GetCtlValue((ControlHandle)item_handle);

if (old_value == CONTROL_ON)
    SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);
else
    SetCtlValue((ControlHandle)item_handle, CONTROL_ON);
}

Passing the dialog pointer and the item number of the DITL check box item to Set_Check_Box() makes the function very versatile. It can now be used for any dialog box and any check box item. The Handle_Dialog() function developed earlier uses Set_Check_Box() when it is determined, via ModalDialog(), that the check box was clicked on:

...
...

ModalDialog(NIL_PTR, &the_item);
switch(the_item)
{
    ...
    ...
    case CHECK_BOX_DITL_ITEM:
        Set_Check_Box(the_dialog, the_item);
        break;

Now that you know how to access a check box DITL item from within your C source code, you can apply the same principles to other types of DITL items, such as radio buttons.

**Getting and Setting Radio Button Items**

A radio button is a dependent item that has two states—on and off. What is a radio button dependent on? Other radio buttons. Unlike check boxes, which may or may not appear alone in a dialog box, radio buttons always
occur in groups. When the mouse is clicked on a radio button, it is turned on, and whatever radio button was on in its group is turned off. The effect is that of a push-button car stereo, where pushing one radio station button turns that station on and the previous station off. Your car's stereo can never have two radio stations on at once. Likewise, you should never allow two radio buttons in the same group to be on at the same time.

Unlike check boxes, radio buttons are not toggled—clicking on a radio button that is already on does not turn it off. When a radio button is on, only clicking on a different radio button can turn it off.

When a mouse click occurs in a radio button, you must first turn off the radio button that was on, the old button. You then turn on the button that the user clicked. Like a check box, a radio button can have just two values—a 0 for off, or a 1 for on.

When dealing with a group of radio buttons, declare one global variable—Old_Button_Num. Note that we capitalize the first letter of each word to remind us that it is a global variable. The purpose of Old_Button_Num is to hold the DITL item number of the radio button that is currently on. Why make it a global variable? Because you'll want the program to remember its value between function calls—something that cannot be done with a local variable. The purpose of preserving this value in a global variable will become clear soon.

As with check box items, before you can access a radio button, you must have a handle to the radio button item. The Toolbox function GetDItem(), covered above, will get you the handle you need. Once you have a handle to the radio button that was on before the mouse click, you pass it to SetCtlValue() to turn that radio button off:

```c
#define RADIO_1_DITL_ITEM 3
short Old_Button_Num = RADIO_1_DITL_ITEM  /* global */

DialogPtr the_dialog; /* these are local to a function */
short item_type;
Handle item_handle;
Rect item_rect;
int old_value;

GetDItem(the_dialog, Old_Button_Num, &item_type, &item_handle, &item_rect);
SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);
```
The DITL of a dialog box provides the item numbers for the items you're working with.

FIGURE 17–23  Look to the DITL of the dialog box for item numbers you need.

This code fragment turns the radio button with a DITL item number of 3 off. The number 3 comes from the DITL of the dialog box you are working with. Figure 17–23 illustrates this.

After turning the old radio button off, turn the clicked-on radio button on by calling SetCtlValue() again. But first, you must call GetDItem() a second time. This time you need a handle to the radio button you want to turn on. Let's repeat the code listed above and add a short variable named new_button_num to hold the item number of the radio button to turn on. We'll also add the second calls to GetDItem() and SetCtlValue():

```c
short Old_Button_Num = RADIO_1_DITL_ITEM

DialogPtr the_dialog;
short item_type;
short Handle item_handle;
short Rect item_rect;
int old_value;
short new_button_num;

GetDItem(the_dialog, Old_Button_Num, &item_type, &item_handle,
         &item_rect);
SetCtlValue ((ControlHandle)item_handle, CONTROL_OFF);
```
GetDItem(the_dialog, new_button_num, &item_type, &item_handle, &item_rect);

SetCtlValue((ControlHandle)item_handle, CONTROL_ON);

Old_Button_Num = new_button_num;

The final line in this code fragment sets Old_Button_Num to the value of new_button_num. Once a radio button is turned on, it becomes the old radio button, and you start waiting for a new radio button to be clicked. Now, you're ready to turn your radio button code fragments into a usable function—Set_Radio_Buttons():

short Old_Button_Num = RADIO_1_DITL_ITEM

Set_Radio_Buttons(DialogPtr the_dialog, short new_button_num) {
    .Handle item_handle;
    short item_type;
    Rect item_rect;

    GetDItem(the_dialog, Old_Button_Num, &item_type, &item_handle, &item_rect);
    SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);

    GetDItem(the_dialog, new_button_num, &item_type, &item_handle, &item_rect);
    SetCtlValue((ControlHandle)item_handle, CONTROL_ON);

    Old_Button_Num = new_button_num;
}

As with Set_Check_Box(), we pass to Set_Radio_Buttons() both the dialog box pointer and the item number of the DITL radio button item that was clicked on.

When Set_Radio_Buttons() ends, the global variable Old_Button_Num retains its value. So when the program returns to the Set_Radio_Buttons() function a second time, it can turn off Old_Button_Num because it will still hold the item number of the radio button to be turned off.
You just learned that you use a global variable for radio buttons. Why don’t you need a global variable to keep track of the state of a check box while the dialog is on the screen? Because when a check box is clicked on, it is always toggled. You don’t have to save its value. When a check box is clicked on, you simply determine the value at that moment and set it to the opposite state.

**Getting and Setting Edit Text Items**

The last DITL item we will examine is the edit text item. The example DITL has one edit text item, numbered 6. Figure 17–24 serves as a reminder of this.

As a programmer, you'll be happy to learn that edit text items require very little intervention on your part. When a user clicks the mouse in an edit text item and starts to type, you do not have to take any action to cause those keystrokes to be echoed back into the edit text item. You become interested when you want to know the value, or contents, of an edit text item. Use the now very familiar routine GetDItem() to get a handle to the edit text item, then call Toolbox routine GetIText() to get the inserted text from the edit text item. On the next page, we demonstrate how to get the string from the edit text item of our DITL.

---

**FIGURE 17–24** The edit text item is item number 6.
#define EDIT_BOX_DITL_ITEM 6

Str255 Speed_Str;

short item_type;
Handle item_handle;
Rect item_rect;

GetDItem(the_dialog, EDIT_BOX_DITL_ITEM, &item_type, &item_handle, &item_rect);

GetIText((ControlHandle)item_handle, &Speed_Str);

Now we'll take this code and incorporate it into a generic function that can be used to get the string from any edit box in any dialog box. You pass the routine a pointer to the dialog box that contains the edit box, the DITL item number of the edit box, and the string that will hold the text from the edit box:

Get_Text_From_Edit(DialogPtr the_dialog, short edit_item, Str255 *the_string)
{
    short item_type;
    Handle item_handle;
    Rect item_rect;

    GetDItem(the_dialog, edit_item, &item_type, &item_handle, &item_rect);

    GetIText((ControlHandle)item_handle, the_string);
}

There are two observations that should be made about the third argument to GetIText(). First, the argument is always a Str255 type—a string variable. Even if the user types in a number, as shown in the example, you read the contents as a string. It is up to you to determine whether the value read should be left as a string or converted to a number. If you want to use the entered value as a number, use StringToNum() after retrieving the string. Here's an example using Handle_Dialog():
#define DONE_BUTTON_DITL_ITEM 1
#define EDIT_BOX_DITL_ITEM 6

Str255 Speed_Str;
long Speed_Int;

Handle_Dialog()
{
...
...
...
ModalDialog(NIL_PTR, &the_item);
switch (the_item)
{

case DONE_BUTTON_DITL_ITEM:
    Get_Text_From_Edit(the_dialog, EDIT_BOX_DITL_ITEM, Speed_Str);
    StringToNum(Speed_Str, &Speed_Int);
    done_with_dialog = TRUE;
    break;
...
...
...
}

Notice that the second argument in StringToNum() must be a pointer to a long rather than an int. The StringToNum() routine is capable of converting a string to a rather large number. We give you another example of the conversion of a string to an integer in Chapter 20.

The second thing to notice about the second argument to GetIText() is its scope—Speed_Str is declared as a global variable. This is not a requirement. But if you want to preserve the string that was in the edit box after the dialog box is dismissed, you must store it in a global variable. An example would be a dialog box that contained an edit box that allowed the user to type in his or her name. When the user clicked on the Done or OK button, you would want to save the name for use later in the program. As a matter of fact, the Simulator software that accompanies this book performs this very trick. The Simulator C menu contains a Configure... option that brings up a dialog box. You can type your name in an edit text item and
your name will be used in the Question Feedback section of Question Pages long after the Configure dialog box is dismissed.

**Getting Final Values of DITL Items**

A very common practice of programmers creating dialog boxes is to let the user click the mouse and type as often he wants—toggling check boxes, switching radio buttons, and typing in edit boxes. Only when the user is done do you go after the final values of all of these items.

As we showed you in our Handle_Dialog() routine, when this happens, you call DisposDialog() to remove the dialog box from the screen. (Note that there is no e in DisposDialog().) But first, you retrieve all information that is available. This information is in the form of values—check box values, radio button values, and the text from edit text items. You are already familiar with all the techniques you need in order to retrieve the final values of these item types. For clarity you can package it all together in one function. The example we'll show you is named Get_Dialog_Info(). Call this function when the user is finished using the dialog box. In the example, it is assumed that a click on the single push button item in the dialog box is the signal to set the done_with_dialog variable to TRUE and to dismiss the dialog box. That is the time to call this Get_Dialog_Info() function from within Handle_Dialog():

```c
while (done_with_dialog == FALSE)
{
    ModalDialog(NIL_PTR, &the_item);
    switch (the_item)
    {
    case DONE BUTTON DITL ITEM:
        Get_Dialog_Info(the_dialog);
        done_with_dialog = TRUE;
        break;
    ....
    }
}
```

Let's take a look at the Get_Dialog_Info() function. It was written for the example DITL that has been used throughout this chapter. Comments are included after the function listing:
#define DONE_BUTTON_DITL_ITEM 1
#define CHECK_BOX_DITL_ITEM 2
#define RADIO_1_DITL_ITEM 3
#define RADIO_2_DITL_ITEM 4
#define EDIT_BOX_DITL_ITEM 6

Boolean Save_Speed_Limit; /* Global variables */
Boolean Raise_Speed_Limit;
Str255 Speed_Str;
long Speed_Int;

Get_Dialog_Info(DialogPtr the_dialog) {
    Handle item_handle;
    short item_type;
    Rect item_rect;
    int cntl_value;

    GetDItem(the_dialog, CHECK_DITL_ITEM, &item_type,
             &item_handle, &item_rect);
    cntl_value = GetCtlValue((ControlHandle)item_handle);

    if (cntl_value == CONTROL_ON)
        Save_Speed_Limit = TRUE;
    else
        Save_Speed_Limit = FALSE;

    switch (Old_Button_Num)
    {
        RADIO_1_DITL_ITEM:
            Raise_Speed_Limit = TRUE;
            break;
        RADIO_2_DITL_ITEM:
            Raise_Speed_Limit = FALSE;
            break;
    }

    Get_Text_From_Edit(the_dialog, EDIT_BOX_DITL_ITEM,
                        Speed_Str);
Most of the Get_Dialog_Info() code should be familiar to you. We'll comment on a few key points here. The first thing the function does is get a handle to the check box and then determine the check box value. The next step demonstrates the usual means of making use of the result of a check box value—set a global variable to TRUE or FALSE based on the check box value:

```c
if (cntl_value == CONTROL_ON)
    Save_Speed_Limit = TRUE;
else
    Save_Speed_Limit = FALSE;
```

With a global Boolean variable set, you can base a decision on this variable's value at any point in the program. At this point, radio buttons are the easiest of the DITL items to work with—you already know the final value of the radio button that is on. Remember, in a set of radio buttons, only one is on at a time. You know the DITL item number of the radio button that is currently on—you've saved it in the global variable Old_Button_Num. Because it's a global variable, you do not have to use its value right away. But since you're handling everything else related to DITL values here, you'll set any global variables that are dependent on the radio button value now.

In this example, set the global variable Raise_Speed_Limit to either TRUE or FALSE:

```c
switch (Old_Button_Num)
{
    RADIO_1_DITL_ITEM:
        Raise_Speed_Limit = TRUE;
        break;
    RADIO_2_DITL_ITEM:
        Raise_Speed_Limit = FALSE;
        break;
}
```

What do you think you'd do if you had more than two radio buttons in a group? A Boolean variable can only have one of two values, so it obvi-
The solution requires simply that you choose a global variable of a type other than Boolean. Because the speed limits in the above example are all integers, you could create a global int variable named New_Speed_Limit, for example:

```c
int New_Speed_Limit;
....
....
switch (Old_Button_Num)
{
    RADIO_1_DITL_ITEM:
        New_Speed_Limit = 40;
        break;
    RADIO_2_DITL_ITEM:
        New_Speed_Limit = 55;
        break;
    RADIO_3_DITL_ITEM:
        New_Speed_Limit = 70;
        break;
    RADIO_4_DITL_ITEM:
        New_Speed_Limit = 100;
        break;
}
```
End the function by getting the value in the edit text item. Remember, edit text items always contain strings. So retrieve the text in a global Str255 variable:

```c
GetDItem(the_dialog, EDIT_BOX_DITL_ITEM, &item_type, &item_handle, 
  &item_rect);
GetIText(item_handle, &Speed_Str);
```

It's been a long road, but you've covered a lot of useful material along the way. You now know how to do all of the following:

- Create both a DLOG and a DITL using ResEdit
- Write the C code to bring a dialog box to the screen
- Monitor and react to the user's actions in your dialog box
- Determine the values of dialog items
- Dismiss a dialog box

Now that you have a good handle on dialog boxes, you'll need to read only a few pages in order to understand the next topic—alerts.

### Alerts: Stripped Down Dialog Boxes

The simplest of all dialog boxes is the alert. The job of the alert is to relay a message—usually a warning that the user may be about to make a mistake. Very little goes on in an alert. Usually, the user gets just one or two choices—to cancel an impending action or to continue. These choices are in the form of push buttons. Check boxes and radio buttons are not allowed in alerts. There are four types of alerts, and all work in the same way. The only difference between the types is the icon that appears in the upper-left corner of the alert. Figure 17–26 shows an example of each type.

### Alerts and Resources

An alert is a simplified dialog box. So, like a dialog box, it requires a DITL. You first use ResEdit to make the ALRT, then to make its DITL.
FIGURE 17–26 The alert types differ only in the icons that represent them.

**ALRT Resource**

From within ResEdit, select Create New Resource from the Resource menu of ResEdit. Click on the ALRT type from the list in the Select New Type dialog box.

An alert editing window will open. The editing window is plainer than that of a DLOG editing window. Figure 17–27 shows an example.

You need do nothing but set the size of the alert and the ID of the DITL that will contain the push buttons and static text items that will appear in the alert.

You’re now ready to create the corresponding DITL resource. Click on the close box of the ALRT window.

**DITL Resource**

Select Create New Resource from the Resource menu of ResEdit. The Select New Type dialog box will open. As you did for your DLOG’s DITL, click on the DITL type, then click the OK button.
Once the DITL editing window opens, add a single push button to it. Add a static text item as well. Look back at Figure 17–26 for some ideas. As long as you have at least one push button for dismissing the alert, you can consider your alert resource complete. Select Save from the File menu, then quit ResEdit.

Creating an Alert in C

Putting up an alert is one of the easiest of all Macintosh tasks. Don’t let your mind wander as you read the next few paragraphs—you could miss it completely:

```
#define MY_ALERT 128
#define NIL_PTR 0L
Alert(MY_ALERT, NIL_PTR);
```
We've shown you many code fragments in this book—the alert routine is not one of them. The Toolbox routine Alert() is really all the code you need to put up an alert.

The Alert() routine waits for the user to click the mouse on any enabled item in it. Remember, any DITL item can be either enabled or disabled. Push buttons are enabled, so a user click on a push button will dismiss the alert—you don't, and can't, take any action while the alert is on the screen.

The Alert() routine puts up an alert with no icon in it. Macintosh C offers three variations of the Alert() function that do put icons in the alert for you. Each has the same arguments as Alert(). The following code puts up the same alert, ID 128, three times. Each time it displays a different icon:

```
NoteAlert(MY_ALERT, NIL_PTR);
CautionAlert(MY_ALERT, NIL_PTR);
StopAlert(MY_ALERT, NIL_PTR);
```

### Chapter Summary

Dialog boxes are the Macintosh's way of communicating with a program user. Dialog boxes can be modal—fixed on the screen—or modeless—movable.

Dialog boxes contain items. The most common are the push button, radio button, check box, edit box, and static text items. Other item types are the icon, picture, and user item.

ResEdit is used to create the resources needed for a dialog box. The resource that defines the size and type of a dialog box is the DLOG resource. A DLOG also contains the ID of another resource type, the DITL. The DITL defines all of the items that appear in a dialog box. Each DLOG is thus paired with a DITL.

Within a DITL are the check boxes, push buttons, and other items that appear in the dialog box. As you add an item to a DITL, you keep track of the item number associated with it. Later, within your C source code, you can refer to a specific dialog item by this number.

A dialog box must be loaded into memory when needed. The Toolbox routine GetNewDialog() does this for you. You specify the ID of the DLOG you want, and the Toolbox loads it into memory and returns a
pointer to the dialog box. This pointer, of DialogPtr type, can then be used whenever you want to refer to the dialog box or the items in it.

Once a dialog box is in memory, you can show it using ShowWindow() or hide it using HideWindow(). To draw text or shapes to it, first use SetPort() to tell the Macintosh that the graphics port of this dialog box is to be the current port. Then use any QuickDraw routine, such as DrawString(), MoveTo(), SetRect(), or FrameRect(), to draw to it.

Once a dialog box is on the screen, you monitor the user's actions by repeatedly calling ModalDialog(). This routine tells you the DITL item number of a clicked-on item so you can take the appropriate action.

The GetDlgItem() routine is used to get a handle to an item. Once you have obtained an item's handle, you can work with that item. You can get an item's value by using GetCtlValue(). Check boxes and radio buttons have a value of 0 if they are off and 1 if they are on. You can set an item's value with SetCtlValue().

To get the text from an edit box, use the routine GetIText(). All text in an edit box is of Str255 type. If the text represents a number, you must convert it to one using the StringToNum() routine.

When you are finished with a dialog box, you should hide it using HideWindow(). Then free the memory in which it resided with a call to DisposDialog().

An alert is a stripped down dialog box. It usually contains just informational text and one or two push buttons to dismiss it. An ALRT resource is created to define the size of an alert. Like a dialog box, an alert has a corresponding DITL. Within the DITL are the static text items that provide information to the user and a push button to dismiss it.

To put an alert on the screen, you load the resource into memory using a call to Alert(). To include one of three available icons in the upper-left corner of an alert, use a call to NoteAlert(), CautionAlert(), or StopAlert() instead of the call to Alert().
In the previous 17 chapters, we've covered a wealth of C language material—both ANSI C and Macintosh C. If you've been patient, and followed along in the Simulator software, you should have a solid understanding of many of the basics of C language and Macintosh C programming. Now, you're ready to take the final step toward Macintosh mastery—creating a complete, compilable Macintosh program.

In this chapter we present the complete resource file for a Macintosh program we call ShapeMaker. This program will tie together the concepts to which you've been exposed so far—the event loop, the dialog box, the items that go in a dialog box, and the resources. Because this is the first complete program we've presented to you, we're going to go through the creation of the resource file in a thorough, step-by-step manner.

Here we'll introduce the ShapeMaker program to you and show you what it does. Then we'll show you how to create all the resources you'll need for it—DLOG, DITL, ALRT, MENU, and MBAR types.

In the next chapter we'll walk through the ShapeMaker source code.
The ShapeMaker Program

The disks that come with this book contain a folder called THINK C Examples. You may have examined it during your reading of Chapter 14. One of the folders in the THINK C Examples folder, ShapeMaker Folder, contains the source code, resource file, project file, and ready-to-run application for ShapeMaker Program.

Run ShapeMaker Program from the ShapeMaker Folder to see what you'll be creating in this and the next chapter. Once you have tried it, you'll want to start from scratch so you can practice creating an entire program. We'll guide you each step of the way.

If you run our version of the ShapeMaker program, you'll see a blank desktop with a menu bar containing two menus—the Apple menu and the File menu. If you click on the Apple menu, the pull-down menu that appears will look something like the one shown in Figure 18-1, but the names of your desk accessories will be different.

Selecting a desk accessory opens that desk accessory. Selecting the menu item titled About ShapeMaker... will bring up an alert like the one shown in Figure 18-2. Clicking on the Back to the Program button dismisses the alert.

The other ShapeMaker menu is the File menu. If you click on the File menu, the pull-down menu will look like the one shown in Figure 18-3.

Selecting Open ShapeMaker brings up the ShapeMaker dialog box. In this dialog box are two buttons, one edit box, and four check boxes. Figure 18-4 shows the ShapeMaker dialog box.

The ShapeMaker dialog box allows the user to type in a title for a picture in the edit box and check any combination of the four check boxes. When the user clicks on the Draw button, the shapes that were checked will
ShapeMaker
from
Prima Publishing's
"Think THINK C"
copyright © 1993 Dan Parks Sydow

FIGURE 18-2  The About ShapeMaker... alert

FIGURE 18-3  The File menu of ShapeMaker

FIGURE 18-4  The ShapeMaker dialog box
be drawn to the bottom half of the dialog box. Figure 18–5 shows the ShapeMaker dialog box after check boxes have been checked and the Draw button has been clicked on.

We've decided to use this program as our first complete program because it shows off many of the programming techniques you'll want to use in your own programs—yet the code is relatively short and straightforward. After studying the code for ShapeMaker, you'll know how to do all of the following:

- Write the basic event-driven code common to all Macintosh programs
- Create a main menu bar, complete with functioning pull-down menus, including the apple desk accessory menu
- Open a dialog box that contains items such as buttons, an edit box, and check boxes
- Track and respond to the mouse as the user clicks on items in the dialog box
- Read the values and text in the items in the dialog box and take appropriate action
- Draw shapes and text to the dialog box using QuickDraw commands
- Use ResEdit to create a resource file with all of the resources necessary to do everything listed above
- Create a stand-alone Macintosh program that you can use, copy, modify, and give to friends!

Now that you have a good idea of what the ShapeMaker program does, let's start by stepping through the creation of the ShapeMaker's resource file using ResEdit.

The ShapeMaker Resource File

If you're unfamiliar with ResEdit, now is the time to go back and read through Chapter 13. There we give the basics for creating a resource file and show you how to create resources. Chapter 17 shows you how to make DLOG, DITL, and ALRT resource types. Chapter 15 describes the MBAR and MENU resources. You'll use all of these types in this program.

Creating the Resource File

Begin by creating a folder called MyShapeMaker Folder. If you have a Development folder on your hard drive, put the MyShapeMaker Folder in it. You'll keep your THINK C ShapeMaker project, resource file, and source code in this folder when you create them. We have our folders set up as shown in Figure 18-6; you might want to try a similar configuration.

FIGURE 18-6 A typical folder setup for THINK C and ShapeMaker
Now you will create the resource file. Begin by running ResEdit. Follow the four steps below to create a resource file. This process is described fully in Chapter 14:

1. Click on the Introductory Dialog.
2. Make sure you're in the MyShapeMaker Folder.
3. Click on the New button.
4. Type in the name ShapeMaker.π.rsdc and click on the New button.

Your new resource file starts out empty. The next several pages will show you how to add resources to it, starting with a DLOG resource.

The DLOG Resource

You'll create your DLOG resource by following these two steps. The creation of a DLOG is covered in Chapter 17:

2. Choose the DLOG type; then click the OK button.

In the dialog window that opens, set the features of the dialog box to match those shown in Figure 18–7 as follows:

![DLOG window](image)

**FIGURE 18–7** The DLOG window
1. Change the dialog box type to dBoxProc by clicking on the appropriate icon.
2. Set the size of the dialog box by filling in the four edit boxes. Make the box small enough to fit on any Mac screen.
3. Uncheck the close box check box—your dialog box will have no close box.

After making these changes, you're ready to give the resource a name. Follow these steps:

1. Close the DLOG window by clicking on its close box.
2. If it's not already highlighted, highlight the informational line for DLOG 128 by clicking once on it in the window that lists the DLOGs (you have only one DLOG).
3. Select Get Resource Info from the Resource menu. See Figure 18–8.

In the Info window that opens, type a DLOG name. The name will be for your information only—it will not actually appear on the dialog. Figure 18–9 illustrates the naming of your DLOG resource.

At this point you're done with the DLOG. Close both the Info window and the window that lists your one DLOG.
FIGURE 18–9 Naming a DLOG

The DITL Resource

Every DLOG has items in it. Your next task is to create the DITL that holds the items that are to be displayed in the DLOG you just created. Create a DITL resource by selecting Create New Resource from the Resource menu and then choosing the DITL type. Click the OK button.

Follow these steps to add a button to the DITL so that it matches the one shown in Figure 18–10:

FIGURE 18–10 Adding a button item to your DITL
1. Enlarge the empty DITL window to fill much of your screen.
2. In the floating items menu, click on the button icon and drag it over to the DITL window. Place it as shown in Figure 18-10.
3. Double-click on the new button; then type Draw in the editing window that opens. Verify that the Enabled box is checked. This is shown in Figure 18-11. Close the window.
4. Repeat Steps 2 and 3 for each of the items, selecting an item, placing it in the window, then double-clicking on it so that you can name and position it. The editing windows for each of the eight remaining DITL items are shown in Figure 18-12.
5. Note in Figure 18-12 that only the last two items, the static text items, do not have their Enabled boxes checked.

When done, your DITL should look like the one in Figure 18-13.

Each of the DITL items has a number, starting with item #1. It is important to note and make a list of the DITL item numbers, because you will refer to specific DITL numbers from within your C language source code when you write it. To make your DITL numbers match ours, select Renumber Items... from the DITL menu.

When you make this menu selection, each item will appear within a rectangular border. A number will appear in the upper-right corner of each. This is the item number. To change the numbering, place the cursor over the item you want to be item #1. While holding the [Shift] key down, click on the item. Make the Draw button item #1. Go from item to item, keeping the [Shift] key held down, clicking on each. If you make a mistake,
FIGURE 18-12 The editing windows for the remaining DITL items
Figure 18-12 The editing windows for the remaining DITL items (cont.)
simply start over by releasing the [Shift] key, pressing it again, and clicking on item #1. When done, click on the Renumber button. The correct item numbering is shown in Figure 18–14.

With the items appropriately placed and numbered, the contents of your DITL are complete. Close the DITL. Your screen should show a window that looks like the one in Figure 18–15.
The ALRT Resource

The ShapeMaker program, like most Macintosh programs, displays a small dialog box when the user selects the About ShapeMaker... item from the menu. Because we just want to put up a simple box displaying the program name and author, we've chosen the simplest dialog box to work with—the alert. Here you will create the ALRT resource your program will use.

To create the ALRT resource, select the Create New Resource from the Resource menu and then choose the ALRT type. In the window that opens, set the alert's features to match those shown in Figure 18–16:
1. Set the size of the alert by filling in the four edit boxes. Make the alert small enough to fit on any Mac screen.

2. Change the DITL ID from 128 to 200.

Why do we recommend that you change the value in the DITL ID edit box? Like a DLOG, an ALRT has an item list. The DITL ID for your DLOG has an ID of 128, and you don’t want the ALRT to use the same DITL. So when you make the DITL for this ALRT, you’ll give it an ID other than 128—we arbitrarily chose 200.

Because our convention is to give a DITL an ID that matches the ID of the DLOG or ALRT associated with it, we want you now to change the ID of the ALRT. Follow these steps:

1. Close the ALRT window by clicking on its close box.
2. Highlight the informational line for ALRT 128 by clicking once on it in the window that lists the ALRTs.
4. In the Info window that opens, type our new ID of 200 and give the ALRT a name. Because this alert will open when the user selects the About ShapeMaker… menu item, pick a descriptive name like About ShapeMaker Alert.

Figure 18–17 illustrates Step 4, the renumbering and naming of your ALRT resource. Now your ALRT has the same ID, 200, as the DITL that will soon be associated with it.
You're now done with the ALRT. Close both the Info window and the window that lists your one ALRT.

One More DITL Resource

Every DLOG has items in it. Every ALRT does as well. You created one DITL for your DLOG. Now you'll want to create one for your ALRT. The process is exactly the same, so we'll go through it a little quicker. Follow these steps:

1. Create the DITL resource by selecting Create New Resource from the Resource menu and then choosing the DITL type. Click the OK button.
2. Add the DITL items from the floating items menu. Use one button for the Back to the Program button. The rest of the items are all static text items. The positioning of the items is not critical, but keep them confined to a relatively small area.
3. Only the DITL item number of the Back to the Program button is crucial—it must be item #1. Select Renumber Items... from the DITL menu. Place the cursor over the Back to the Program button, hold down the s key and click the mouse button.
4. Click on the Renumber button. Verify that the Back to the Program button has a number 1 in its upper corner.

After you have followed this sequence, your screen should look like the one shown in Figure 18–18.

The last step is to give the DITL an ID that matches the ALRT with which it will be associated:

1. Close the DITL window by clicking on its close box.
2. Highlight the informational line for DITL 128 by clicking once on it in the window that lists the DITLs (you now have two DITLs).
4. In the Info window that opens, type our new ID of 200 and give the DITL a descriptive name like About Alert Dialog Item List.

You're now done with the second DITL. Close both the Info window and the window that lists the two DITLs.

The MENU Resource

In this section you will create the two menus that will appear in the ShapeMaker’s main menu bar. To create a MENU resource, select Create New
Resource from the Resource menu, then choose the MENU type. Click the OK button. One window will open, as shown in Figure 18–19.

Use this for the Ⓞ menu. Click in the radio button titled Ⓞ (Apple menu) to give the menu the Ⓞ title. To add a menu item, select Create New
Item from the Resource menu. Type in the menu item’s title, About ShapeMaker....

To add the dashed separator line that will appear between the About ShapeMaker... menu option and the desk accessories in the Apple menu, again select Create New Item from the Resource menu. Click on the radio button labeled (separator line).

You’re done with this menu. Click on the close box of the MENU window. Your screen should look like Figure 18–20. Does it seem like something is missing in Figure 18–20? Where are the desk accessories? The adding of desk accessories is not part of the process of creating the Apple menu. You’ll see how desk accessories are added in Chapter 19.

The ShapeMaker program will have two menus in its menu bar—the Apple menu and the File menu. Now you will create the File MENU resource. Select Create New Resource from the Resource menu. In the window that opens, type File in the Title edit box.

To add a menu item, select Create New Item from the Resource menu. Type in Open ShapeMaker, as shown in Figure 18–21.

To add a second menu item, again select Create New Item from the Resource menu. Type in Quit.

ResEdit gave the first MENU an ID of 128 and this MENU an ID of 129. You can leave these IDs as they are.
IMPORTANT

To change the ID of most types of resources, you just select Get Resource Info from the Resource menu. If you want to change the ID of a MENU resource, you must perform one additional step. You must also change the ID in the Edit MENU & MDEF ID... from the MENU menu.

Click on the close box of the MENU window. Your screen should look like Figure 18–22.
Close the window that lists your two DITLs. Your screen should look like the one shown in Figure 18–23.

**The MBAR Resource**

The two MENU resources will be in the same menu of ShapeMaker—the main menu bar. To group them together, you must create an MBAR resource. To create an MBAR resource, select Create New Resource from the Resource menu and then choose the MBAR type. Click the OK button. A window will open, as shown in Figure 18–24.
Click on the row of asterisks—it will become enclosed in a rectangle. You want to add the IDs of the MENU resources that are to appear in the MBAR. To add a MENU ID, select Insert New Field(s) from the Resource menu. An edit box titled Menu res ID will appear. Type in the number 128, the ID of the MENU resource. Figure 18–25 illustrates this.

To add the File MENU resource to the MBAR, repeat the steps performed for the first MENU ID:

1. Click on the row of asterisks by the 2.
2. Select Insert New Field(s) from the Resource menu.
3. Type the number 129, the MENU ID of the File menu, in the edit box that appears.

Click on the close box of the MBAR window. You're now done with the MBAR and all of the other resources you'll need. Close the window that lists the MBAR. Your screen should look like the one shown in Figure 18–26. Be sure to select Save from the File menu; then Quit ResEdit.

FIGURE 18–25 Typing in the MENU ID

FIGURE 18–26 The ShapeMaker resource file is complete.
You've created all the resources ShapeMaker will need. Now, you're ready to write the source code you'll need for the ShapeMaker program—so it's on to Chapter 19.

Chapter Summary

A Macintosh program is made up of resources and source code. This chapter demonstrated the steps necessary to create a resource file for an application called ShapeMaker.

ResEdit is the program used to create a resource file and to add items to it.

The next chapter illustrates how to merge a resource file with your source code to create a stand-alone Macintosh program.
Complete Source Code—ShapeMaker

In the previous chapter you created the complete resource file for your own version of the ShapeMaker program. In this chapter we'll walk through the ShapeMaker source code, section-by-section, explaining the programming decisions we've made at each step.

The ShapeMaker program covers many of the Macintosh programming concepts covered in this book. After examining all of the source code, you'll compile the program into your first stand-alone, executable Macintosh program!

The ShapeMaker Program

In the previous chapter we introduced and explained the ShapeMaker program. To recap, ShapeMaker opens a dialog box that allows the user to type in a title for a picture, check any combination of four check boxes, and then click on a Draw button. When the Draw button is clicked on, the shapes that were checked are drawn to the bottom half of the dialog box. Figure 19–1 shows the result of this sequence.
We've included the ShapeMaker application in the THINK C Examples Folder of one of the disks that accompany this book. If you haven't seen the program yet, you might want to run it now to become familiar with it before examining the source code.

**Function Prototypes**

The ShapeMaker source code, like most C language programs, contains several functions. As we noted in Chapter 8, some C compilers require you to list prototypes for all of your functions at the start of your source code. This list tells the compiler what to expect if it encounters a function call before it sees the function defined. Here's an example of a function prototype:

```c
int A_Function(void); /* function prototype */
```

The prototype tells the compiler what to expect of the function—the return type, the function name, and the types of passed parameters. The same function prototype appears below as it would be used in a C program:
int A_Function(void); /* function prototype */

main()
{
    int x;
    ... 
    x = A_Function(); /* calling the function */
}

int A_Function(void) /* defining the function */
{
    /* body of function */
}

The first line of this code contains the prototype for the function A_Function(). It first gives the function return value, int. Next comes the function name, A_Function. Finally, the types of the passed parameters are given. Because there are none, we use the C keyword \texttt{void}. Chapter 8 provides more information on function prototypes.

To form good programming habits and to help out the THINK C compiler, we’ll list prototypes for each function in ShapeMaker.

\section*{Stepping through the Code}

The remainder of this chapter is devoted to a thorough examination of the source code. Be sure you understand each step—you’ll repeat most of them every time you write a Macintosh program.

\section*{The Function Prototypes}

The first order of business in the source code is to list the function prototypes—right up front for the THINK C compiler to see first.

\begin{verbatim}
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Initialize_Toolbox( void );
void Set_Up_Menu_Bar( void );
void Main_Event_Loop( void );
void Handle_One_Event( void );
void Handle_Mouse_Down( void );
\end{verbatim}
void Handle_Menu_Choice( long);
void Handle_Apple_Choice( short);
void Handle_File_Choice( short);
void Open_ShapeMaker_Dialog( void);
void Set_Check_Box( DialogPtr, short);
void Get_Dialog_Info( DialogPtr );
void Get_Text_From_Edit( DialogPtr, short, Str255 );
void Draw_Shapes_In_Dialog( void );

The #define Statements

The ShapeMaker code starts with a couple dozen #define statements. Throughout this book we've emphasized the use of symbolic constants—words that represent, or symbolize, numbers. Here, we follow our own advice.

We'll cover each #define as we encounter it in the code. But we'll discuss the first one now to emphasize the importance of #defines.

In the DITL resource you created in ResEdit, you gave each DITL item a number. You set the button you labeled Draw to DITL item #1. If you ever have to use the DITL item in your code (and of course, you will), you refer to it by its item number. In the code, the name DRAW_BUTTON_DITL_ITEM is defined as equal to the number 1. Whenever you want to refer to the Draw button, you use the word DRAW_BUTTON_DITL_ITEM instead of the number 1.

When you compile the code, the THINK C compiler will substitute the number 1 for every occurrence of the word DRAW_BUTTON_DITL_ITEM. If the compiler is going to change each DRAW_BUTTON_DITL_ITEM to a 1, why don't you just type in the number 1 each time yourself? The answer has to do with program maintenance. There may come a time when you want to change the items in your DITL in the resource file. You might want to add more features to the program, and this might require more items—and a change in the numbering of the DITL items. The item number of the Draw button might change from a 1 to a 2. If this happens, you must update your source code. Every time you refer to the Draw button in your code, you must change the reference from a 1 to a 2. But since you were wise enough to use a #define, you can just change your #define from

#define DRAW_BUTTON_DITL_ITEM 1

to

#define DRAW_BUTTON_DITL_ITEM 2
Then you compile your program again and let the compiler do all the substituting. It will search out every reference to DRAW_BUTTON_DITL_ITEM and replace it with a 2—the new DITL item value of the Draw button.

It might seem like a little extra work to type the name DRAW_BUTTON_DITL_ITEM instead of the number 1, but as you begin to write larger and larger programs, these few extra keystrokes will pay off.

If you recall your creation of the program’s resource file, many of the numbers in the following #define statements will look familiar:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
#define DRAW_BUTTON_DITL_ITEM 1
#define DONE_BUTTON_DITL_ITEM 2
#define TITLE_DITL_ITEM 3
#define FRAME_RECT_DITL_ITEM 4
#define FILL_RECT_DITL_ITEM 5
#define FRAME_OVAL_DITL_ITEM 6
#define FILL_OVAL_DITL_ITEM 7
#define SHAPE_DLOG_ID 128
#define ABOUT_ALRT_ID 200
#define MAIN_MBAR_ID 128
#define APPLE_MENU_ID 128
#define FILE_MENU_ID 129
#define ABOUT_ITEM 1
#define OPEN_ITEM 1
#define QUIT_ITEM 2
#define CONTROL_ON 1
#define CONTROL_OFF 0
#define NIL_PTR 0L
#define IN_FRONT (WindowPtr)-1L
#define REMOVE_EVENTS 0
#define WNE_TRAP_NUM 0x60
#define UNIMPL_TRAP_NUM 0x9F
```
The Global Variables

After the #defines come the program’s global variables. Recall that our convention is to start each word of a global variable with an uppercase letter and each word of a nonglobal variable with a lowercase letter. This is our convention—you are not bound to it. But no matter how you choose to name variables, remember that C is case-sensitive. See Figures 19–2 and 19–3.

The ShapeMaker global variables are listed on the next page.

![Diagram](image)

**FIGURE 19–2** Our variable naming convention

![Diagram](image)

**FIGURE 19–3** In C, upper- and lowercase matter!
The variable Title_Str holds the title of the picture. The variables Framed_Rect, Filled_Rect, Framed_Oval, and Filled_Oval keep track of whether or not these four shapes should be drawn. The variable All.Done determines when the program is finished. The variable Apple_Menu is a handle to the Apple menu, and File_Menu is a handle to the File menu. The variable Multifinder_Active tells the program if the user has MultiFinder running. The last global variable, The_Event, is an EventRecord that holds details about the most recent event.

The main() Routine

ShapeMaker's first routine is the main() function. The main() function calls three other functions. The first is Initialize_Toolbox(), which groups together all the necessary Toolbox initializations. Next, the main menu bar is set up by calling Set_Up_Menu_Bar(). Finally, the main event loop is entered by calling the function Main_Event_Loop(). Keep in mind that all three of these functions were written by us—they are not Toolbox functions. See Figure 19–4 for another naming convention of ours. We'll cover each of the three called routines as we encounter them:

main()
{
    Initialize_Toolbox();
    Set_Up_Menu_Bar();
    Main_Event_Loop();
}
Toolbox routines contain no underscores

Give your own routines underscores

Program Initialization

We introduced a Toolbox initialization routine in Chapter 11. We use this same routine in all of our Macintosh programs, and we suggest that you do the same. All Mac programs must make the calls that are in this routine in exactly this order. Here we encounter two of the #defines—NIL_PTR and REMOVE_EVENTS. They are used as passed parameters. These #defines are more descriptive than OL and 0:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Initialize_Toolbox( void )
{
    InitGraf(&thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NIL_PTR);
    FlushEvents(everyEvent, REMOVE_EVENTS);
    InitCursor();
}
```
Setting up the Menu

The second routine called by main() is Set_Up_Menu_Bar(). Here you finally get to see the interaction between source code and resources. This routine loads the MBAR resource and the two MENU resources into memory. The Toolbox routine GetNewMBar() loads your MBAR into memory. SetMenuBar() lets the program know which menu bar is the current one. This would be especially important if you had more than one MBAR (which you don't). Call the Toolbox routine GetMHandle() twice to load your two menus into memory and get handles to them. When you have to work with these menus, handles give you a way to access them. The Toolbox routine AddResMenu() adds the names of all of the available desk accessories to the Apple menu. Finally, the Toolbox function DrawMenuBar() displays the main menu bar on the Mac's screen:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Set_Up_Menu_Bar( void )
{  
    Handle main_menu_bar;
    
    main_menu_bar = GetNewMBar(MAIN_MBAR_ID);
    SetMenuBar(main_menu_bar);

    Apple_Menu = GetMHandle(APPLE_MENU_ID);
    File_Menu = GetMHandle(FILE_MENU_ID);

    AddResMenu(Apple_Menu, 'DRVR');

    DrawMenuBar();
}
```

The Main Event Loop

The last routine called by main() is Main_Event_Loop(). Here the variable All_Done is set to FALSE—we've only just begun! All_Done will become TRUE when the user quits the program.

Next, check to see if the Macintosh on which the program is running has MultiFinder active. This is a little tricky—you might want to just take our word for it when we tell you that this is a valid check. When you use the Toolbox routine NGetTrapAddress(), what you're doing is looking
directly into the system to see if a certain Toolbox routine—WaitNextEvent()—has been loaded into memory. If it's there, MultiFinder is running.

After the MultiFinder check, the program enters a while loop that repeats until the user quits the program. The while loop's only purpose is to call the Handle_One_Event() function:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Main_Event_Loop( void )
{
    All_Done = FALSE;

    Multifinder_Active = (NGetTrapAddress(WNE_TRAP_NUM, ToolTrap)
                        != NGetTrapAddress(UNIMPL_TRAP_NUM, ToolTrap));

    while (All_Done == FALSE)
        Handle_One_Event();
}

Handling a Single Event

The next routine, Handle_One_Event(), is repeatedly called by the program's main event loop. In this routine, the event that occurred most recently is examined—you want to determine what it was so that it can be handled properly. Get the event by calling one of two Toolbox routines—WaitNextEvent() or GetNextEvent(). Which one you call depends on whether or not MultiFinder is installed. This routine was developed in Chapter 14. Here it is scaled down to handle the only event type of interest to ShapeMaker—a mouseDown event. The ShapeMaker program cares only about mouse clicks, nothing else. We left the other cases in the switch to allow you to get a feel for the operation of this switch statement:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Handle_One_Event( void )
{
    if (Multifinder_Active == TRUE)
        WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS,
                      MOUSE_REGION);
```
else
{
    SystemTask();
    GetNextEvent(everyEvent, &The_Event);
}

switch (The_Event.what)
{
    case mouseDown:
        Handle_Mouse_Down();
        break;
    case keyDown:
        break;
    case autoKey:
        break;
    case updateEvt;
        break;
}

**Handling a mouseDown Event**

Once an event has been determined to be a mouse click, the program ends up at this routine—Handle_Mouse_Down(). This is the first function that makes use of local variables. Once this function is exited, the three local variables cease to exist.

Determine where the mouse click occurred by calling the Toolbox function FindWindow(). Pass it a field of the event record, The_Event.where. If a mouse click occurred in a window, FindWindow() returns a pointer to the window in the which_window variable. The FindWindow() routine returns a number that tells what was clicked on—the contents of a window, a menu, the desktop, etc. This value is assigned to the_part. At this point ShapeMaker is concerned only with mouse clicks that occur in the program's menu bar or in a desk accessory.

For mouse clicks in the menu bar, the Toolbox function MenuSelect() is called to track the mouse until the user releases the mouse button. The program then responds by calling routine Handle_Menu_Choice().

To handle mouse clicks in a desk accessory, the Toolbox routine SystemClick() is called:
void Handle_Mouse_Down( void )
{
    WindowPtr which_window;
    short the_part;
    long menu_choice;

    the_part = FindWindow(The_Event.where, &which_window);

    switch (the_part)
    {
    case inMenuBar:
        menu_choice = MenuSelect(The_Event.where);
        Handle_Menu_Choice(menu_choice);
        break;
    case inSysWindow:
        SystemClick(&The_Event, which_window);
        break;
    case inDrag:
        break;
    case inGoAway:
        break;
    case inContent:
        break;
    }
}

Handling a mouseDown in a Menu

Handle_Menu_Choice() handles a mouse click in a menu. In the Handle_Mouse_Down() routine, the Toolbox routine MenuSelect() was called. It returned a value to the long variable menu_choice. As discussed in Chapter 15, the Toolbox embeds both the menu (File or File) and the menu item selected within this long. One of the jobs of Handle_Menu_Choice() is to decode this variable—to determine which menu item was clicked on.

A menu_choice value of 0 means no choice was made—the program is finished with this routine for now. Any value other than 0 is decoded. Decoding is accomplished with the Toolbox HiWord() and LoWord() func-
tions. A switch statement is then used to determine what routine to call next. A different menu-handling routine exists for each of the two menus. If ShapeMaker had more menus, you would have to add more cases to the switch.

The Handle_Mouse_Down() routine calls MenuSelect(), which highlights the selected menu. After branching off and performing one of the two menu-handling routines—either Handle_Apple_Choice() or Handle_File_Choice()—unhighlight the menu with a call to HiliteMenu():

```c
void Handle_Menu_Choice( long menu_choice )
{
    int the_menu;
    int the_menu_item;
    if (menu_choice != 0)
    {
        the_menu = HiWord(menu_choice);
        the_menu_item = LoWord(menu_choice);
        switch (the_menu)
        {
        case APPLE_MENU_ID:
            Handle_Apple_Choice(the_menu_item);
            break;
        case FILE_MENU_ID:
            Handle_File_Choice(the_menu_item);
            break;
        }
    HiliteMenu(0);
    }
}
```

Handling a mouseDown in the Apple Menu

The Handle_Apple_Choice() function handles selections made in the Apple menu. The passed parameter the_item tells the routine which menu item was selected. If the menu choice was the About ShapeMaker... item, put up an alert. Pass the Toolbox routine NoteAlert() the resource ID of the alert you want to display. When you developed the ShapeMaker resource file in Chapter 18, you created an ALRT resource and gave it an ID of 200. In the
The source code.

```c
#define ABOUT_ALRT_ID 200

NoteAlert( ABOUT_ALRT_ID, NIL_PTR );
```

The compiler makes a substitution...

```
ABOUT_ALRT_ID

NoteAlert( 200, NIL_PTR );
```

When the ShapeMaker program is running and NoteAlert() is encountered, the program's resources are searched for the ALRT resource with ID 200.

```
NoteAlert( 200, NIL_PTR );
```

FIGURE 19-5 The relationship of #define, NoteAlert(), and resources

#defines of the source code is one with a value of 200—the define called ABOUT_ALRT_ID. Figure 19–5 shows how the #define, the call to NoteAlert(), and the resources are tied together.

If a desk accessory was selected from the menu, first call GetItem() to get the name of the desk accessory. Be careful not to confuse this with GetDlgItem(), which is used to get a handle to a dialog item. Once you have the desk accessory name, pass it to the Toolbox routine OpenDeskAcc(). The OpenDeskAcc() routine returns an integer reference number for the desk accessory—you won't need to save or use this.

The OpenDeskAcc() routine is responsible for opening a desk accessory. The SystemClick() routine called in Handle_Mouse_Down() is responsible for responding to a user click on an open desk accessory:
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Handle_Apple_Choice( short the_item )
{
    Str255    desk_acc_name;
    int       desk_acc_number;

    switch (the_item)
    {
    case ABOUT_ITEM:
        NoteAlert(ABOUT_ALRT_ID, NIL_PTR);
        break;
    default:
        GetItem(Apple_Menu, the_item, desk_acc_name);
        desk_acc_number = OpenDeskAcc(desk_acc_name);
        break;

    }
}

Handling a mouseDown in the File Menu

If the user makes a menu selection from the File menu, the routine Handle_File_Choice() will be called. Passed parameter the_item tells the routine which of the two menu items, Open ShapeMaker or Quit, was selected. If the menu choice was the Open ShapeMaker item, the function Open_ShapeMaker_Dialog() is called to bring a dialog to the screen. If the Quit item was selected, global variable All.Done is simply set to TRUE. When the code returns to the Main_Event_Loop(), the program ends:

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Handle_File_Choice( short the_item )
{
    switch (the_item)
    {
    case OPEN_ITEM:
        Open_ShapeMaker_Dialog();
        break;
    case QUIT_ITEM:
        All.Done = TRUE;
        break;
    }
Opening and Handling a Dialog Box

If the user selects Open ShapeMaker from the File menu, the routine Open_ShapeMaker_Dialog() is called. The mission of this function is to open a dialog box and then monitor the user's actions to see if he or she clicks the mouse on one of the items in the dialog box. This is an important function, because it demonstrates how to open a dialog box, draw in it, and monitor and respond to events that occur in it. We'll examine it in detail.

Bring on the Dialog Box

After the declaration of local variables, a call to GetNewDialog() loads the DLOG resource with an ID of 128 (SHAPE_DLOG_ID) into memory. Calling ShowWindow() ensures that the dialog box will be visible on the screen, not hidden.

Set all Flags

Open_ShapeMaker_Dialog() uses several flag variables. Boolean variables are often called flags because, like waving flags, they send a signal to the program about the status, or condition, of something. The first of these is the local variable done_with_dialog. Because the dialog box has just opened, this variable is initially set to FALSE.

The global flags Framed_Rect, Filled_Rect, Framed_Oval, and Filled_Oval let the program know which of the four check boxes are checked. Initially none of the check boxes is checked, so all four flags are set to FALSE.

Drawing to a Dialog Box

Next, use the Toolbox routines GetPort() and SetPort() to save, or remember, the port that was previously active and to set the port to the ShapeMaker dialog box. Before drawing to a window or a dialog box, you should make sure that its port is the active port. These two calls will become very important when you begin to write programs that have multiple windows open on the screen. You want to direct the Toolbox to the correct window before you perform any drawing operations.

Drawing to a dialog box or window is easy: set the port to the desired dialog box using SetPort(); then use QuickDraw routines to carry out the drawing. You only have to set the port once. Then, all calls to MoveTo(), Line(), DrawString(), etc. will take place in your dialog box. The Open_ShapeMaker_Dialog() routine moves the graphics pen to the desired location with MoveTo(), then calls Line() to draw a line.
Handling the Dialog Box

Now, the heart of the procedure. To allow the user to perform more than one action in the dialog box, a while loop is used. The loop repeats over and over until the user clicks the Done button. Within the while loop is a call to ModalDialog(). This Toolbox routine monitors and intercepts mouse events, then reports them in the form of the variable the_item. Variable the_item is then used in a switch statement—the routine bases its next action on the item that was clicked on by the user.

If the user clicks in any one of the four check boxes, the check box is toggled. If it was checked, uncheck it. If it was unchecked, check it. The routine Set_Checked() takes care of this. Because all four check boxes are treated in the same manner, you need only one case section for them.

If the Draw button is clicked on, Open_ShapesMaker_Dialog() calls Get_Dialog_Info() to determine which check boxes were checked. Draw_Shapes_In_Dialog() is then called to draw the appropriate shapes to the dialog box.

If the Done button was clicked, the flag done_with_dialog is set to TRUE. When the top of the while loop is reached again, the while expression will fail and the loop will end.

Wrapping It Up

When the while loop is over, it's time to get rid of the dialog box by calling the Toolbox routine DisposeDialog(). Finally, the port is set back to the port that was active before this function was called:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Open_ShapesMaker_Dialog( void )
{
    DialogPtr the_dialog;
    short the_item;
    Handle item_handle;
    short item_type;
    Rect item_rect;
    Boolean done_with_dialog;
    GrafPtr old_port;

    the_dialog = GetNewDialog (SHAPE_DLOG_ID, NIL_PTR, IN_FRONT);
    ShowWindow(the_dialog);
```
done_with_dialog = FALSE;
Framed_Rect = FALSE;
Filled_Rect = FALSE;
Framed_Oval = FALSE;
Filled_Oval = FALSE;

GetPort(&old_port);
SetPort(the_dialog);

MoveTo(0, 130);
Line(500, 0);

while (done_with_dialog == FALSE)
{
    ModalDialog(NIL_PTR, &the_item);
    switch (the_item)
    {
    case FRAME_RECT_DITL_ITEM:
    case FILL_RECT_DITL_ITEM:
    case FRAME_OVAL_DITL_ITEM:
    case FILL_OVAL_DITL_ITEM:
        Set_Check_Box(the_dialog, the_item);
        break;
    case DRAW_BUTTON_DITL_ITEM:
        Get_Dialog_Info(the_dialog);
        Draw_Shapes_In_Dialog();
        break;
    case DONE_BUTTON_DITL_ITEM:
        done_with_dialog = TRUE;
        break;
    }
}
DisposDialog(the_dialog);

SetPort(old_port);
Setting a Check Box

The ShapeMaker program responds to a mouse click in any of its check boxes by calling the Set_Check_Box() routine. Pass this routine a pointer to the dialog box you're dealing with and the item number of the clicked-on item. This function appears exactly as it was developed in Chapter 17.

To gain access to the check box item, you use a Toolbox routine that will become very familiar to you—GetDItem(). GetDItem() was first introduced in Chapter 17. Pass GetDItem() the dialog item number of the item you are interested in. GetDItem() will respond by returning a handle to the item in the item_handle variable you pass to it. Once you have a handle to a dialog item, you can use the item. In this case the handle is passed to the Toolbox routine GetCtlValue(), and the returned int is saved in a variable called old_value.

A check box is always toggled. If it was checked, it gets unchecked. If it was unchecked, it gets checked. The if-else statement takes care of this toggling. It tests the current value of the check box. Then a call to SetCtlValue() is used to set the check box to the opposite value.

One final note on the Set_Check_Box() routine: whenever possible you should try to make code reusable. In other words, write functions that can be copied from one program to another and used with little or no change. As a programmer, you will be doing enough hard work that you will want to avoid duplicated effort whenever you can. The Set_Check_Box() routine is a good example of a reusable function. You can cut and paste it into any of your programs. Then just call it by passing it a pointer to the dialog box that's on the screen and the item number of the check box item that was clicked on:

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Set_Check_Box( DialogPtr the_dialog, short the_item )
{
    Handle item_handle;
    short item_type;
    Rect item_rect;
    int old_value;

    GetDItem(the_dialog, the_item, &item_type, &item_handle,
             &item_rect);
    old_value = GetCtlValue((ControlHandle)item_handle);
}
if (old_value == CONTROL_ON)
    SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);
else
    SetCtlValue((ControlHandle)item_handle, CONTROL_ON);
}

Getting Information from a Dialog Box

Once the Draw button in the dialog box has been clicked on, you have to get information from the dialog box before the dialog box can be drawn to. Get_Dialog_Info() does this. Specifically, you must get the string the user typed into the edit box, and you must determine which of the four check boxes have been checked.

To get the user-entered title string, Get_Dialog_Info() calls Get_Text From_Edit(). We cover this routine a little later.

Get_Dialog_Info() may seem like a long function, but there is a great deal of repetition in it. Let's take a close look at how the frame rectangle check box is handled—the other three check boxes are handled in the same manner.

First GetDItem() is called to get a handle to the check box titled Framed Rectangle:

GetDItem(the_dialog, FRAME_RECT_DITL_ITEM, &item_type, &item_handle, &item_rect);

GetDItem() fills item_handle with a handle to the check box. After type casting, this handle is passed to GetCtlValue(). GetCtlValue() returns the value of the check box. The returned value is stored in variable cntl_value:

cntl_value = GetCtlValue((ControlHandle)item_handle);

A check box always has a value of either 1 or 0. If it's a 1, the box is checked. If it's a 0, the box is unchecked. Two #define statements were set up at the start of the program to make it clear what the values 1 and 0 mean:

#define CONTROL_ON 1
#define CONTROL_OFF 0
An if-else statement is used to assign the global flag Framed_Rect a value of either TRUE or FALSE. If the box is checked, cnt1_value is a 1 (CONTROL_ON), and the flag is set to TRUE:

```c
if (cnt1_value == CONTROL_ON)
    Framed_Rect = TRUE;
else
    Framed_Rect = FALSE;
```

This process is repeated for each of the remaining three check boxes:

1. Get a handle to the check box.
2. Determine the check box value.
3. Assign to a flag variable a value based on the check box value.

The ShapeMaker program will use these four flag variables a little later in the program. That's why they're global variables—other functions need to be aware of their values:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Get_Dialog_Info( DialogPtr the_dialog )
{
    Handle     item_handle;
    short      item_type;
    Rect       item_rect;
    int         cnt1_value;

    Get_Text_From_Edit(the_dialog, TITLE_DITL_ITEM, Title_Str);

    GetDItem(the_dialog, FRAME_RECT_DITL_ITEM, &item_type,
             &item_handTe, &item_rect);

    cnt1_value = GetCtlValue((ControlHandle)item_handle);

    if (cnt1_value == CONTROL_ON)
        Framed_Rect = TRUE;
    else
        Framed_Rect = FALSE;
```
GetDItem(the dialog, FILL_RECT_DITL_ITEM, &item_type, &item_handle, &item_rect);

cntl_value = GetCtlValue((ControlHandle)item_handle);

if (cntl_value == CONTROL_ON)
    Filled_Rect = TRUE;
else
    Filled_Rect = FALSE;

GetDItem(the dialog, FRAME_OVAL_DITL_ITEM, &item_type, &item_handle, &item_rect);

cntl_value = GetCtlValue((ControlHandle)item_handle);

if (cntl_value == CONTROL_ON)
    Framed_Oval = TRUE;
else
    Framed_Oval = FALSE;

GetDItem(the dialog, FILL_OVAL_DITL_ITEM, &item_type, &item_handle, &item_rect);

cntl_value = GetCtlValue((ControlHandle)item_handle);

if (cntl_value == CONTROL_ON)
    Filled_Oval = TRUE;
else
    Filled_Oval = FALSE;
}

Getting Edit Item Contents

Getting the contents of an edit item is a very straightforward process. First we get a handle to the edit item using GetDItem(), then we use the handle in a call to GetIText():
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Get_Text_From_Edit(DialogPtr the_dialog, short edit_item,
-Str255 the_string )
{
    Handle item_handle;
    short item_type;
    Rect item_rect;

    GetDItem(the_dialog, edit_item, &item_type, item_handle,
-&item_rect);
    GetIText(item_handle, the_string);
}

Using Dialog Box Information to Draw in the Dialog Box

We're now at the end of the program. The last routine is the one that draws shapes to the dialog box. And in keeping with our policy of giving functions descriptive names, we've called it Draw_Shapes_In_Dialog().

The function starts by setting a rectangle the size of the lower half of the dialog box, then filling it with a white pattern. The effect? Anything that might have already been drawn in this area is erased. If the user clicks on the Draw button, the program draws shapes in the dialog box. If the user changes the check box settings and again clicks on the Draw button, the program will draw in the same area of the dialog box. So we want to erase the previous drawing first.

Next, the picture title is drawn to the dialog box. Here ShapeMaker makes use of the global variable Title_Str. Recall that the Get_Dialog_Info() function obtained the title from the edit box. Now the title is used to draw a string to the dialog box. What about the call to SetPort() that we said should be made before drawing? Remember, Draw_Shapes_In_Dialog() is called from within function Open_ShapeMaker_Dialog(). SetPort() was called in that function, so there is no need to call it again.

To determine if a rectangle should be drawn, this routine checks the value of the global flag Framed_Rect. If it has a value of TRUE, the frame rectangle check box was checked, so Draw_Shapes_In_Dialog() draws a rectangle. The coordinates of the rectangle are purely arbitrary. You can change them if you want. Or, you might want to alter the program to add the option of letting the user enter the coordinates.
The following code shows how we determine if a rectangle should be drawn:

```c
if (Framed_Rect == TRUE)
{
    SetRect (&the_rect, 20, 170, 120, 260);
    FrameRect (&the_rect);
}
```

These steps are repeated for each of the shapes. Then, one final test is performed. A check is made to see if all of the global shape-drawing flags are FALSE. If they are, the user did not have any of the check boxes checked at the time the Draw button was clicked on. The user might not know what was expected, so the routine draws a message to the dialog box:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Draw_Shapes_In_Dialog( void )
{
    Rect the_rect;

    SetRect(&the_rect, 0, 135, 450, 320);
    FillRect(&the_rect, white);

    MoveTo(20, 150);
    DrawString(Title_Str);

    if (Framed_Rect == TRUE)
    {
        SetRect(&the_rect, 20, 170, 120, 260);
        FrameRect(&the_rect);
    }

    if (Filled_Rect == TRUE)
    {
        SetRect(&the_rect, 80, 200, 230, 280);
        FillRect(&the_rect, gray);
    }

    if (Framed_Oval == TRUE)
```
Chapter 19  Complete Source Code—ShapeMaker 411

{  
  SetRect(&the_rect, 310, 170, 410, 270);
  FrameOval(&the_rect);
}

if (Filled_Oval == TRUE)  
{  
  SetRect(&the_rect, 240, 180, 320, 240);
  FillOval(&the_rect, dkGray);
}

if (Framed_Rect == FALSE &&  
    Filled_Rect == FALSE &&  
    Framed_Oval == FALSE &&  
    Filled_Oval == FALSE)  
{  
  MoveTo( 20, 190 );
  DrawString("\nNothing to draw! Click in Check Boxes, then 'Draw'.");
}

Chapter Summary

In Chapter 18 you saw that a Macintosh program is composed of resources and source code. Chapter 18 showed you how to create the resources for a program called ShapeMaker, and this chapter walked you through the source code that is used in that same application.
In Chapters 18 and 19 we described the entire process of creating a Macintosh program, using the ShapeMaker program as an example. We finish the book with one more example—the MacCertificate program.

While ShapeMaker used just a dialog box, MacCertificate uses both a dialog box and a window and has a few additional features, including error-checking and the handling of radio buttons.

The MacCertificate Program

In the THINK C Examples folder on the disk that comes with this book you’ll find a folder called MacCertificate Folder. This folder contains the source code, resource file, project file, and application for a Macintosh program called MacCertificate. Because we covered the process of creating a Macintosh program so thoroughly in Chapters 18 and 19 with the ShapeMaker program, we aren’t going to go into as much detail for the MacCertificate program. Here, we’ll take a cursory look at the resource file and
source code, slowing down to spend more time on the elements that are new to you.

Running the MacCertificate program gives you an empty screen with a menu bar containing two menus—the Apple menu and the File menu. Click on the Apple menu to see the About MacCertificate... menu option, as shown in Figure 20–1.

Selecting a desk accessory will open that desk accessory. Selecting About MacCertificate... brings up an alert just like the one we saw in the ShapeMaker program. See Figure 20–2. As a matter of fact, to create this ALRT and the DITL that is displayed in it, we simply copied parts of the ShapeMaker resource file into our MacCertificate resource file.
Don't think of copying resources from one of your programs to another as cheating. Resources exist to be borrowed. They are intended to save time and effort as you develop programs. When starting a new program, always try to reuse the resources you've created in other programs.

The second menu is the File menu. Clicking on the File menu opens a menu like the one shown in Figure 20–3.

Selecting New brings up the MacCertificate dialog box. Figure 20–4 shows that this dialog box contains one button, two edit boxes, one check box, and two radio buttons.
The MacCertificate dialog box gives the user the opportunity to list the features he or she wants in a certificate. After selecting the features, the user clicks the OK button. The dialog box disappears from the screen, and a window containing the certificate opens, as shown in Figure 20–5.

The MacCertificate program includes all of the features we found in ShapeMaker, and then some; the MacCertificate program demonstrates the use of radio buttons in a dialog box and a window that is draggable.

**The MacCertificate Resource File**

Because we covered resources so thoroughly in Chapter 18, we'll cover the MacCertificate resources quickly here. The complete resource file is shown in Figure 20–6.

**The DLOG and Its DITL**

MacCertificate contains one DLOG—the dialog box that allows the user to set certificate display options. It is shown in Figure 20–7.
Chapter 20  One for the Road—MacCertificate

FIGURE 20–6  The MacCertificate resource file

FIGURE 20–7  The certificate DLOG window
The MacCertificate resource file contains several DITL resources. One of them, ID 300, is for our DLOG. The rest are for the program's ALRTs. Figure 20–8 shows the list of DITLs.

The DITL resource for the certificate DLOG is shown in Figure 20–9. Note that we gave both the DLOG and the DITL the same ID, 300.
The MacCertificate program contains several ALRT resources. Figure 20–10 lists them.

One of the ALRTs, ID 500, is used to display the About MacCertificate... item under the menu, as shown in Figure 20–11.
The DITL corresponding to this ALRT, ID 500, is shown in Figure 20–12. We have included the About ShapeMaker... ALRT DITL from the ShapeMaker program for comparison. Notice any differences between the two? Just one item has been changed—the program title.

To create the MacCertificate About... DITL we simply copied the About... DITL from the chapter’s ShapeMaker program. We then double-clicked on number 2, the static text item that holds the program’s title, and typed in the new title.
IMPORTANT

Resources—copy, copy, copy. If you are creating a program similar to one you have already written, by all means copy the entire resource file. Rename it and then edit it. Whatever you do, don’t “reinvent the wheel.”

The remaining six ALRTs have a common purpose—to display an error message if something goes wrong in the MacCertificate program. For simplicity, such alerts haven’t been included in the previous programs. Now that you’re an old pro at creating ALRT and DITL resources, you can easily add a few more of each.

As you’ve probably noticed at one time or another, programs sometimes crash. It’s important to anticipate fatal errors that might occur in your program and try to give the user information about them, even if it’s impossible to recover from them.

NOTE

An error is said to be fatal when it causes the program to hang or freeze—the worst type of program error. Once the Macintosh is frozen, there is nothing the user, or the program, can do to recover.

One of the primary causes of severe errors is a missing, or misnumbered, resource. Imagine that a program attempts to open a dialog with the following call:

```c
the_dialog = GetNewDialog(300, NIL_PTR, IN_FRONT);
```

The program expects to find a DLOG resource with an ID of 300 in the resource part of the program. If there is no DLOG resource with this ID, the call to GetNewDialog() fails and the program crashes. If you can write your source code in such a way that when this situation arises an alert comes to the screen with an error message, you’ll be able to pinpoint the exact cause of the program failure. Here, we’ll describe the ALRT resources you’ll need to accomplish this. Later in this chapter we’ll tell you what C language code to include in your source code.
Many moons ago we issued a warning about ResEdit. We said that although it is an extremely useful and powerful programming tool, it is subject to abuse.

Remember, all resources start out in a resource file. When a program is compiled, the resources in this file are merged with the source code into one program. That means that all programs have resources in them. ResEdit can be used to look at the resources in any application, not just those found in separate resource files.

If a user of a program you wrote tries to look at the program’s resources, he or she could inadvertently delete a needed resource, such as a DLOG. When this user later tries to run your program, it will crash. You can’t stop a person from fiddling with your program’s resources, but you can put up a message that lets him or her know that a resource is at fault. The user could then either use a backup copy of the program or report to you the content of the error message.

We’ve created a separate ALRT resource for some of the anticipated errors that could occur in the MacCertificate program. In Figure 20–13 we show the ALRT, and in Figure 20–14 we show the corresponding DITL for...
Chapter 20  One for the Road—MacCertificate  423

FIGURE 20–14  The DITL for ALRT 501

one error situation. If MacCertificate fails to find DLOG ID 300, ALRT ID 501 will appear on the user’s screen.

You may wonder why, if DLOG ID 300 is missing or damaged, we should think that the alert meant to come to the screen in that situation, ALRT 501, might not also be gone. If that’s the case, then certainly when the program attempts to post an alert, that effort will fail as well. You’re absolutely correct. But there is only so much you can do.

MacCertificate does not claim to be able to handle every type of error that can occur—error handling is an art in itself. But it does serve as a good introduction to the topic.

Figure 20–15 shows that each ALRT has a corresponding DITL. We’ve elected to show you the ALRT and DITL for just one of the error alerts, ID 501. The others are very similar.

The MENU Resource

The MacCertificate program has two MENU resources. The first is the Apple menu, with an About MacCertificate… item and a dashed line to separate that item from the desk accessories that will be added in the source code. The second menu is a File menu with two items—New and Quit. The two MENU resources are shown in Figure 20–16.

The MBAR Resource

The two MENU resources will appear in the main menu bar of MacCertificate. Bind them together into that one menu with the MBAR resource shown in Figure 20–17.
FIGURE 20-15 Every ALRT has a corresponding DITL.

FIGURE 20-16 The MENU window
The WIND Resource

MacCertificate opens a window that displays the certificate. The resource file's single WIND resource is shown in Figure 20–18.
The PICT Resource

MacCertificate is the first program that uses a PICT resource—a picture. The MacCertificate source code loads the PICT resource into memory, then displays the picture in the certificate window. Figure 20–19 shows the picture—a simple drawing of a ribbon.

To create a PICT resource, you must first find an existing clip art picture, or you can draw one of your own, as we did. Use any Macintosh paint or draw program for your picture. Then copy it to the Macintosh Scrapbook. Next, run ResEdit. Open the Scrapbook, copy the picture, then paste it into your resource file. ResEdit is smart enough to save it as a PICT resource type. You can change the ID of the PICT, as we did, by selecting the Get Resource Info option from ResEdit’s Resource menu.

When we discuss the MacCertificate source code, we’ll show you how to load and display the PICT resource.

That completes our walk through the MacCertificate resource file. Now, you’re ready to examine the source code.
The MacCertificate Code

The MacCertificate code is similar in many ways to the ShapeMaker source code presented in Chapter 19. We list all of the code here, but provide detailed comments for only those sections that are new.

The Function Prototypes

As we did for ShapeMaker, the first thing we do is list a prototype for each function:

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Initialize_Toolbox( void );  
void Set_Up_Menu_Bar( void ); 
void Post_Error_Alert( int );  
void Set_Window_Drag_Boundaries( void ); 
void Main_Event_Loop( void );  
void Handle_One_Event( void );  
void Handle_Update_Event( void );  
void Update_Window( WindowPtr );  
void Update_Certificate_Window( void );  
void Draw_Certificate_Border( void );  
void Draw_Row_Of_Shapes( int, int );  
void Draw_Ribbon_Picture( void );  
void Close_Window( WindowPtr );  
void Handle_Mouse_Down( void );  
void Handle_Menu_Choice( long );  
void Handle_Apple_Choice( short );  
void Handle_File_Choice( short );  
void Open_Certificate_Dialog( void );  
void Set_Radio_Buttons( DialogPtr, short );  
void Get_Dialog_Info( DialogPtr the_dialog );  
void Get_Text_From_Edit( DialogPtr, short, Str255 );  
int Check_For_Valid_Score( void );  
void Open_Certificate_Window( void );  
void Set_Check_Box( DialogPtr, short );
The define Statements

The code begins with several #define statements, some of which you've seen in the past. About half are resource IDs:

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

#define OK_BUTTON_DITL_ITEM 1
#define NAME_DITL_ITEM 2
#define SUBJECT_DITL_ITEM 3
#define CHECK_DITL_ITEM 4
#define NUM_DITL_ITEM 5
#define OVAL_DITL_ITEM 6
#define RECTANGLE_DITL_ITEM 7

#define CERTIFICATE_DLOG_ID 300
#define CERTIFICATE_WIND_ID 400
#define ABOUT_ALRT_ID 500
#define DLOG_ERROR_ALRT_ID 501
#define MBAR_ERROR_ALRT_ID 502
#define MENU_ERROR_ALRT_ID 503
#define PICT_ERROR_ALRT_ID 504
#define WIND_ERROR_ALRT_ID 505
#define BAD_SCORE_ALRT_ID 506

#define RIBBON_PICT_ID 600
#define MAIN_MBAR_ID 200
#define APPLE_MENU_ID 201
#define FILE_MENU_ID 202

#define ABOUT_ITEM 1
#define NEW_ITEM 1
#define QUIT_ITEM 2

#define CONTROL_ON 1
#define CONTROL_OFF 0

#define OVAL_FRAME 1
#define RECT_FRAME 2
The Global Variables

Next our global variables are listed:

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

long Score_Int;
Str255 Score_Str;
Boolean Print_Score;
Str255 Name_Str;
Str255 Subject_Str;
int Frame_Shape;
short Old_Button_Num;
Rect Drag_Rect;
WindowPtr Certificate_Window_Ptr;
Boolean All_Done;
MenuHandle Apple_Menu;
MenuHandle File_Menu;
EventRecord The_Event;
Boolean Multifinder_Active;

The variable Score_Str is a string that holds the score that the user can type into the certificate dialog box. Score_Int holds the same number,
but as an actual number—a long. Print_Score tells the program whether or not to print the score the user types in. The user can enter a name to be written on the certificate, Name_Str, and a subject for which the certificate is being awarded, Subject_Str. The certificate will have a simple border; Frame_Shape tells which of the two border types the user selected.

Old_Button_Num helps MacCertificate keep track of which of the two radio buttons is currently selected. The variable Drag_Rect defines the screen coordinates of the area in which the certificate window can be dragged. The variable Certificate_Window_Ptr is a pointer to the certificate window.

The remaining five variables—All_Done, Apple_Menu, File_Menu, The_Event, and Multifinder_Active—were all encountered in the Shape-Maker program. Once again, to save time and effort, borrow from existing code wherever possible.

**NOTE**

Is the variable name Certificate_Window_Ptr too long? Might its length make it a bit awkward? Perhaps. But you must admit it is very descriptive. You’re the programmer; you make the choice.

**The main() Routine**

The main() function is very similar to the one we saw in ShapeMaker. Here, one new function call has been added. The call to Set_Window_Drag_Boundaries() establishes the boundaries within which the user can drag a window. More on that when we look at the definition of the function:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
main()
{
    Initialize_Toolbox();
    Set_Up_Menu_Bar();
    Set_Window_Drag_Boundaries();
    Main_Event_Loop();
}
```
Program Initialization

Initialize_Toolbox() appears exactly as it has several times before:

/**++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Initialize_Toolbox( void )
{
    InitGraf(&thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEinit();
    InitDialogs(NIL_PTR);
    FlushEvents(everyEvent, REMOVE_EVENTS);
    InitCursor();
}

Setting up the Menu

You saw the routine Set_Up_Menu_Bar() in ShapeMaker. Here it has been modified to include a check to verify that the MENU resources are all present in the resource file. ShapeMaker’s call to GetNewMBar() looked like this:

main_menu_bar = GetNewMBar(MAIN_MBAR_ID);

Now, the same call has been embedded in an if statement that compares the returned pointer to the NIL_PTR:

if ((main_menu_bar = GetNewMBar(MAIN_MBAR_ID)) == NIL_PTR)
    Post_Error_Alert(MBAR_ERROR_ALRT_ID);

If the MBAR resource is not present, the call to GetNewMBar() will return a nil pointer. That’s a signal to you that there is a problem. An alert, which will be described in a moment, is then put up by the Post_Error_Alert() routine.

The same type of error checking is done for the two MENU resources that are loaded in this routine. If there is a problem, Post_Error_Alert() is again called, with the appropriate #define constant passed as a parameter:
Think THINK C

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Set_Up_Menu_Bar( void )
{
    Handle main_menu_bar;

    if ((main_menu_bar = GetNewMBar(MAIN_MBAR_ID)) == NIL_PTR)
        Post_Error_Alert(MBAR_ERROR_ALRT_ID);

    SetMenuBar(main_menu_bar);

    if ((Apple_Menu = GetMHandle(APPLE_MENU_ID)) == NIL_PTR)
        Post_Error_Alert( MENU_ERROR_ALRT_ID );

    if ((File_Menu = GetMHandle(FILE_MENU_ID)) == NIL_PTR)
        Post_Error_Alert( MENU_ERROR_ALRT_ID );

    AddResMenu( Apple_Menu, 'DRVR' );

    DrawMenuBar();
}

Handling an Error Condition

If an attempt to load a resource fails, Post_Error_Alert() is called. If all goes well in the program, this routine should never be called.

Post_Error_Alert() is passed the ALRT ID of the alert that is to be posted on the screen. In the resource file are six error alerts, five of which deal with resource loading errors. This routine posts the alert corresponding to the error that has occurred. When the user clicks the alert's button, the Toolbox routine ExitToShell() is called. This routine exits the current program and returns the user to the Finder—the DeskTop. Figure 20–20 shows one of the error alerts posted in MacCertificate.

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Post_Error_Alert( int error_id )
{
    StopAlert(error_id, NIL_PTR);

    ExitToShell();
}
Setting up the Window Drag Boundaries

`Set_Window_Drag_Boundaries()` is a routine called from `main()`. When a window is placed on the screen, the user is allowed to move it. You'll want to set some limits as to how close to the edge of the screen it can be dragged. This prevents the window from appearing to slide off the screen.

Global variable `Drag_Rect` is set to hold the area of the screen in which the window can be moved. To do this, you can take advantage of one of the Macintosh's own global variables, `screenBits`. The `screenBits` variable is a structure that need not be declared in your program—it resides in RAM memory and is available for use by any program. Here the `screenBits.bounds` field of the `screenBits` structure is used. This is a Rect that defines the area of the Macintosh screen. `Set_Window_Drag_Boundaries()` offsets this rectangle a few pixels using the `#define DRAG_EDGE`. When the user drags a window, that window will not be allowed to go beyond any of the dimensions of `Drag_Rect`. This is demonstrated later in the routine `Handle_Mouse_Down()`.

```c
/*+++++++++++++++++++++++++++++++++++++++*/
void Set_Window_Drag_Boundaries( void )
{
    Drag_Rect = screenBits.bounds;
    Drag_Rect.left += DRAG_EDGE;
    Drag_Rect.right -= DRAG_EDGE;
    Drag_Rect.bottom -= DRAG_EDGE;
}
```
The Main Event Loop

The last routine called by main() is the main event loop. This is the same as the one we saw in ShapeMaker with a couple of minor additions. Here, three variables—Old_Button_Num, Frame_Shape, and All.Done—are initialized. These are variables used in the certificate dialog box:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Main_Event_Loop( void )
{
    Old_Button_Num = OVAL_DITL_ITEM;
    Frame_Shape = OVAL_FRAME;
    All.Done = FALSE;

    Multifinder_Active = (NGetTrapAddress(WNE_TRAP_NUM, ToolTrap)
        != NGetTrapAddress(UNIMPL_TRAP_NUM, ToolTrap));

    while (All.Done == FALSE)
        Handle_One_Event();
}

Handling a Single Event

The next routine, Handle_One_Event(), is much like the similarly named routine in ShapeMaker. The additions made here allow the program to update the certificate window—something that was not a concern in the windowless ShapeMaker program.

In the updateEvt case of the switch statement, a call to Handle_Update_Event() has been added. This routine carries out the actual updating of a window. We'll discuss it next.

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Handle_One_Event( void )
{
    if (Multifinder_Active == TRUE)
        WaitNextEvent(everyEvent, &The_Event, SLEEP_TICKS,
            &MOUSE_REGION);
    else
    {
        SystemTask();
    }
```
Handling an Update Event

Long ago you learned that every window has its own port. If there is more than one window on the screen and one of them needs to be updated, follow these steps:

1. Save the current port.
2. Set the port to the window that needs updating.
3. Perform the update.
4. Set the port back to the original port.

Although MacCertificate has only one window, we want to lend a little insight into the way a Macintosh program handles updating when more than one window is involved. Here is how MacCertificate's code implements the four steps listed above:

1. To save the current port, Handle_Update_Event() uses Toolbox routine GetPort() and the local GrafPtr variable old_port:

   GetPort(&old_port);

2. To set the port to the port of the window that needs updating, the Toolbox routine SetPort() is called. If the event was an update event, the message field of The_Event tells which window was involved. The
local variable the_window is set to the message field of The_Event, after it is typecast to a WindowPtr. Then the port is set:

```c
the_window = (WindowPtr)The_Event.message;
SetPort(the_window);
```

3. To update the window, Handle_Update_Event() calls the Update_Window() routine, passing a pointer to the window that needs to be updated. The Update_Window() routine does the actual updating—we'll cover it after we finish with Handle_One_Event(). The call to Update_Window() is surrounded by the Toolbox routines BeginUpdate() and EndUpdate(). These two routines tell the Mac to update only the parts of a window that need updating. If a window is moved partially offscreen, then back on, only that part that went off screen needs to be redrawn. The Mac knows what part of a window needs updating, so let it decide how much to redraw:

```c
BeginUpdate(the_window);
    Update_Window(the_window);
EndUpdate(the_window);
```

4. With the updating complete, the port is set back to the original port by again calling SetPort()—this time passing the old port, which was saved in the variable old_port:

```c
SetPort(old_port);
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Handle_Update_Event( void )
{
    GrafPtr old_port;
    WindowPtr the_window;

    the_window = (WindowPtr)The_Event.message;
    GetPort(&old_port);
    SetPort(the_window);
    BeginUpdate(the_window);
        Update_Window(the_window);
    EndUpdate(the_window);

    SetPort(old_port);
}
*/
WindowPtr Certificate_Window_Ptr;
WindowPtr Honor_Window_Ptr;

Update_Window( WindowPtr the_window )
{
    if ( the_window == Certificate_Window_Ptr )
        Update_Certificate_Window();
    else if ( the_window == Honor_Window_Ptr )
        Update_Honor_Window();
}

Update_Certificate_Window()
{
    /* take steps to update a certificate window */
}

Update_Honor_Window()
{
    /* take steps to update an honor window */
}

FIGURE 20-21 One way to handle the updating of multiple windows

Updating a Window

The Update_Window() routine is just a couple of lines of code, but preparation has been made for future additions to MacCertificate. A check is made to see which window needs updating by comparing the WindowPtr variable that was passed to the routine to the global WindowPtr variable Certificate_Window_Ptr. If MacCertificate dealt with more than one type of window, the comparison of the passed WindowPtr would also be made with all other WindowPtr variables. For a routine that handles more than one window type, see Figure 20-21.

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Update_Window( WindowPtr the_window )
{
    if (the_window == Certificate_Window_Ptr)
        Update_Certificate_Window();
}
The actual updating of the certificate window takes place in Update_Certificate_Window(). Though the routine may look long, it's composed primarily of code that draws text to the certificate window.

Of note is the way the two user-entered strings are drawn. The goal is to center them in the window, even though their pixel lengths are not known in advance. This means that a MoveTo() with hard-coded numbers can't be used. Hard-coded values are values that are constant, like MoveTo(65,60). Instead, the method discussed in Chapter 12 is used. First, the length of a string is found using Toolbox routine StringWidth(). Then, the starting pixel location is based on this value and the width of the window.

One line of text may or may not be drawn, depending on whether the user checked the check box in the certificate dialog box. The global variable Print_Score is checked to see if the score string should be written to the window.

Finally, two more routines are called—one to draw a border in the window and another to draw the ribbon PICT in the window. Figure 20–22 shows how the certificate window looks after a typical call to Update_Certificate_Window().

![Certificate Window](image)

**ACHIEVEMENT CERTIFICATE**

*In recognition of work accomplished in the study of Zoology*

*This certificate is presented to Joe Smith for passing with a test score of 80%*

**FIGURE 20–22 Result of calling Update_Certificate_Window**
void Update_Certificate_Window( void )
{
    int string_pixel_size;
    int horiz_location;

    TextFont(geneva);
    TextSize(24);
    TextFace(bold);

    MoveTo(65, 60);
    DrawString("\pACHIEVEMENT CERTIFICATE");

    TextSize(12);
    MoveTo(60, 100);
    DrawString("\pIn recognition of work accomplished in the study of");

    string_pixel_size = StringWidth(Subject_Str);
    horiz_location = (WINDOW_WIDTH - string_pixel_size)/2;
    MoveTo(horiz_location, 130);
    DrawString(Subject_Str);

    MoveTo(125, 165);
    DrawString("\pThis certificate is presented to");

    TextSize(18);
    string_pixel_size = StringWidth(Name_Str);
    horiz_location = (WINDOW_WIDTH - string_pixel_size)/2;
    MoveTo(horiz_location, 195);
    DrawString(Name_Str);

    if (Print_Score == TRUE)
    {
        TextSize(12);
        MoveTo(110, 230);
        DrawString("\pFor passing with a test score of ");
Figure 20–22 shows the simple border that is drawn on the certificate—a row of small shapes running across the top and bottom. To achieve this, the graphics pen is first moved to the starting location of a row. Then a call to Draw_Row_Of_Shapes() is made to do the actual drawing:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Draw_Certificate_Border( void )
{
    int left;
    int top;

    left = 0;
    top = 10;
    Draw_Row_Of_Shapes(left, top);

    left = 0;
    top = 270;
    Draw_Row_Of_Shapes(left, top);
}
```

Draw_Row_Of_Shapes() receives two passed parameters, which are used as the starting coordinates for one row of shapes. A global #define is used to determine the coordinates of the right and bottom sides of the first shape. A for loop is then used to draw the shape repeatedly. The value of the global variable Frame_Shape determines whether a row of rectangles or a row of ovals is drawn. Depending on the value of FrameShape, a call to either FrameOval() or Frame()Rect is made.
void Draw_Row_Of_Shapes( int left, int top )
{
    int index;
    Rect the_rect;
    int right, bottom;

    right = left + SHAPE_SIZE;
    bottom = top + SHAPE_SIZE;

    for (index = 0; index < NUM_SHAPES; index++)
    {
        left += SHAPE_SIZE + SHAPE_SPACING;
        right = left + SHAPE_SIZE;
        SetRect(&the_rect, left, top, right, bottom);

        if (Frame.Shape == OVAL_FRAME)
            FrameOval(&the_rect);
        else
            FrameRect(&the_rect);
    }
}

The final window updating routine is Draw_Ribbon_Picture(). Here, the PICT resource is loaded into memory and displayed to the certificate window.

The first step is to load the PICT resource and get a handle to it. The handle you'll need is a PicHandle, which is a predefined Macintosh type. This is accomplished with a call to the Toolbox routine GetPicture(). A simple call to GetPicture() looks like this:

PicHandle certificate_picture;

certificate_picture = GetPicture(RIBBON_PICT_ID);

We elected to take advantage of our error handling routine by embedding GetPicture() in an if statement:

if (certificate_picture = GetPicture(RIBBON_PICT_ID)) == NIL_PTR)
    Post_Error_Alert(PICT_ERROR_ALRT_ID);
Once the PICT is in memory, it can be placed anywhere on the certificate, but you must first set a rectangle for it to be drawn into. Before doing that, determine the size of the PICT, in pixels. You can obtain the PICT size from the PicHandle. Remember, a handle points to a pointer, which in turn points to an actual data structure—in this case the data structure is the Macintosh C type Picture. One of the fields of a Picture is picFrame—the rectangle that encloses the picture. Here’s how we set our local Rect variable picture_rect to the rectangle that encloses the PICT:

```c
picture_rect = (**(certificate_picture)).picFrame;
```

If this all seems a bit confusing, you can either take our word for it or refer back to Chapter 9, where we discuss handles.

Once the enclosing rectangle information for the picture is obtained, we can use SetRect() to set a rectangle of this size anywhere we want in the certificate window. We chose to place the ribbon in the lower-left corner. The actual drawing of the picture is done with the Toolbox routine DrawPicture():

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void Draw_Ribbon_Picture( void )
{
    Rect    picture_rect;
    int     left, top, right, bottom;
    PicHandle certificate_picture;
    int     picture_width;
    int     picture_height;

    if ((certificate_picture = GetPicture(RIBBON_PICT_ID)) == NIL_PTR)
        Post_Error_Alert(PICT_ERROR_ALRT_ID);

    picture_rect = (**(certificate_picture)).picFrame;

    picture_width = picture_rect.right - picture_rect.left;
    picture_height = picture_rect.bottom - picture_rect.top;

    left= 15;
    right = left + picture_width;
    top = 130;
```
bottom = top + picture_height;
SetRect(&picture_rect, left, top, right, bottom);

DrawPicture(certificate_picture, &picture_rect);
}

Closing a Window

When it comes time to close the certificate window, Close_Window() is called. Like the updating routine, Close_Window() has been written in such a way that it can be made to handle the closing of other types of windows with little modification. Closing a window involves simply removing it from memory with DisposeWindow() and setting the pointer to that window to the nil pointer:

/**++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Close_Window( WindowPtr the_window )
{
  DisposeWindow(the_window);

  if (the_window == Certificate_Window_Ptr)
    Certificate_Window_Ptr = NIL_PTR;
}

Handling a mouseDown Event

When a mouse click occurs, Handle_Mouse_Down() is called. This version of Handle_Mouse_Down() is very similar to the one in the ShapeMaker program. Here, we'll just discuss the additions.

Because MacCertificate uses a window, the switch statement is given the ability to deal with mouseDown events that occur in a window. If the mouse is clicked in a window's drag region—that is, the title bar—Handle_Mouse_Down() responds by calling the Toolbox routine DragWindow(). As long as the user is holding the mouse button down, DragWindow() will handle the moving of the window.

If the mouseDown occurs in the window's close box, TrackGoAway() is called. This Toolbox routine will return a value of TRUE if the mouse button is released in the close box. If this is the case, we know the user fol-
lowed through on his or her intention to close the window, and Close_Window() is called to do just that.

If the user clicks the mouse anywhere else in the window, SelectWindow() is called. This Toolbox routine makes the window active if it's not already so:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Handle_Mouse_Down( void )
{
    WindowPtr which_window;
    short the_part;
    long menu_choice;

    the_part = FindWindow(The_Event.where, &which_window);

    switch (the_part)
    {
        case inMenuBar:
            menu_choice = MenuSelect(The_Event.where);
            Handle_Menu_Choice(menu_choice);
            break;
        case inSysWindow:
            SystemClick(&The_Event, which_window);
            break;
        case inDrag:
            DragWindow(which_window, The_Event.where, &Drag_Rect);
            break;
        case inGoAway:
            if ( TrackGoAway(which_window, The_Event.where))
                Close_Window(which_window);
            break;
        case inContent:
            SelectWindow(which_window);
            break;
    }
}
```
Handling a mouseDown in a Menu

Handle_Menu_Choice() handles a mouse click in a menu. This routine was lifted word-for-word from the ShapeMaker program, with no changes needed. As we stressed with resources, don't let your previous efforts go to waste. If a piece of code you've written in the past helps you in a new program, use it!

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Handle_Menu_Choice( long menu_choice )
{
    int the_menu;
    int the_menu_item;

    if (menu_choice != 0)
    {
        the_menu = HiWord(menu_choice);
        the_menu_item = LoWord(menu_choice);

        switch (the_menu)
        {
            case APPLE_MENU_ID:
                Handle_Apple_Choice(the_menu_item);
                break;
            case FILE_MENU_ID:
                Handle_File_Choice(the_menu_item);
                break;
        }
        HiliteMenu(0);
    }
}
```

Handling a mouseDown in the Apple Menu

The Handle_Apple_Choice() function takes care of selections made in the Apple menu. This routine is identical to the Handle_Apple_Choice() that was developed for ShapeMaker:
void Handle_Apple_Choice( short the_item )
{
    Str255    desk_acc_name;
    int       desk_acc_number;

    switch (the_item)
    {
        case ABOUT_ITEM:
            NoteAlert(ABOUT_ALRT_ID, NIL_PTR);
            break;
        default:
           .GetItem(Apple_Menu, the_item, desk_acc_name);
            desk_acc_number = OpenDeskAcc(desk_acc_name);
            break;
    }
}

Handling a mouseDown in the File Menu

Handle_File_Choice() is called when the user makes a menu selection from the File menu. Although this routine is different from the Handle_File_Choice() of ShapeMaker, it has a similar style.

If the user selects MacCertificate's New item from the File menu, the program first checks to see if a certificate window is already open. If one is, it is closed. Open_Certificate_Dialog() is then called to open the dialog box that lets the user enter information about the certificate. When the user is done with the dialog box—when he or she has dismissed it—control returns to Handle_File_Choice(). Open_Certificate_Window() is then called to open the window that displays the certificate:

void Handle_File_Choice( short the_item )
{
    switch (the_item)
    {
        case NEW_ITEM:
            if (Certificate_Window_Ptr != NIL_PTR)
Close_Window( Certificate_Window_Ptr);
Open_Certificate_Dialog();
Open_Certificate_Window();
break;
case QUIT_ITEM:
    All_Done = TRUE;
    break;
}
}

When the user selects New from the File menu, we call Open_Certificate_Dialog(). This routine differs quite a bit from the Open_ShapeMaker_Dialog() routine of ShapeMaker, but both use a similar format for opening a dialog box.

Because the dialog box contains radio buttons, we have an obligation to turn one of them on upon opening the dialog box. The #define CONTROL_ON is used when a control such as a radio button is turned on. A grouping of radio buttons always has one button turned on. We arbitrarily decided to turn the Oval button on.

When we chose a button to turn on, we also had to consider the effect of that button being on. In the MacCertificate program, if the oval button is on, the global variable Frame_Shape is set to the #define OVAL_FRAME. We took care of this in our Main_Event_Loop() routine. Figures 20–23, 20–24, and 20–25 show the source code needed for initializing a button.

![Select a Certificate Border Style:](image)

Select a Certificate Border Style:
- Oval
- Rectangle

DIITL Item #6

#define OVAL_DITL_ITEM 6

#define OVAL_FRAME 1  
#define RECT_FRAME 2  
A way to distinguish between frame types

**FIGURE 20–23** #defines used in initializing one of the buttons to on
After we turn one button on, a while loop is entered. It will take two conditions to end the loop—the OK button must be clicked, and the user-entered score must be valid. We've defined valid as a number between 0 and 100. We'll examine the function that checks the score, Check_For_Valid_Score(), a little later. For now, be aware that it returns a Boolean value, which is assigned to the local variable valid_score. If the score isn't in our range, we put up an alert with a descriptive message to let the user know something is wrong. If we feel the user did something wrong, it is up to us as programmers to provide some feedback so that he or she can try again. Figure 20–26 shows this alert.
Enter a percentage score:
It must be an integer greater than or equal to 0, and less than or equal to 100.

Enter Person's Percent Score: 200

Select a Certificate Border Style:
- Oval
- Rectangle

FIGURE 20-26 Checking the user's input

If either of the radio buttons is clicked on, the action is handled with a call to Set_Radio.Buttons(), which turns the button on. We describe Set_Radio.Buttons() next. Additionally, the global variable Frame.Shape is set to the appropriate frame shape for later use:

*+-----------------------------------------------*

void Open_Certificate_Dialog( void )
{
    DialogPtr the_dialog;
    Boolean valid_score = FALSE;
    short the_item;
    Handle item_handle;
    short item_type;
    Rect item_rect;

    if ((the_dialog = GetNewDialog(CERTIFICATE_DIALOG_ID, NIL_PTR,
                                     0, IN_FRONT)) == NIL_PTR)
        Post_Error_Alert(DLOG_ERROR_ALRT_ID);

    GetDItem(the_dialog, Old_Button_Num, &item_type,
              Old_Handle, &item_rect);
SetCtlValue((ControlHandle)item_handle, CONTROL_ON);

while(valid_score == FALSE)
{
    ModalDialog(NIL_PTR, &the_item);
    switch (the_item)
    {
        case OK BUTTON DITL_ITEM:
            Get_Dialog_Info(the_dialog);
            if (Print_Score == TRUE)
                valid_score = Check_For_Valid_Score();
            else
                valid_score = TRUE;
            if (valid_score == FALSE)
                StopAlert(BAD_SCORE_ALRT_ID, NIL_PTR);
                break;
        case CHECK DITL_ITEM:
            Set_Check_Box(the_dialog, the_item);
            break;
        case OVAL DITL_ITEM:
            Set_Radio_Buttons(the_dialog, the_item);
            Frame_Shape = OVAL_FRAME;
            break;
        case RECTANGLE DITL_ITEM:
            Set_Radio_Buttons(the_dialog, the_item);
            Frame_Shape = RECT_FRAME;
            break;
    }
}

DisposDialog(the_dialog);

**Setting a Radio Button**

The Set_Radio.Buttons() routine has a very simple purpose—to turn one radio button off and turn another one on. This routine is passed a pointer to the dialog box the user is currently working in, which makes the routine usable in other programs. We also pass in the DITL item number of the radio button that was clicked on—the new button.
A call to GetDItem() is used for the sole purpose of getting a handle to the button that was previously on—the button that was on before the new button was clicked on. The Toolbox routine SetCtlValue() is then used to set that button value to 0, CONTROL_OFF.

GetDItem() is then called a second time, this time to get a handle to the new button. And, again, SetCtlValue() is called, this time setting the button value to 1, CONTROL_ON. Finally, the new button is saved as the old button in anticipation of the next time the user clicks on a radio button.

What happens if the user clicks on a radio button that is already on? In one call to this routine, the button will be turned off and immediately turned back on:

```c
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Set_Radio.Buttons(DialogPtr the_dialog, short
new_button_num)
{
    Handle item_handle;
    short item_type;
    Rect item_rect;

    GetDItem(the_dialog, Old Button Num, &item_type,
new_button_num, &item_handle, &item_rect);
    SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);

    GetDItem(the_dialog, new_button_num, &item_type,
new_button_num, &item_handle, &item_rect);
    SetCtlValue((ControlHandle)item_handle, CONTROL_ON);

    Old_Button_Num = new_button_num;
}

Getting Information from a Dialog Box

MacCertificate’s Get_Dialog_Info() routine is very similar to ShapeMaker’s Get_Dialog_Info() routine. So similar, in fact, that we feel we can sum it up in just a few sentences.

Get_Dialog_Info() repeatedly calls our Get_Text_From_Edit() function to get the string contents of each of the three edit items. Next, a call to
GetCtlValue() is used to get the value of the one check box in the dialog box.

Notice that no effort is made to get the values of the radio buttons. The Open_Certificate_Dialog() routine set the value of Frame_Shape. This global variable provides all the information needed of the radio buttons.

/*++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Get_Dialog_Info( DialogPtr the_dialog )
{
    Handle      item_handle;
    short       item_type;
    Rect        item_rect;
    int         cntl_value;

    Get_Text_From_Edit(the_dialog, NAME_DITL_ITEM, Name_Str);
    Get_Text_From_Edit(the_dialog, SUBJECT_DITL_ITEM, Name_Str);
    Get_Text_From_Edit(the_dialog, NUM_DITL_ITEM, Name_Str);

    GetDItem(the_dialog, CHECK_DITL_ITEM, &item_type, &item_handle, &item_rect);
    cntl_value = GetCtlValue((ControlHandle)item_handle);

    if (cntl_value == CONTROL_ON)
        Print_Score = TRUE;
    else
        Print_Score = FALSE;
}

**Getting Edit Item Contents**

To get the contents of an edit item, we use the function developed for the ShapeMaker program—Get_Text_From_Edit():

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Get_Text_From_Edit(DialogPtr the_dialog, short edit_item, Str255 the_string )
{

Chapter 20 One for the Road—MacCertificate

Handle item_handle;
short item_type;
Rect item_rect;

GetDItem(the dialog, edit_item, &item_type, &item_handle,
   &item_rect);
GetIText(item_handle, the_string);
}

Checking User Input

Looking back a little, recall that the while loop in the Open_Certificate-Dialog() routine will not terminate until two things happen: the user clicks the OK button, and the user-entered score is valid—that is, within our range of 0 to 100. The Check_For_Valid_Score() routine determines if the second of these two events has occurred.

This routine is the only one in this program that has a return value. It also illustrates a couple of tricks you’ll find useful when dealing with strings.

Whenever text is retrieved from an edit box, it is just that—text. Even if the user types in a number, it is read in as text using the Toolbox function GetIText(). It is saved as a Str255 variable, as shown in Figure 20–27.

This may at first seem like a big disadvantage to us as programmers. We want a number, but we have a string. With just a little extra work, we can have both. We just have to understand how a string is stored in memory. With the exception of the very first byte of the string, a Str255 is an array of characters. When a Str255 variable is placed in memory, it is put in memory in such a way that it takes up as little memory as possible. Recall that one character takes up one byte of memory. A Str255 variable can contain up to 255 characters, or bytes, but a typical string usually has far fewer. The Score_Str shown in Figure 20–27, entered by the user as 85, is four bytes long. Here’s why:

- The first byte contains the number of characters in the string. In our example there are two characters—an 8 and a 5.
- The second and third bytes contain the two characters.
- For reasons important to only the Macintosh, a Str255 must always hold an even number of bytes. Your C compiler is aware of this, and it adds an extra byte of padding to the end of the string to comply with the Macintosh’s wishes.
Enter Person's Percent Score: 85

The user types in a number but we always retrieve edit box contents as strings:

```
Str255 Score_Str;
GetDlgItem ( the_dialog, NUM_DTL_ITEM, &item_type,
&item_handle, &item_rect );
GetDlgItemText ( item_handle, &Score_Str );
```

**We can write the string:**

```
DrawString ( Score_Str );  OK!
```

**But we can't use it in math:**

```
int_100 = Score_Str + 15;  NOT ALLOWED!
```

**FIGURE 20–27** Text is always retrieved as a Str255.

As a programmer, you don't have to concern yourself with whether a string holds an even or odd number of bytes—let the compiler worry about that. This padded byte is of no use or importance to you. Figure 20–28 illustrates a Str255 in memory.

Enough theory—let's now put our understanding of how a Str255 is stored in memory to good use. You know that the first byte of a Str255 array, element 0, contains not a character but the length of the string itself. From this you can get the length of, that is, the number of characters in, a string:

```
Str255 Score_Str;
int  str_length;

str_length = Score_Str[0];
```

From your knowledge of Str255 storage, you can also determine what any individual character in the string is:
If a string holds an odd number of bytes, the compiler will always add 1 byte to make an even number of bytes.

Figure 20–29 illustrates the accessing of Str255 characters.

Now, back to the Check_For_Valid_Score() routine. We find the length of the user-entered Score_Str because we want to examine each character in it. We perform two checks. The first is to verify that the user entered only digits—integers 1 to 9. This is a test you may find useful in many of your programs. Remember that just because you ask the user to type in a number, you can’t assume that the user won’t type 1z0, 44@, ase, or worse! Now you have a way to check for that:

```c
for (index = 1; index <= str_length; index++)
{
    if ((Score_Str[index] < '0') || (Score_Str[index] > '9'))
        valid_number = FALSE;
}
```
If the Str255 survives this first test, you know that at the very least the user typed in all digits. Now, to verify that the digits represent an integer in the range of 0 to 100, we use a handy Toolbox routine called StringToNum() to convert a Str255 to a variable of type long:

```c
StringToNum(Score_Str, &Score_Int);
```

```c
if ((Score_Int < 0) || (Score_Int > 100))
    valid_number = FALSE;
```

Take a close look at the Check_For_Valid_Score() routine—it will yield valuable string information that you will use again and again:

```c
int Check_For_Valid_Score( void )
{
    int       index;
    int       str_length;
    Boolean   valid_number;

    str_length = Score_Str[0];
```
valid_number = TRUE;

for (index = 1; index <= str_length; index++)
{  
    if ((Score_Str[index] < '0') || (Score_Str[index] > '9'))
        valid_number = FALSE;

    if (valid_number == TRUE)
    {
        StringToNum(Score_Str, &Score_Int);

        if ((Score_Int < 0) || (Score_Int > 100))
            valid_number = FALSE;
    }

    return(valid_number);
}

Opening a Window

We're almost at the end of our MacCertificate listing. The next function is a simple one. It loads the WIND resource into memory and opens a window:

/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/

void Open_Certificate_Window( void )
{
    if ((Certificate_Window_Ptr = GetNewWindow(CERTIFICATE_WIND_ID, NIL_PTR, IN_FRONT)) == NIL_PTR)
        Post_Error_Alert(WIND_ERROR_ALRT_ID);

    ShowWindow(Certificate_Window_Ptr);
}

Setting a Check Box

The last routine in MacCertificate—Set_Check_Box()—appeared in ShapeMaker exactly as we have it here:
void Set_Check_Box( DialogPtr the_dialog, short the_item )
{
    Handle item_handle;
    short item_type;
    Rect item_rect;
    int old_value;

    GetDItem(the_dialog, the_item, &item_type, &item_handle, &item_rect);
    old_value = GetCtlValue((ControlHandle)item_handle);

    if (old_value == CONTROL_ON)
        SetCtlValue((ControlHandle)item_handle, CONTROL_OFF);
    else
        SetCtlValue((ControlHandle)item_handle, CONTROL_ON);
}

In Conclusion

With the listing of MacCertificate completed, you should have mastered the basic skills needed to write Macintosh C programs of your own. If you wish, use either ShapeMaker or MacCertificate as the starting point for your own programs. And be sure to take advantage of all the tricks and tips presented in this book and the Simulator software!

Chapter Summary

This chapter listed all the resources and the source code you need to create a stand-alone application that has many of the graphical features found in a typical Macintosh program.
# Appendix—ASCII Codes

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
<th>ASCII Code</th>
<th>Character</th>
<th>ASCII Code</th>
<th>Character</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>64</td>
<td>@</td>
<td>96</td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!</td>
<td>65</td>
<td>A</td>
<td>97</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&quot;</td>
<td>66</td>
<td>B</td>
<td>98</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>#</td>
<td>67</td>
<td>C</td>
<td>99</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$</td>
<td>68</td>
<td>D</td>
<td>100</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>%</td>
<td>69</td>
<td>E</td>
<td>101</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&amp;</td>
<td>70</td>
<td>F</td>
<td>102</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>'</td>
<td>71</td>
<td>G</td>
<td>103</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(</td>
<td>72</td>
<td>H</td>
<td>104</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>)</td>
<td>73</td>
<td>I</td>
<td>105</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>74</td>
<td>J</td>
<td>106</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>+</td>
<td>75</td>
<td>K</td>
<td>107</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>,</td>
<td>76</td>
<td>L</td>
<td>108</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>77</td>
<td>M</td>
<td>109</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>.</td>
<td>78</td>
<td>N</td>
<td>110</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>/</td>
<td>79</td>
<td>O</td>
<td>111</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>80</td>
<td>P</td>
<td>112</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>81</td>
<td>Q</td>
<td>113</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>82</td>
<td>R</td>
<td>114</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>83</td>
<td>S</td>
<td>115</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>84</td>
<td>T</td>
<td>116</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>85</td>
<td>U</td>
<td>117</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>6</td>
<td>86</td>
<td>V</td>
<td>118</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>87</td>
<td>W</td>
<td>119</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>88</td>
<td>X</td>
<td>120</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>89</td>
<td>Y</td>
<td>121</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>.</td>
<td>90</td>
<td>Z</td>
<td>122</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>;</td>
<td>91</td>
<td>[</td>
<td>123</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>&lt;</td>
<td>92</td>
<td>\</td>
<td>124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>=</td>
<td>93</td>
<td>]</td>
<td>125</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>&gt;</td>
<td>94</td>
<td>^</td>
<td>126</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>?</td>
<td>95</td>
<td>_</td>
<td>127</td>
<td>del</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

- `decimal`: ASCII
- Character codes 0-31 and 127 are nonprinting.
- Character code 32 prints a space.
- bs = backspace
- ht = horizontal tab
- cr = carriage return
- esc = escape
Symbols

#define statements, See define statements
OL notation, 262, 307
68000 microprocessor, 2

A
Add_Two_Integers function, 132
addresses, 145, 147
alerts
    creating, 360–362
defined, 359
item numbers, 379
types, 359–360
using, 399–401
ALRT resource type, 359–361
American National Standards Institute,
See ANSI C
animation examples, 201–203
ASCII Table, 40, Appendix
application window, 314
assignment statement, 14, 18, 84
arrays
    applications of, 172–173
    assignment, 163–165
declaration, 161–163
element, 162
global, 172
initialization of, 163
memory usage, 162
name as a pointer, 169, 170, 171
passing elements of, 166–168
passing entire, 169–171
subscript, 162
types of, 163

B
base 2 numbers, 141–145
base 10 numbers, 141
BeginUpdate function, 318, 436
binary numbers, 141–145
bit-mapped, 188
BitAnd function, 270
bits, 140–145
blank lines, 51
body loops, 89
Boolean data type, 402
branching, See conditional statement
break statement, 117
bugs
compiler reporting of, 6
eliminating, 7
types of, 18–19
See also debugging code
Button function, 252, 257
bytes, 140–145

C

C (computer language)
advantages of, 2
case-sensitivity of, 12, 392
origin of, 2
case label, 117, 404
case-sensitivity of, 12, 392
casting, 43, 308, 317, 346–347
centering a string, 222–224, 438
char data type, See character
characters
ASCII representation of, 39, 270
assignment, 40
char data type, 39–40
constants, 34–35
declaration, 41
initialization, 41
Cheerleader function, 127–128
Close_Window function, 316
code
execution of, 14
fragments, 15
comments, 14
compaction, of memory, 159
comparative operators, 91, 93–95, 96
compilers
prototypes and, 134–138
resources and, 230
THINK C, 6, 29
compiling code, 6
compound statement, 89
conditional statement
definition of, 16
program flow and, 84, 99
See also particular conditional statement type
constants
definition of, 35
examples of, 34
conventions, naming, 13, 44, 56, 65, 392, 430
conversion, of data types, 43, 308, 317, 346–347
coordinate system, 189–190
counter variables, 67
CPU, 2
creating resource files, 233, 244–245, 369–370
creating resources, 231, 232–238, 276, 280,
326–330

data types
ANSI C, 32
Macintosh C, 32
mixing of, 78
See also particular data type
debugging code, 7, 18–19, 61
declaration statement, 12, 17, 84
decrementing variables, 79–80
default label, 117
define statements
advantages of, 44–45, 390–391
definition of, 44
dereferencing pointers, 149–150
desk accessories, 265, 276, 289, 302, 381, 400–
401, 445–446
Determine_Minimum function, 133–135
dialog boxes
closing, 343
creating, 338–340
DialogPtr data type, 337–338
DialogRecord data type, 336–338
DisposDialog function, 343, 355
drawing to, 339, 402, 410–411
erasing contents of, 409
GetNewDialog function, 338–340, 402
handling of, 403–404
items, See dialog box items
modal, 323, 340–343
ModalDialog function, 340–342
modeless, 323
numbering resources, 382
opening, 402
ShowWindow function, 338–339
types, 323–324
dialog box items
accessing values from, 343–359
adding items, 372–373
defined, 324–326, 330–336
check boxes, 325, 344–348, 357,
405–407
dials, 325edit text, 326, 334, 343, 352–354, 359,
408–409
enabling, 332, 340
GetCtlValue function, 345–347, 406
GetDltem function, 344–345, 349–350,
352, 405
GetlText function, 352–354
handles to, 344–348
labels of, 330–332
numbering of, 330, 332–334, 373–376,
378, 382, 390–391
pushbuttons, 325, 330–332, 344,
372–373
radio buttons, 325, 333, 348–352, 357–
358, 447–451
SetCtlValue function, 345, 347, 349–350,
451
static text, 326, 334, 342
user items, 326
Dialog Manager, 322
DialogPtr data type, 337–338
DialogRecord data type, 336–338
DisposeDialog function, 343, 355
DisposeWindow function, 311–312
DITL resource type, 238, 326, 330–336
DLOG resource type, 238
do-while loop, 103–106
drawing,
pictures, 441–443
See also particular shape to draw
DragWindow function, 315, 443
DrawChar function, 34, 21
DrawPicture function, 442
DrawString function, 34, 216–219, 250
drivers, 289
DRVR resource type, 289
E
editing resources, 229–231
derients, 112–116
EndUpdate function, 318, 436
EraseOval function, 211
EraseRect function, 200–201
EraseRoundRect function, 212–213
erasing graphics, 409
error checking, 431–432, 448–449,
453–457
events
definition of, 25, 241
driven programs, 25
everyEvent, 257
FindWindow function, 292–293
GetNextEvent function, 256–257, 261–
264, 291
handling, 397–398
keyDown event, 266, 269
loop, 25, 252–254, 257–261, 264
mask, 256–257
menu handling and, 290–297
mouseDown event, 260, 266, 269, 292
processing, 252–253
queue, 25, 254
record, 254–256, 270
retrieving, 256–257
types of, 242–243, 255
updateEvt event, 260
WaitNextEvent function, 261–263
event mask, 256–257
Event program, 243–254
Event2 program, 265–270
everyEvent, 257
executable code, 6
execution of code
definition of, 14
program flow, 52
ExitToShell function, 432
expressions, 82

F

files
resource, 23, 230
source code, 5, 246–247
FillOval function, 211
FillRect function, 207–208
FillRoundRect function, 212–213
FindWindow function, 292–294, 314–316, 397
flag variables, 402, 407
float data type, See floating-point
floating-point
constants, 34–35
examples of, 41
float data type, 41–42
truncation of, 77
using, 42
fonts, 219–220
for loop, 100–102, 104–106
fragmentation of memory, 158–159
fragments, See code fragment
FrameOval function, 210–211
FrameRect function, 200–201
FrameRoundRect function, 212–213
functions
advantages of, 57–62
clarity of, 59–61
effect on program flow, 52–54
invoking, 14, 51
naming of, 56
parameters, 127, 129–132, 150–154
prototypes, 134–138, 388–390
purpose of, 12, 17
return value of, 70, 132–134, 150–151
reusability of, 58
See also specific function name

G

Get_Dialog_Info function, 355
Get_Text_From_Edit function, 353
Get_User_Input function, 70, 92–93, 95, 116
GetCtlValue function, 345–347, 406
GetDItem function, 344–345, 349–350, 352, 405
GetText function, 352–354
GetMHandle function, 286–289
GetNewDialog function, 338–340, 402
GetNewWindow function, 250, 307–309, 338
GetNextEvent function, 256–257, 261–264, 291
GetNMBar function, 286
GetPicture function, 441–443
GetPort function, 403, 435
global variables, 64–65, 67–68, 407
GrafPort data type, 225, 302–306
graphical user interface, 22
graphics pen
definition of, 196
drawing with, 402
PenSize function, 196–199
thickness, 196
graphics ports
definition of, 224
GrafPort data type, 225
SetPort function, 225
GUI, 22

H

Handle_Apple_Choice function, 297
Handle_File_Choice function, 297
Handle_Menu_Choice function, 398–399
Handle_Mouse_Down function, 290–297, 313–314, 397–398
Handle_One_Event function, 263–265, 290, 396–397
handles, 158–159, 344–345
header files, 247–249
HiliteMenu function, 399
HiWord function, 295–296, 398–399

I

if statement, 107–110, 121–123
if-else statement, 110–112
IN_FRONT constant, 266
incrementing variables, 79–80
index variables, 67
infinite loops, 90
Initialize_Toolbox function, 267
inMenuBar constant, 294
input
  user, 69–70
  See also Get_User_Input function
Inside Macintosh books, 56, 191
int data type, See integer
integer
  assignment, 38
  constants, 34–35
  declaration, 38
  examples of, 37
  initialization, 38
  int data type, 12
  range, 37
InvertOval function, 211
InvertRect function, 209–210
InvertRoundRect function, 212–213
J-K
keyDown event, 266, 269
keystroke handling, 269–270
keywords, 12
L
L notation, 262, 307
Line function, 193–194
lines
  drawing, 193–194
  Line function, 193–194
  LineTo function, 193–194
LineTo function, 193–194
linking, 6
local variables, 62–64, 66–68, 397
logical operators, 119
long data type 39
  declaration, 39
  examples of, 39
  L notation, 262, 307
  menu items and, 295–296
  range, 39
loops
  body, 89
  choosing type, 104–106
  incrementing counter, 90
  infinite, 90
  See also particular loop type
LoWord function, 295–296, 398–399
M
MacCertificate program
  define statements, 428–429
  function prototypes, 427
  global variables, 429
  resources
    ALRT, 419–423
    DITL, 416–419
    DLOG, 416
    MBAR, 423–425
    MENU, 423
    PICT, 426
    WIND, 425
  running, 414
MacHeaders, 247–249
Macintosh
  CPU, 2
  hardware, 2
  User Interface Toolbox, See Toolbox
Macintosh C
  combined with ANSI C, 9, 194–196
  types, 247
Macintosh program structure, 24–25
MacTraps, 247
Main_Event_Loop function, 264
main function, 11
maintenance of programs, 61
Make_A_Window function, 309–311
managers
  definition of, 22
  Window, 299–300
  Menu, 274
  Dialog, 322
mathematical precedence, 80–82
MBAR resource type, 275, 283–285
memory
  addresses, 145, 147
  allocating, 36
  bits, 140–145
  bytes, 140–145
compaction, 159
fragmentation, 158–159
number representation in, 141–145
reserving, 36
voltages in, 141
Menu Manager, 274
MENU resource type, 238, 275–283
MenuHandle data type, 286, 289
menus
adding items to, 280–283
bar, 275
dashed line in, 276–277
desk accessories and, 276–278, 289, 399–401
determining menu item clicked, 295–296
events and, 290–297
FindWindow function, 294
GetNewMBar function, 431
highlighting, 294, 399
inMenuBar constant, 294
items, 275
loading to memory, 286–288
MenuHandle data type, 286, 289
MenuSelect function, 293, 397–399
mouseDown event, 295, 398–399
OpenDeskAcc function, 400–401
resource IDs, 279
responding to click in, 293–297
MenuSelect function, 293, 397–399
mixed mode, 42, 78
modal dialog, 323, 340–343
ModalDialog function, 340–342
modeless dialog, 323
mouseDown event, 260, 266, 269, 292, 295, 398–399
mouse events, 313–316
Move function, 193–194
MoveTo function, 193–194, 217–218, 250
MultiFinder, 261–262, 395–396
N
naming conventions, 13, 44, 56, 65, 392, 430
naming resource files, 23, 371
NGetTrapAddress function, 262, 395–396
NIL_PTR constant, 266
nil pointers, 266, 307, 431
NoteAlert function, 399
operator
OpenDeskAcc function, 400–401
addition (+), 75, 76
and, logical (&&), 119–120
assignment (=), 74, 75–76, 97
assignment variations, 97
basic, 75
comparative, 91, 93–95, 96
decrement (--), 79–80
division (/), 77
equality, 94
greater than (>), 75, 94
greater than or equal to (>=), 94
increment (++), 79–80
inequality, 95
less than (<), 93
less than or equal to (<=), 94
logical, 119
math, 76
multiplication (*), 77
not, logical (!), 120
or, logical (||), 120, 455
pointer declaration (*), 147
processing data with, 2
sign, 78
structure member (.), 177–179
structure member (->), 183–184
subtraction (->), 76
unary, 78
origin of screen, 189
ovals
boundaries of, 210–211
Toolbox drawing functions, 210–211
P
PaintOval function, 211
PaintRect function, 207–208
PaintRoundRect function, 212–213
parameters, 127, 129–132, 150–154
part codes, 314
passed parameters, See parameters
Pattern data type, See patterns
patterns
data type, 206
lists of, 207
PenPat function, 207
standard patterns, 206–207
PenPat function, 207
PenSize function, 196–199
PicHandle data type, 441–443
Picture data type, 442
pictures, drawing, 426
pixels, 188–191, 215
planning of programs, 3–4, 27, 61
pointers
changing parameter values with, 154–158
declaration, 147
definition of, 145–146
dereferencing, 149–150
giving value to, 147–148
point at what, 146–149
size of, 148
swapping variable values with, 154–158
post-execution check, 103
postfix mode, 79
prefix mode, 79
precedence, 80–82
pre-execution check, 98, 102
printf function, 68, 217
program
ending, 270, 401
structure, 24–25
programming methods
debugging, 61
maintenance, 61
planning, 3–4, 27, 61
process, 3–9,
testing, 6
projects, See THINK C projects
pseudo code, 27–29, 257–260

Q-R
QuickDraw

animation and, 201–203
shapes, See particular shape
uses of, 187–188
real numbers, See floating-point
rectangles
boundaries of, 199
EraseRect function, 200–201
FillRect function, 207–208
FrameRect function, 200–201
InvertRect function, 209–210
PaintRect function, 207–208
Rect data type, 198–199
round, round rectangles
SetRect function, 198–201
Rect data type, 198
ResEdit
editing resources, 229–231
creating resource file, 233, 244–245, 369–370
creating resources, 231, 232–238, 276, 280, 326–330
MiniScreen, 235
mistakes using, 422
obtaining, 24, 231
quiting, 238
running, 232, 244
reserved words, 12
resources
creating, See ResEdit
definition of, 23
DLOG/DITL relationship, 329, 335
editing, 230
files, 23, 230
getting information about, 371
IDs, 279, 329, 421
interaction with Toolbox, 24
missing, 421–422
naming, 23, 371
reusability, 23
titles, 279–280
types
ALRT resource type, 359–361
DLOG resource type, 238
DITL resource type, 238, 326, 330–336
DRVR resource type, 289
MBAR resource type, 275, 283–285
MENU resource type, 238, 275–283
WIND resource type, 234–237

See also ResEdit

return keywords, 151
return values, 70, 132–134, 150–151
Ritchie, Dennis, 2, 9
ROM
    Toolbox functions and, 54–55, 287–288
round rectangles
    boundaries of, 212–213
    Toolbox drawing functions, 212–213
routines, See functions

S

scope
    definition of, 62
screen
    coordinate system, 189–190
    origin, 189
    pixels, 188
screenBits global variable, 315–316, 433
SelectWindow function, 316, 443
semantic errors, 19
Set_Check_Box function, 347–348, 405
Set_Radio_Button function, 351
SetCtlValue function, 345, 347, 349–350, 451
SetMenuBar function, 286
SetPort function, 225, 309–310, 402, 435–436
SetRect function, 198–201
ShapeMaker program
    defines, 390–392
    description, 366–369
    function prototypes, 388–390
    global variables, 392–393
    resources
        ALRT, 377–379
        DITL, 372–377, 379
        DLOG, 370–372
        MBAR, 383–384
        MENU, 379–383
Sharks, 17, 108
short data type, 341
ShowWindow function, 308–309, 338–339
signed data type, 37
Simulator software
    code fragments and, 26
    purpose of, 27
sorting, 154
sound, 269
source code
    comments in, 14
    compiling, 6
    file, 5, 246–247
    reusing, 445
statement
    assignment, 14, 18, 84
    break, 117
    C instruction, 12
    conditional, 16, 84
    declaration, 12, 17, 84
    definition of, 83
    comparison vs. assignment, 94–95
    compound, 89
Str255 data type, See text, Str255 data type
StringToNum function, 354–355, 456
StringWidth function, 223–224, 438
struct, See structure
structure
    accessing members of, 176–179, 183–184
    arrays of, 179–180
    body of, 174
    declaration, 174
    members, 174
    passing entire, 182–184
    passing members of, 180–182
    pointers to entire, 183–184
    pointers to members of, 181–182
    tag, 174
    template, 174
    types of, 174
    user-defined, 175
    variables of type, 175–176
structured coding, 49
Style data type, 221
style of text, 221–222
swapping variable values, 154–158
switch statement, 116–119, 121–123
Symantec, See THINK C projects
syntax errors, 18–19
SysBeep function, 269
system window, 314
SystemClick function, 314–315, 397
SystemTask function, 264

T

testing code, 6
text
centering, 222–224, 438
characteristics, 219–222
converting to number, 353–354
displaying of, 34, 215
DrawChar function, 34, 219
DrawString function, 34, 216–219
face, 221–222
fonts, 219–220
graphical representation of, 215
length of string, 453
lines of, 217–218
pixels and, 215
size of, 220–221
storage in memory, 455
Str255 data type, 216, 218–219, 353, 453–457
StringToNum function, 354–355, 456
StringWidth function, 223–224, 438
style, 221–222
Style data type, 221
TextFace function, 221–222
TextFont function, 220
TextSize function, 220–221

THINK C projects
about the compiler, 6, 29
adding a file, 247
bringing up to date, 247
creating, 246
error messages, 248
MacHeaders, 247–249
MacTraps, 247
options, 248–249
naming, 246
resource files, 246

running a program, 249
Thompson, Ken, 2, 9

Toolbox
accessing routines, 192
functions, 16, 22, 54–56, 287–288
initializing, 192–193, 249, 267, 394
interaction with resources, 24
managers, 22
relationship with ROM, 54–55, 287–288
TrackGoAway function, 316, 443
truncation, 43, 77
truth
expression of a, 90
value of, 90–91
type casting, 43, 308, 317, 346–347

UNIX, 2
updateEvt event, 260
user-defined types, 175
user input, See input
User Interface Toolbox, See Toolbox

variables
declaration, 12, 17
decrementing variables, 79–80
definition of, 35
flag, 402, 407
giving value to, 14, 18,
global, 64–65, 67–68, 407
incrementing variables, 79–80
local to functions, 62–64
memory usage, 66–67
naming, 13, 44, 65, 392, 430
value retention, 65–66

void keyword, 135

WaitNextEvent function, 261–263, 396
while loop, 88–90, 97–100, 104–106
white space, 51
WIND resource type, 234–237, 244, 250, 306
window

application type, 314
BeginUpdate function, 318, 436
closing, 311–312
creating, 68–69, 307–309
DisposeWindow function, 311–312
dragging, 315–3126, 433
DragWindow function, 315, 443
drawing of, 300–301
drawing to, 309–310, 311
elements of, 301–302
EndUpdate function, 318, 436
FindWindow function, 314–316
GetPort function, 403, 435
GetNewWindow function, 250, 307–309
GrafPort data type, 302–306
inContent constant, 314, 316
inDrag constant, 314, 315
inGoAway constant, 314, 316
inSysWindow constant, 314–315
loading to memory, 267–268
Make_A_Window function, 309–311
mouse events in, 313–316
pixel origin, 190–191
port saving, 402
SelectWindow function, 316, 443
SetPort function, 309–310, 402, 435–436
ShowWindow function, 308–309
SystemClick function, 314–315, 397
system type, 314
TrackGoAway function, 316, 443
types, 302
update events in, 316–319, 435–443
visible, 308
WIND resource, 250, 306
WindowPtr data type, 293, 305–309, 437
WindowRecord data type, 304–306
WindowPtr data type, 293, 305–309, 437
WindowRecord data type, 304–306
**Prima Computer Books**

Available Now

<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WINDOWS Magazine Presents: Access from the Ground Up</strong></td>
<td>19.95</td>
</tr>
<tr>
<td>Advanced PageMaker 4.0 for Windows</td>
<td>27.95</td>
</tr>
<tr>
<td>DESQview: Everything You Need to Know</td>
<td>22.95</td>
</tr>
<tr>
<td>DOS 6: Everything You Need to Know</td>
<td>24.95</td>
</tr>
<tr>
<td><strong>WINDOWS Magazine Presents: Encyclopedia for Windows</strong></td>
<td>27.95</td>
</tr>
<tr>
<td>Excel 4 for Windows: Everything You Need to Know</td>
<td>24.95</td>
</tr>
<tr>
<td>Excel 4 for Windows: The Visual Learning Guide</td>
<td>19.95</td>
</tr>
<tr>
<td><strong>WINDOWS Magazine Presents:</strong> Freelance Graphics for Windows: The Art of Presentation</td>
<td>27.95</td>
</tr>
<tr>
<td>Harvard Graphics for Windows: The Art of Presentation</td>
<td>27.95</td>
</tr>
<tr>
<td>LotusWorks 3: Everything You Need to Know</td>
<td>24.95</td>
</tr>
<tr>
<td>Microsoft Works for Windows By Example</td>
<td>24.95</td>
</tr>
<tr>
<td>NetWare 3.x: A Do-It-Yourself Guide</td>
<td>24.95</td>
</tr>
<tr>
<td>Novell NetWare Lite: Simplified Network Solutions</td>
<td>24.95</td>
</tr>
<tr>
<td>PageMaker 4.0 for Windows: Everything You Need to Know</td>
<td>19.95</td>
</tr>
<tr>
<td>PageMaker 4.2 for the Mac: Everything You Need to Know</td>
<td>19.95</td>
</tr>
<tr>
<td>PageMaker 5 for Windows: Everything You Need to Know</td>
<td>19.95</td>
</tr>
<tr>
<td><strong>WINDOWS Magazine Presents:</strong> The Power of Windows and DOS Together, 2nd Edition</td>
<td>24.95</td>
</tr>
<tr>
<td>Quattro Pro 4: Everything You Need to Know</td>
<td>19.95</td>
</tr>
<tr>
<td>QuickTime: Making Movies with Your Macintosh</td>
<td>24.95</td>
</tr>
<tr>
<td>Smalltalk Programming for Windows (with 3½&quot; disk)</td>
<td>39.95</td>
</tr>
<tr>
<td>Superbase Revealed!</td>
<td>27.95</td>
</tr>
<tr>
<td>SuperPaint 3: Everything You Need to Know</td>
<td>24.95</td>
</tr>
<tr>
<td>Windows 3.1: The Visual Learning Guide</td>
<td>19.95</td>
</tr>
<tr>
<td>Word for Windows 2: The Visual Learning Guide</td>
<td>19.95</td>
</tr>
<tr>
<td>Word for Windows 2 Desktop Publishing By Example</td>
<td>24.95</td>
</tr>
<tr>
<td>WordPerfect 5.1 for Windows Desktop Publishing By Example</td>
<td>24.95</td>
</tr>
<tr>
<td>WordPerfect 6 for DOS: The Visual Learning Guide</td>
<td>19.95</td>
</tr>
</tbody>
</table>
**To order by phone** with Visa or MasterCard, call (916) 786-0426, Monday–Friday, 9 a.m.– 5 p.m. Pacific Standard Time.

**To order by mail** fill out the information below and send with your remittance to: Prima Publishing, P.O. Box 1260, Rocklin, CA 95677-1260

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Title</th>
<th>Unit Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subtotal

7.25% Sales Tax (CA only)

Shipping*

Total

Name

Street Address

City          State          Zip

Visa/MC No.          Expires

Signature

*$4.00 shipping charge for the first book and $0.50 for each additional book.
About These Disks

The Simulator software is a Macintosh program created especially for this book, *Think THINK C*. This software tutorial will teach you how to program your Macintosh using the C language. Through the use of text, graphics, and simulations, the hundreds of screens of the Simulator will bring to life the concepts taught in the book.

Each of the two 800K disks contains one self-extracting compressed file. When expanded, these two files give you a single 4MB program. Self-extracting means that you need no additional software to decompress these files—everything you need is right here on these two disks.

The Simulator software runs on any Macintosh running System 6.0.1 or greater—including System 7. The Simulator will run on either a black-and-white or a color system with any size monitor.

For detailed instructions on how to place the Simulator software on your hard drive, read the subsection Installing the Simulator Software in the Introduction of this book.

Notice of Limited Warranty

By opening the sealed disk containers in this book, you agree to the following terms and conditions. If, upon reading the following disclaimer and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disks to the place where you purchased it for a refund.

The enclosed disks are warranted by Prima Publishing to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disk combination. During the sixty-day term of the limited warranty, Prima will provide a replacement disk upon the return of a defective disk.

The remedy for breach of this limited warranty shall consist entirely of replacement of the defective disks and shall cover no other damages, including loss or corruption of data, changes in the functional characteristics of the hardware or operating system, deleterious interaction with other software, or any other special, incidental, or consequential claims that may arise.

Prima and the author specifically disclaim any and all other warranties, either express or implied, including warranties of merchantability, suitability to a particular task or purpose, or freedom from errors. In no event shall Prima or the author be liable for damages in excess of the purchase price paid for the book/disk combination, even if Prima and/or the author have previously been notified that the possibility of such damages exists.

The enclosed software is copyrighted by the copyright holders of the book and may not be copied except onto one hard disk for use by a single concurrent user or to a backup disk, and may not be distributed for profit, rented, or used for other commercial purposes except by permission of the copyright holders.
Simulator C 1.0
to accompany the book
Think THINK C
Disk 1

© 1993 Dan Parks Sydow.
800K 1 of 2

Simulator C 1.0

to accompany the book
Think THINK C
Disk 2

© 1993 Dan Parks Sydow.
800K 2 of 2
The Fastest, Easiest Way to Learn How to Program Your Macintosh!

The revolutionary approach of this book is centered around Macintosh Simulator C, a computer-based training tool that teaches programming concepts and techniques interactively. The Macintosh Simulator C software included with this book is an integral part of the book itself!

Breakthrough Interactive Disk Bridges the Gap to Understanding!

There's no better way to learn Macintosh programming than with Think THINK C! with Macintosh Simulator C. This multimedia software teaching tool is the next best thing to having a private tutor in your home – for a fraction of the cost.

What's so great about Macintosh Simulator C? It parallels the book's 20 chapters, bringing them to life with on-screen text, graphics, interactive questions and answers, and – best of all – more than 100 animated simulations of the true programming environment. Using this all-new way to learn practical programming skills, you can:

• Run sample programs with the disk, then turn to the book for an in-depth explanation
• Interact with the simulation software, getting immediate feedback to reinforce learning
• Move at your own pace, running simulations again and again until you are comfortable with the subject

DAN PARKS SYDOW is a professional software engineer who has developed educational software for the Macintosh. He currently programs the Macintosh for medical nuclear imaging purposes at St. Luke's Medical Center in Milwaukee, Wisconsin.

User Level: Beginning to Intermediate

PRIMA COMPUTER BOOKS
An imprint of Prima Publishing

U.S. $39.95
U.K. £37.60 Net (Inc. VAT)
Can. $55.95

Macintosh/Programming