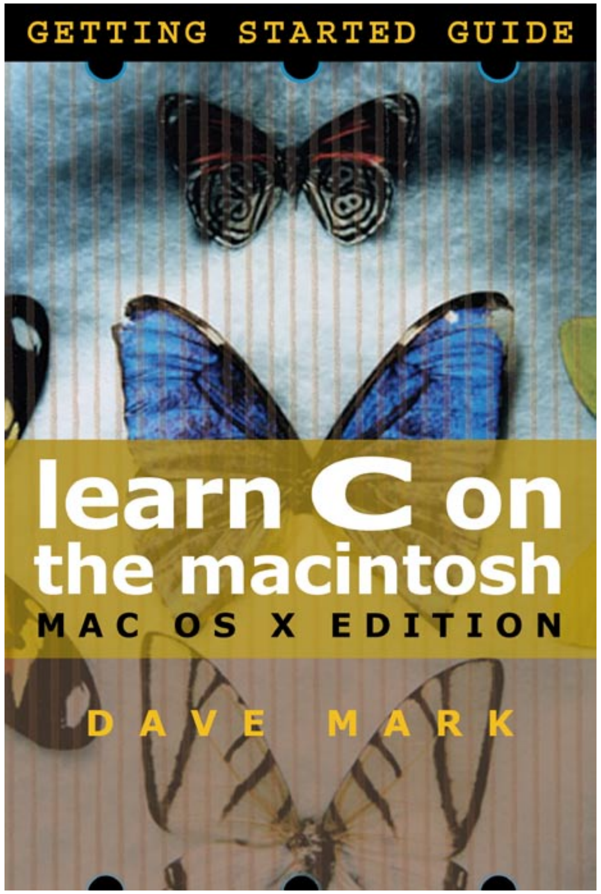


$14.95 USD ISBN: 0-9744344-1-8

Version 1.0 - Check for Updates

Table Of Contents

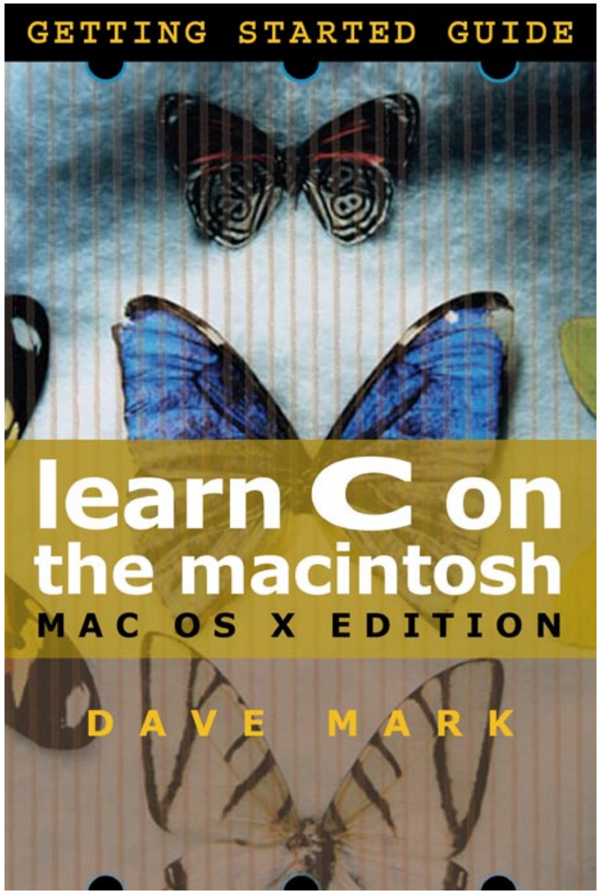


Table Of Contents Continued

About This eBook

The Author

Dave Mark is a long-time Mac developer and author and has written a number of books on Macintosh development, including *Learn C on the Macintosh*, *The Macintosh Programming Primer* series, and *Ultimate Mac Programming*. Dave is the Editor-in-Chief of *MacTech Magazine* and has been writing for *MacTech* since its birth.

The Publisher

Optimized for easy on-screen reading, yet perfect for printing, **SpiderWorks** eBooks are uniquely formatted and hyperlinked for fast access and quick learning.

Cover Design: **Mark Dame**

Interior Page Design: **Robin Williams**

PDF Production: **Dave Wooldridge**

On-Screen Viewing

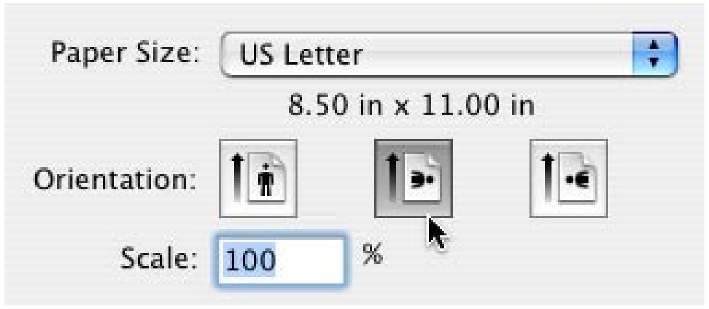
We recommend using Adobe Acrobat or the free Adobe Reader to view this ebook. Apple Preview and other third-party PDF viewers may also work, but many of them do not support the latest PDF features. For best results, use Adobe Acrobat/Reader.

To jump directly to a specific page, click on a topic from either the Table of Contents on the first page or from the PDF Bookmarks. In Adobe Reader, the PDF Bookmarks can be accessed by clicking on the Bookmarks tab on the left side of the screen. In Apple Preview, the PDF Bookmarks are located in a drawer (Command-T to open).

If your mouse cursor turns into a hand icon when hovering over some text, that indicates the text is a hyperlink. Table of Contents links jump to a specific page within the ebook when clicked. Text links that begin with “http” or “ftp” will attempt to access an external web site or FTP server when clicked (requires an Internet connection).

Printing

Since SpiderWorks ebooks utilize a unique horizontal page layout for optimal on-screen viewing, you should choose the “Landscape” setting (in Page Setup) to print pages sideways on standard 8.5” x 11” paper. If the Orientation option does not label the choices as “Portrait” and “Landscape”, then choose the visual icon of the letter "A" or person’s head printed sideways on the page (see example below).


Paper Size: US Letter
8.50 in x 11.00 in
Orientation:
Scale: 100 %

Installing the Project Files

Requirements

The collection of companion project files from this book are contained in a download called *LearnC-Projects.sit*, which can be downloaded from the SpiderWorks Customer Download Center at:

http://www.spiderworks.com/extras/

To login, you will need your Customer Username and Password that was listed in your order confirmation e-mail.

Installation

Once you have downloaded and decompressed *LearnC-Projects.sit* (using Stuffit Expander), you will see a directory called *Learn C Projects*. Nested inside that directory are further subdirectories labeled for the chapter to which the project files apply. Not all chapters have project files in the *Learn C Projects* collection. Move the *Learn C Projects* directory to a convenient location on your hard disk from which you can open the files with Apple's Xcode Tools.

Chapter 1 Welcome Aboard

Welcome! Chances are, you are reading this because you love the Macintosh. And not only do you love the Mac, but you also love the idea of learning how to design and develop your very own Mac programs.

You've definitely come to the right place.

This book assumes that you know how to use your Macintosh. That's it. You don't need to know *anything* about programming. Not one little bit. We'll start off with the basics and each step we take will be a small one to make sure that you have no problem following along.

This is the first in a series of books designed to teach you how to design and build your own Macintosh applications. The first book will focus on the basics of programming. At the same time, you'll learn C, one of the most widely used programming languages in the world. And once you know C, you'll have a leg up on learning programming languages like C++, Java, Objective-C and Microsoft's new C# (pronounced C-sharp), all of which are based on C. If you are going to write code these days, chances are you'll be writing it in C or in one of these other languages.

Once you get through the first few books, you'll be ready to move on to object oriented programming and Objective-C, the official language of Mac OS X. Not to worry. Each book takes the same basic approach: small steps, nobody gets lost. You can *definitely* do this!

Who is This Book For?

When I wrote the very first edition of Learn C back in 1991, I was writing with college students in mind. After all, college was where I really learned to program. Hrm. Seems I was way off. My first clue that I had underestimated my audience was when I started getting emails from fifth graders who were making their way through the book. *Fifth* graders! And not just one. *Lots* of 9, 10, 11 year old kids were digging in and learning to program. Cool! And the best part of all was when these kids started sending me actual shipping products that they created. You can't imagine how proud I was and still am.

Over the years, I've heard from soccer moms, hobbyists, even from folks who were using the Mac for the very first time, all of whom made their way through *Learn C* and came out the other end, proud, strong, and full of knowledge.

So what do you need to know to get started? Although it is possible to learn C just by reading a book, you'll get the most out of this book if you run each example program as you encounter it in the book. To do this, you'll need a Macintosh running Mac OS X (preferably version 10.3 or later) and an internet connection. You'll need the internet connection to download the free tools Apple has graciously provided for anyone interested in programming the Mac.

If you know nothing about programming, don't worry. The first few chapters of this book will bring you up to speed. If you have some programming experience (or even a lot), you might want to skim the first few chapters, then dig right into the C fundamentals that start in Chapter 3.

Ready to get started? Let's go…

Chapter 2 Go Get the Tools!

Before we dig into the specifics of programming, you'll need to download a special set of tools from Apple's web site. The good news is, these tools are absolutely *free* – Cool! And more importantly, Apple's tools give you absolutely everything you'll need to create world class Macintosh programs, whether they be written in C, Objective C, Java, or even C++. And did I mention that these awesome tools are free?

Before you start downloading the tools, be aware that this is one *big* download. The version I downloaded today was broken into 13 segments, most of which were about 29 megabytes *each*, for a total download of more than 372 megabytes. Ouch!

Obviously, you won't want to tackle this task via a dialup, and you'll want to make sure you have plenty of hard drive space available before you begin.

If you don't have the bandwidth, there is an alternative! Follow all the steps for creating an account and logging in to Apple's ADC web site below. Then, instead of clicking on the *Download Software* link, click on the *Purchase* link instead. At the bottom-right of the page that appears, click on the *Developer Tools* button. This will take you to a page where you can buy the tools CD. Cool!

To gain access to these tools, go to Apple's web site and sign up as a member of the *Apple Developer Connection* (ADC) program. Here's a link to the front page of the ADC site:

http://developer.apple.com

You'll want to bookmark this page in your browser so you can refer to it later. In fact, you might want to create a *Learn C* bookmark folder in your browser just for web sites we mention in this series.

Don't let the sheer volume of information on this site overwhelm you. Over time, it will start to make a lot more sense. For now, let's get in, get the tools, then get out. No need to linger just yet.

Create an ADC Account

Before Apple will let you download the tools, you'll first need to join ADC (remember, it's free!) To join, click on the *Not a Member?* link in the grey bar towards the top of the page. This will bring you to the *Membership Overview* page. This page tells you about the different ADC memberships available. For now, we'll take the *Online membership* option. The Online option is free and still gives us access to the tools we'll need.

To start the sign up process, click on the blue link that says *Apple Developer Connection member* or just go to this page:

http://connect.apple.com

Click the button labeled *Join ADC Now*.

Read the license agreement, then click on the *Agree* button.

Next, you'll be prompted to enter your name and email address and to select an Apple ID and a password. Pretty straightforward.

Once you've entered and confirmed your password, click the *Continue* button.

If your Apple ID is already taken, a red error message will appear. Pick a different Apple ID and try again. Eventually, you'll find one that isn't already used.

The next step is to provide clues in case you forget your password. You'll provide your birth date, a question Apple can ask you, and the answer to the question. Click the *Continue* button.

Next, you'll come to a page that asks you to fill out your *Developer Account Profile*. Do this, then click the *Save* button.

That should do it. Congratulations, you are now a proud member of Apple Developer Connection! Be sure to copy down your Apple ID and your password. You'll need this info every time you come back to the site.

Download the Tools

Once you have your Apple ID, you can login to the ADC site by going to this link:

http://connect.apple.com

This is the page with the *Join ADC Now* button. On the right side of that page is a place for you to type in your *Apple ID* and *Password*. Click the *Continue* button to login.

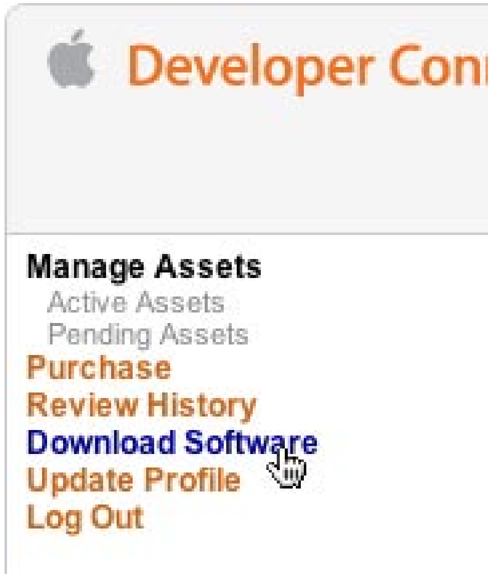


Figure 2.1 *Once logged in to ADC, click this link to download software.*

Once you are logged in, you'll see a list of links similar to those shown in Figure 2-1. Click on the *Download Software* link. This will take you to a page listing the most recent tools available for download. Scroll down the list looking for something titled *Xcode Tools*, followed by a version number.

If you don't see any Xcode downloads available on the main *Download Software* page, click on the *Developer Tools* link that appears below the *Download Software* link to reveal a more extensive list of downloads (see Figure 2-2).

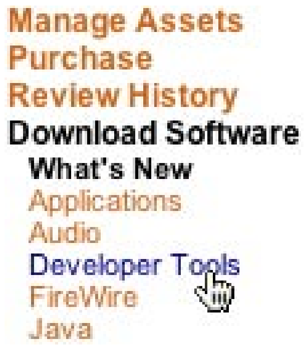
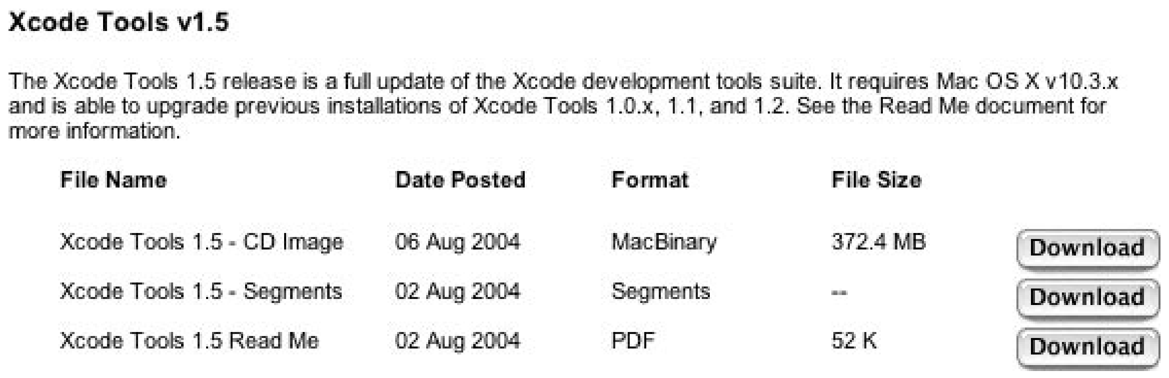


Figure 2.2 *You may find Xcode in the* Developer Tools *section.*

Figure 2-3 shows the download section for *Xcode Tools v1.5*, which was the most recent version of Xcode available when I wrote this chapter.



Xcode Tools v1.5

The Xcode Tools 1.5 release is a full update of the Xcode development tools suite. It requires Mac OS X v10.3.x and is able to upgrade previous installations of Xcode Tools 1.0.x, 1.1, and 1.2. See the Read Me document for more information.

File Name	Date Posted	Format	File Size	
Xcode Tools 1.5 - CD Image	06 Aug 2004	MacBinary	372.4 MB	Download
Xcode Tools 1.5 - Segments	02 Aug 2004	Segments	--	Download
Xcode Tools 1.5 Read Me	02 Aug 2004	PDF	52 K	Download

Figure 2.3 *When I wrote this chapter, this was the latest version of Xcode available for download.*

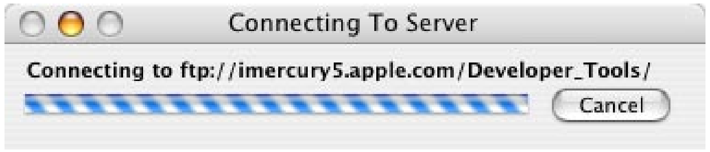
Downloading the Segments

At this point, you've signed up for an ADC account, logged in, and located the Xcode tools. Chances are, you'll see something like the screenshot shown in Figure 2-3. Feel free to download the *Read Me*, but the real goal is to download either the full CD image, hidden behind the first *Download* button, or the set of segments that make up the *Xcode Tools 1.5 CD*, hidden behind the second *Download* button.

When you click on either *Download* button, Mac OS X will attempt to connect to an FTP server using whatever you have defined as your default FTP application. If you've never used an FTP application before, you might want to either find an FTP-savvy buddy who can help you through this process or click the *Purchase* link (as described at the beginning of the chapter) and order the Xcode CD from Apple.

Note that your browser may try to handle FTP requests all by itself. Safari, for example, will download single files without help, but will hand off a request for a directory of files to the default FTP client.

If you don't own an FTP application, the Finder *can* do the job for you. For example, when I clicked on the *Download* button in my browser window, the window shown in Figure 2-4 appeared, then a server named *segments* appeared on my desktop, just like a newly mounted hard drive.



Connecting to ftp://imercury5.apple.com/Developer_Tools/

Figure 2.4 *This window appears when the Finder tries connecting to Apple's server.*

If you double-click on the *segments* icon, a Finder window will appear listing all the files you need to download. Create a new folder (I called mine *Xcode parts*) and drag all the files from the *segments* window to that new folder. The files for Xcode 1.5 took up about 372 megabytes of hard drive space (be *sure* you have enough space on your hard drive before you start – In fact, be sure you have at least a couple of gigabytes free, just to be safe) and took about 45 minutes to copy using a cable modem. Your mileage may vary!

That said, I've found that using the Finder as an FTP client a hit-or-miss proposition. An application specifically written to do FTP will get the job done much more quickly.

There are a number of excellent FTP clients that run under *Mac OS X*. One that I've been using for a number of years is called *Interarchy*. You can find it at:

http://www.interarchy.com

Interarchy is amazing! It handles pretty much everything I throw at it (including FTP, SFTP, HTTP, ping, traceroute, DNS lookup, and packet sniffing), and is both fast and reliable. When you download Interarchy, it automatically runs in demo mode allowing you to test it before you buy it.

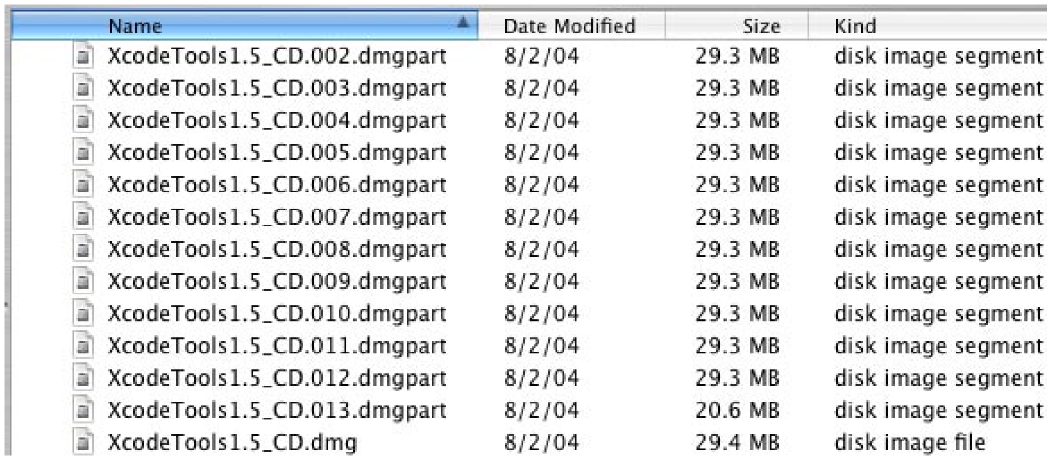
If you do run Interarchy, when it asks you if you want it to be your default FTP application, say yes. Now, if you go back to the ADC site and click the *Download* button again, your web browser should use Interarchy to open the server instead of the Finder.

Putting Mr. Dumpty Back Together Again

Whether you used the Finder or an FTP client like Interarchy, at this point, you should have a folder on your hard drive containing one file ending with *.dmg* and a series of consecutive files ending with *.dmgpart*. Figure 2-5 shows my Finder listing after I downloaded the parts that make up the Xcode version 1.5 installer. If you downloaded the archive as a single piece, instead of in segments, you can skip down a few paragraphs to where I tell you to double-click the *Xcode Tools* icon.

Important - Things change! The most recent version of the ADC web site only offers *Xcode Tools v1.5* as a single archive, not as a series of *.dmgparts*. By the time you read this, there may be a newer version of Xcode, done as a single file or multiple parts. Use your noodle, download the latest, you'll be fine. I'll make sure that the Learn C projects are always updated with the latest and greatest release (i.e., non-beta) version of Xcode.

Note that there is only one *.dmg* file. Once all the pieces are in place, drop the *.dmg.bin* file onto the *StuffIt Expander* icon.



Name	Date Modified	Size	Kind
XcodeTools1.5_CD.002.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.003.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.004.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.005.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.006.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.007.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.008.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.009.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.010.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.011.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.012.dmgpart	8/2/04	29.3 MB	disk image segment
XcodeTools1.5_CD.013.dmgpart	8/2/04	20.6 MB	disk image segment
XcodeTools1.5_CD.dmg	8/2/04	29.4 MB	disk image file

***Figure 2.5** The segments that go together to make the Xcode 1.5 installer.*

Looking for StuffIt Expander? Chances are good it is already on your hard drive. If not, you'll find the latest version here:

http://www.stuffit.com/mac/

StuffIt Expander will do the right thing. First, it will search to make sure you have all the parts needed to reassemble the Xcode installer. Then, it will start patching things together until the *.dmg* file is reassembled. A *.dmg* file is a mountable disk-image, like a fake hard drive. Once the *.dmg* file is mounted, a new volume should appear on your desktop with the name *Xcode Tools* (or something quite similar).

Double-click the *Xcode Tools* disk icon. A new Finder window will appear, listing the contents of the disk. There will be a series of PDF files that talk about the contents of the package, folders containing the files to be installed, and a master installer file called *Developer.mpkg* (see Figure 2-6).

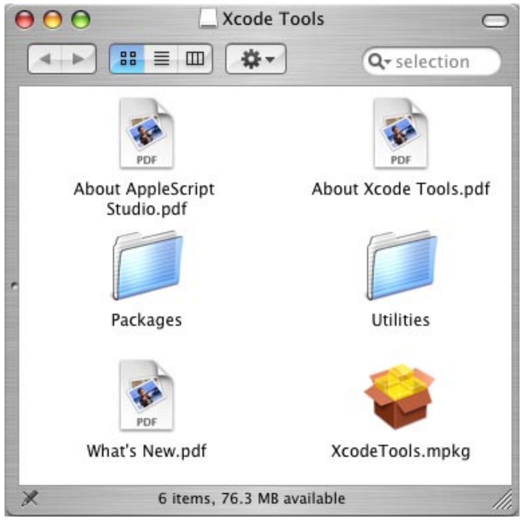


Figure 2.6 *The Xcode installer and associated files. Woohoo!*

This is the moment you've been waiting for! Double-click *Developer.mpkg* and install the tools. As you would with any installer, click *Continue* a few times, then click *Agree* (assuming you agree with the license agreement). Select a hard drive on which to perform the installation (you'll need about 100 Meg of additional space), type in your Admin password when prompted for it, then you are off to the races.

Congratulations! You've just installed the Mac OS X developer tools.

If you run into problems during this process, be sure to head over to http://www.spiderworks.com and check out the support page for this book.

Take Your Tools for a Test Drive

Now that you've installed the tools, let's explore. The first thing to note is the new *Developer* folder at the top level of your hard drive. Go ahead and take a look. It is at the same level as your *Applications* folder. As you make your way down your programming path, you will spend a *lot* of time in the *Developer* folder.

Unix folks have a very efficient system for describing where files live. Files and folders at the top level of your hard drive start with a slash character "/", then follow that with the file or folder name. Thus, we might refer to */Applications* or */Developer*. To dive deeper, add another slash and another file or folder name. For example, inside the *Applications* folder is a *Utilities* subfolder and, inside that, is an application named *Terminal*. Unix folks would refer to the *Terminal* application using this path:

/Applications/Utilities/Terminal

Get the idea?

The tools package you just installed came with its own set of applications. They live inside their own *Applications* folder within the *Developer* folder. Unix folks refer to this folder as */Developer/Applications/*. We'll use this Unix path naming convention throughout the book. It really works well.

In the Finder, navigate into */Developer/Applications/*. Inside that folder, you'll find several subfolders along with two applications, *Interface Builder* and *Xcode*. Interface Builder gives you a powerful set of tools you can use to add the Mac OS X look and feel to your programs. In this book, we'll be focusing on the basics of programming. All of our programs will run in a single scrolling text window. We'll learn what we need to learn to add elements such as windows, menus, scrollbars, buttons, checkboxes and the like in later volumes in this series. For now, you can ignore Interface Builder.

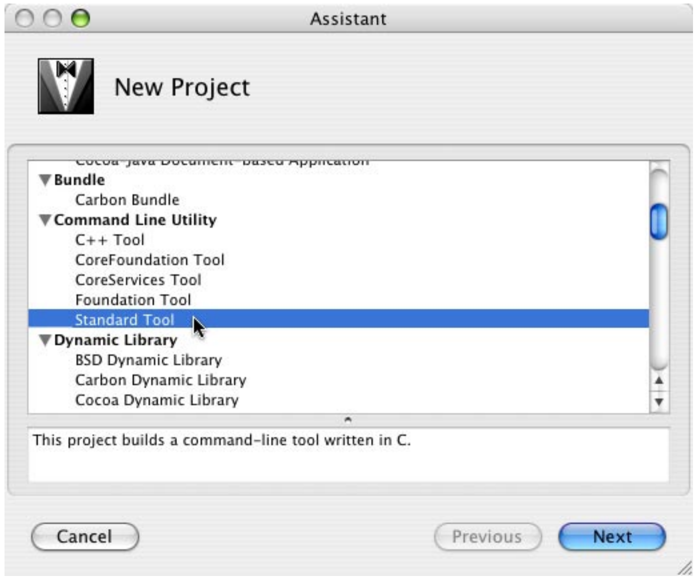
The tool we *will* be focused on in this book is Xcode.

As you'll learn throughout the book, Xcode is a program that helps you organize and build your own programs. If you've never programmed before, don't worry about the specifics. At this point, our goal is to run Xcode, create a test program, and run the test program, just to verify that we have installed Xcode properly.

Double-click on the Xcode icon.

Xcode organizes all the files you use to build a specific program using something called a project file. Depending on your default settings, Xcode may prompt you to open an existing project when it launches. Since we'll be creating a new test project from scratch, click cancel.

To create a new project, select *New Project…* from the *File* menu. Xcode will bring up a new window asking you to select the project type that you want to create. As you can see in Figure 2-7, there are a *lot* of different project types. You can create projects for AppleScript, Java, C++, Objective-C, even projects to build your very own screen saver, to name but a few. And, of course, you can create a project for programming in C.



This project builds a command-line tool written in C.

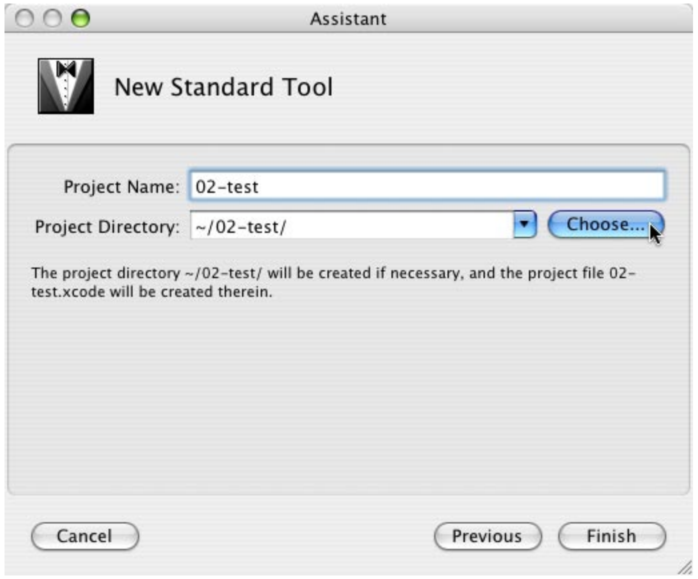
***Figure 2.7** Xcode prompting you to determine what type of new project you want to create.*

To do this, scroll down past the *Application* and *Bundle* sections to the section labeled *Command Line Utility* and select the *Standard Tool* project. Notice (Figure 2-7) that *Standard Tool* is under the *Tool* category. If you can't find the words *Standard Tool*, try clicking the grey triangle to the left of the word *Command Line Utility* to reveal the subcategories below it.

Once you've selected *Standard Tool*, click on the *Next* button on the bottom right to move to the next step.

A new pane appears allowing you to give your project a name and select its location. In the *Project Name* field, type in the name *02-test* (see Figure 2-8). The 02 is for chapter 2 and the test is for, well, test.

Next, we'll fill in the *Project Directory* field. We could edit the text field and type in a Unix path name, but easier still, we can click the *Choose…* button to select a folder. Go ahead and click the *Choose…* button.



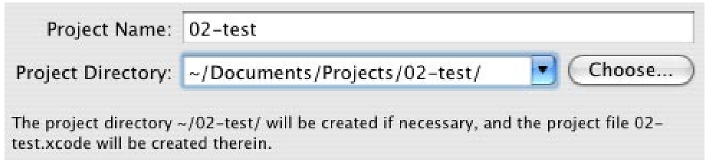
Project Name: 02-test
Project Directory: ~/02-test/

The project directory ~/02-test/ will be created if necessary, and the project file 02-test.xcode will be created therein.

Figure 2.8 *Where to save your new project? Click the Choose... button to select a destination.*

When you click the *Choose...* button, you will be prompted to select a folder in which to store your projects. Start by navigating to your *Documents* folder, then click the *New Folder* button (lower left corner) and create a new folder named *Projects*. Once the Projects folder is created, select it and click the *Choose* button.

Now the *Project Directory* field should look like the one shown in Figure 2-9. We've asked Xcode to create a new C project in a folder name *02-test* within our newly created *Projects* folder.



Project Name: 02-test
Project Directory: ~/Documents/Projects/02-test/

The project directory ~/02-test/ will be created if necessary, and the project file 02-test.xcode will be created therein.

Figure 2.9 *We'll create all our projects in our new* Projects *folder.*

Hmmm. What's that weird character at the beginning of the *Project Directory* field? As it turns out, in the Unix world the tilde character stands for your home directory, the directory assigned to you when your Mac OS X account was created on your machine.

For example, on my machine my login name is davemark and my home directory is */Users/davemark/*. So on my machine, the tilde stands for */Users/davemark/*. On my machine, the path *~/Documents/Projects/02-test/* is shorthand for */Users/davemark/Documents/Projects/02-test/*.

It's also worth noting that in Unix, directory names are traditionally ended with a slash, while file names are not. So *Documents/MyDirectory/* should end with a slash, while *Documents/MyDirectory/myfile* should not end with a slash.

Make sense?

Now that you've specified a project name and directory, click the *Finish* button to create your new project.

Running the Project

When you click the *Finish* button, Xcode will create a project window you'll use to manage your new project. The project window (Figure 2-10) is jam-packed with all sorts of buttons, controls, and text. *Don't* worry about all that stuff. Over time, you'll become quite comfortable with everything you see. For now, all you need to know is that this project window shows you that Xcode is installed properly.

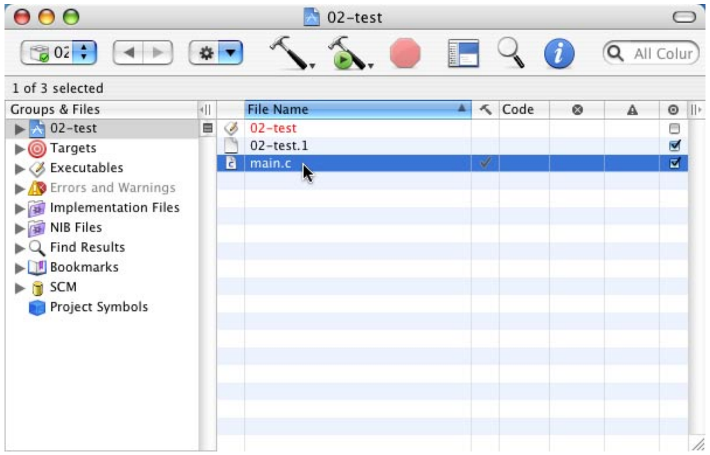


Figure 2.10 *Your new project window.*

Our test program puts up a window, displays the text, *Hello, World!*, then exits gracefully. We didn't have to do anything special to the project. When Xcode creates a new C project, that's what comes right out of the box.

Let's give it a try.

Select *Build and Run* from the *Build* menu. Xcode will do a little thinking, a little behind the scenes action, and will put up a window displaying the following text:

```
Hello, World!

02-test has exited with status 0.
```

This is perfect. Exactly what we were looking for.

Notice that instead of showing a picture of the results window, we just listed the text that appeared in the window. Get used to this. We'll use this approach throughout the rest of this book. Every program we write in this book will produce text as its results. We've set the text in a special font so you can tell it apart from the rest of the book text.

Let's Move On

Well, that's about it for this chapter. You've accomplished a lot. You've joined ADC, logged in, downloaded all the pieces that make up the Xcode installer, reassembled the installer, installed Xcode, created a new project, and built and run your very first program. Awesome! I'd say that calls for a nice, cool beverage of your choice and a well deserved round of applause.

Feel free to quit Xcode if you like. We'll fire it up again in the next chapter. See you there!

Chapter 3 Programming Basics

Before we dig into C programming specifics, we'll spend a few minutes reviewing the basics of programming. We'll answer such questions as "Why write a computer program?" and "How do computer programs work?" We'll look at all of the elements that come together to create a computer program, such as source code, a compiler, and the computer itself.

If you've already done some programming, skim through this chapter. If you feel comfortable with the material, skip ahead to Chapter 4. Most of the issues covered in this chapter will be C-independent.

Programming

Why write a computer program? There are many reasons. Some programs are written in direct response to a problem too complex to solve by hand. For example, you might write a program to calculate the constant π to 5,000 decimal places, or to determine the precise moment to fire the boosters that will safely land the Mars Rover.

Other programs are written as performance aids, allowing you to perform a regular task more efficiently. You might write a program to help you balance your checkbook, keep track of your baseball card collection, or lay out this month's issue of *Dinosaur Today*.

Whatever their purpose, each of these examples shares a common theme. They are all examples of the art of programming. Your goal in reading this book is to learn how to use the C programming language to create programs of your own. Before we get into C, however, let's take a minute to look at some other ways to solve your programming problems.

Some Alternatives to C

As mentioned in Chapter 1, C is one of the most popular programming languages around. There's very little you can't do in C (or in some variant of C), once you know how. On the other hand, a C program is not necessarily the best solution to every programming problem.

For example, suppose you are trying to build a database to track your company's inventory. Rather than writing a custom C program to solve your problem, you might be able to use an off-the-shelf package like FileMaker Pro or, perhaps, a Unix-based solution like MySQL or PostgreSQL to construct your database. The programmers who created these packages solved most of the knotty database management problems you'd face if you tried to write your program from scratch. The lesson here: Before you tackle a programming problem, examine all the alternatives. You might find one that will save you time, money, or that will prove to be a better solution to your problem.

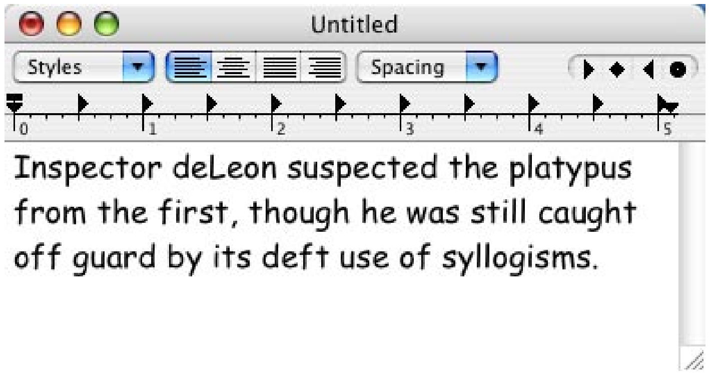
Some problems can be solved using the Mac's built-in scripting language, AppleScript. Just like C, AppleScript is a programming language. Typically, you'd use AppleScript to control other applications. For example, you might create an AppleScript that gets your daily calendar from iCal, formats it just the way you like it using TextEdit, then prints out the results. Or, perhaps, you might write a script that launches Safari and opens each of your bookmarked news sites, each in a separate window. If you can use existing applications to do what you need, chances are good you can use AppleScript to get the job done.

Want to mess with AppleScript? Everything you need to do just that should already be on your hard drive. Look in your *Applications* folder for an *AppleScript* subfolder. Inside the *AppleScript* subfolder, you'll find an application named *Script Editor*. Script Editor lets you create and run AppleScript scripts.

To try your hand at scripting, launch *TextEdit* (it's in the *Applications* folder) and type a few lines of text into the text editing window that appears (see Figure 3-1). Next, launch Script Editor, type in this script and press the **Run** button:

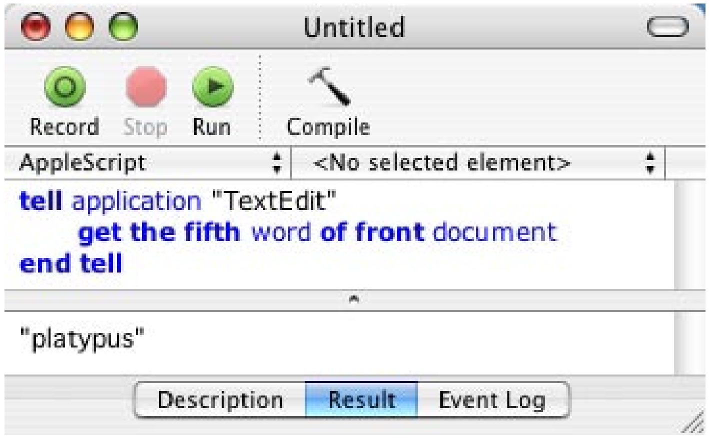
```
tell application "TextEdit"

      get the fifth word of front
document

end tell
```

If all goes well, the fifth word from the TextEdit window should appear in the results pane at the bottom of the Script Editor window (see Figure 3-2). If you are interested in learning more, check out the brand new edition of Danny Goodman's *AppleScript Handbook*, updated for Mac OS X. You'll find it on the http://www.spiderworks.com web site.



Inspector deLeon suspected the platypus from the first, though he was still caught off guard by its deft use of syllogisms.

Figure 3.1 First, I opened TextEdit and typed in a few lines of text...



```
tell application "TextEdit"
    get the fifth word of front document
end tell
```

```
"platypus"
```

Figure 3.2 Next, I typed this script in to Script Editor and clicked the Run *button. The result is shown at the bottom of the window.*

Some applications feature their own proprietary scripting language. For instance, Microsoft Excel lets you write programs that operate on the cells within a spreadsheet. Some word processing programs let you write scripts that control just about every word processing feature in existence. Though proprietary scripting languages can be quite useful, they aren't much help outside their intended environments. You wouldn't find much use for the Excel scripting language outside Excel, for example.

What About C++, Java, and C#?

A while back, there was a big debate in the programming community as to which programming language to learn first. Naturally, the C++ people thought that C++ was by far the best language to start with. Java and C# people felt the same way about Java and C#. But the truth is, each of those languages is based on C. And if you learn C first, you'll have a huge leg up on learning any of these languages. And when the next C-based languages hit the streets (there are several in the works), you'll have a leg up on them, as well.

Learn C first, and all that C knowledge will count towards your C++, Java, and C# education.

The Programming Process

In Chapter 2, you installed the Macintosh developer tools and went through the process of creating a project, then building and running the project. Let's take a look at the programming process in a bit more detail.

Source Code

No matter their purpose, most computer programs start as **source code**. Your source code will consist of a sequence of instructions that tells the computer what to do. Source code is written in a specific programming language, such as C. Each programming language has a specific set of rules that defines what is and isn't "legal" in that language.

Your mission in reading this book is to learn how to create useful, efficient, and, best of all, legal C source code.

If you were programming using everyday English, your source code might look like this:

```
Hi, Computer!
Do me a favor. Ask me for five numbers, add
 them together, then tell me the sum.
```

If you wanted to run this program, you'd need a programming tool that understood source code written in English. Since CodeWarrior doesn't understand English, but does understand C, let's look at a C program that does the same thing:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int index, num, sum;

 sum = 0;

 for ( index=1; index<=5; index++ )
 {
    printf( "Enter number %d --->", index );
    scanf( "%d", &num );
    sum = sum + num;
 }

 printf( "The sum of these numbers is %d.",
 sum );

 return 0;
}
```

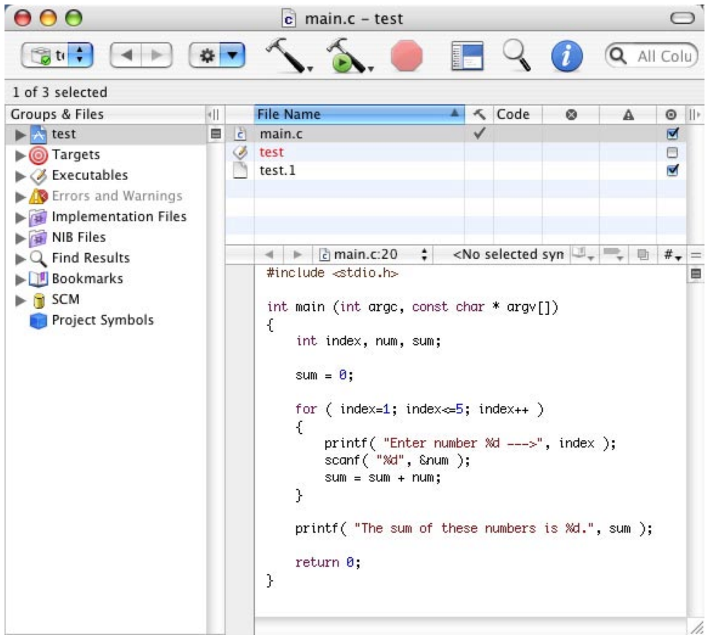
If this program doesn't mean anything to you, don't panic. Just keep reading. By the time you finish reading this book, you'll be writing C code like a pro.

Compiling Your Source Code

Once your source code is written, your next job is to hand it off to a **compiler**. The compiler translates your C source code into instructions that make sense to your computer. These instructions are known as **machine language** or **object code**. Source code is for you, machine language/object code is for your computer. You write the source code using an editor, then the compiler translates your source code into machine readable form.

Don't let the terminology bog you down. And that's an order! Read the rest of this chapter, just to get a basic idea of the programming process, then move on to Chapter 4. I'll lay everything out for you, step-by-step, so you won't get lost.

Xcode collects everything needed to build your project into a **project file**. Figure 3-3 shows a project I built to run the source code above. Again, don't worry about all the details. There's a lot here to absorb. For now, think of the project file as a file folder filled up with all your important papers. But instead of papers, the project file is a collection of all the files that come together to make your project work.



```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int index, num, sum;

    sum = 0;

    for ( index=1; index<=5; index++ )
    {
        printf( "Enter number %d --->", index );
        scanf( "%d", &num );
        sum = sum + num;
    }

    printf( "The sum of these numbers is %d.", sum );

    return 0;
}
```

Figure 3.3 *An Xcode project window, showing some source code.*

Think of the process of running your program as a three stage process. First, Xcode compiles all your source code into object code. Next, all the object code in your project is **linked** together by a program called a **linker** to form your application. That linked application is what actually runs on your computer.

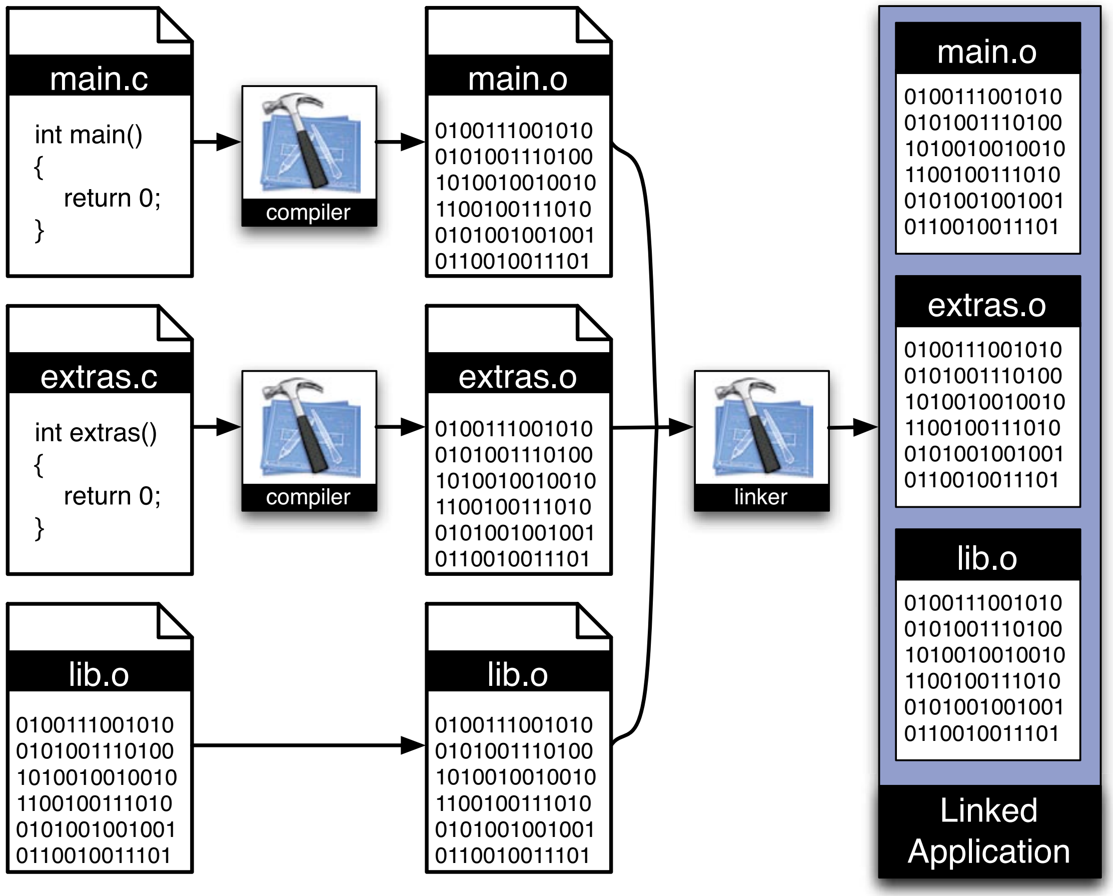


Figure 3.4 *Building your application. First, your source code is compiled, then your object code is linked. The linked application is ready to run.*

Take a look at Figure 3-4. This project contains two source code files, one named *main.c* and another named *extras.c*, as well as an object file named *lib. o*. Sometimes, you'll find yourself making use of some code that someone already compiled. Perhaps they want to share their code, but do not want to show you their source code. Or, perhaps, you built a **library** of code that you'll use again and again and don't want to recompile each time you use the code. By **precompiling** the file into object code and adding the object code into your project, you can save some time.

As you can see in Figure 3-4, Xcode starts by compiling *main.c* and *extras.c* into object code. Next, all three object files are linked together by the linker into a runnable application. Once this is done, Xcode can run your application for you.

What's Next

At this point, don't worry too much about the details. The basic concept to remember from this chapter is that your C programs will start life as source code, then get converted to object code by the compiler. Finally, all the object code gets linked together to form your runnable application.

Ready to get into some source code? Get out your programming gloves - we're about to go to code!

Chapter 4 C Basics: Functions

Every programming language is designed to follow strict rules that define the language's source code structure. The C programming language is no different. These next few chapters will explore the syntax of C.

Chapter 3 discussed some fundamental programming topics, including the process of translating source code into machine code through a tool called the compiler. This chapter focuses on one of the primary building blocks of C programming, the **function**.

C Functions

C programs are made up of functions. A function is a chunk of source code that accomplishes a specific task. You might write a function that adds together a list of numbers, or one that calculates the radius of a given circle. Here's an example:

```
int SayHello( void )
{
 printf( "Hello!!!\n" );
}
```

This function, called **`SayHello()`**, prints a message in a special scrolling text window known as the **run window**. On some systems, this window is also known as the **console window** or just plain **console**. Though technically Xcode treats the run window and console as two separate things, we'll use both terms to refer to the output window used by our programs.

Throughout this book, we'll refer to a function by placing a pair of parentheses after its name. This will help distinguish between variable names and function names. For example, the name **`doTask`** refers to a variable (variables are covered in Chapter 5), while **`doTask()`** refers to a function.

The Function Definition

Functions start off with a **function specifier**, in this case:

```
int SayHello( void )
```

A function specifier consists of a **return type**, the function name, and a pair of parentheses wrapped around a **parameter list**. We'll talk about the return type and parameter list later. For now, the important thing is to be able to recognize a function specifier and be able to pick out the function's name from within the specifier.

Following the specifier comes the **body** of the function. The body is always placed between a pair of curly braces: "**{**" and "**}**". These braces are known in programming circles as "left-curly" and "right-curly". Here's the body of **`SayHello()`**:

```
{
 printf( "Hello!!!\n" );
}
```

The body of a function consists of a series of one or more **statement**s, each followed by a semicolon "**;**". If you think of a computer program as a detailed set of instructions for your computer, a statement is one specific instruction. The **`printf()`** featured in the body of **`SayHello()`** is a statement. It instructs the computer to display some text in the console window.

As you make your way through this book, you'll learn C's rules for creating efficient, compilable statements.

Creating efficient statements will make your programs run faster with less chance of error. The more you learn about programming (and the more time you spend at your craft) the more efficient you'll make your code.

Syntax Errors and Algorithms

When you ask the compiler to compile your source code, the compiler does its best to translate your source code into object code. Every so often, the compiler will hit a line of source code that it just doesn't understand. When this happens, the compiler reports the problem to you. It does not complete the compile. The compiler will not let you run your program until every line of source code compiles.

As you learn C, you'll find yourself making two types of mistakes. The simplest type, called a **syntax error**, prevents the program from compiling. The syntax of a language is the set of rules that determines what will and will not be read by the compiler. Many syntax errors are the result of a mistyped letter, or **typo**. Another common syntax error occurs when you forget the semicolon at the end of a statement.

Syntax errors are usually fairly easy to fix. If the compiler doesn't tell you exactly what you need to fix, it will usually tell you where in your code the syntax error occurred and give you enough information to spot and repair the error.

The second type of mistake is a flaw in your program's **algorithm**. An algorithm is the approach used to solve a problem. You use algorithms all the time. For example, here's an algorithm for sorting your mail:

1) Start by taking the mail out of the mailbox.
2) If there's no mail, you're done! Go watch TV.
3) Take a piece of mail out of the pile.
4) If it's junk mail, throw it away, then go back to step 2.
5) If it's a bill, put it with the other bills, then go back to step 2.
6) If it's not a bill and not junk mail, read it, then go back to step 2.

This algorithm completely describes the process of sorting through your mail. Notice that the algorithm works, even if you didn't get any mail. Notice also that the algorithm always ends up at step 2, with the TV on.

Figure 4.1 shows a pictorial representation of the mail-sorting algorithm, commonly known as a **flow chart**. Much as you might use an outline to prepare for writing an essay or term paper, you might use a flow chart to flesh out a program's algorithm before you actually start writing the program. Here's how this works.

This flow chart uses two types of boxes. The rectangular box portrays an action, such as taking mail out of the mailbox or recycling the junk mail. Once you've taken the action, follow the arrow leading out of the rectangle to go on to the next step in the sequence.

Each diamond-shaped box poses a yes/no question. Unlike their rectangular counterparts, diamond shaped boxes have two arrows leading out of them. One shows the path to take if the answer to the question inside the box is yes, the other shows the path to take if the answer is no. Follow the flow chart through, comparing it to the algorithm described above.

In the C world, a well-designed algorithm results in a well-behaved program. On the other hand, a poorly designed algorithm can lead to unpredictable results. Suppose, for example, you wanted to write a program that added three numbers together, printing the sum at the end. If you accidentally printed one of the numbers instead of the sum of the numbers, your program would still compile and run. The result of the program would be in error, however (you printed one of the numbers instead of the sum), because of a flaw in your program's algorithm.

The efficiency of your source code, referred to earlier, is a direct result of good algorithm design. Keep the concept of algorithm in mind as you work your way through the examples in the book.

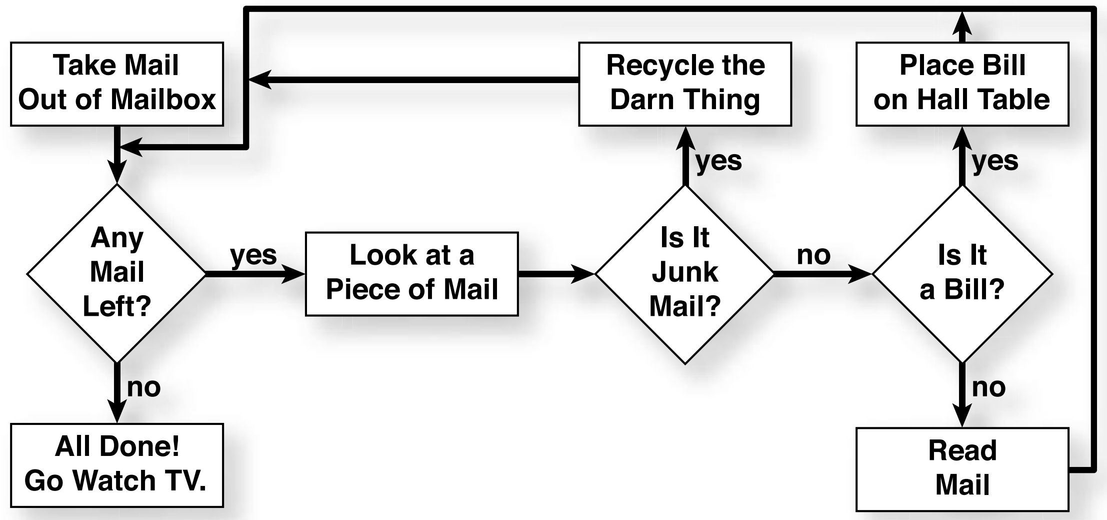


Figure 4.1 *The mail sorting flow chart.*

Calling a Function

In Chapter 2, you ran a test program to make sure Xcode (your programming software) was installed properly. The test program sat in a file called *main.c*, and consisted of a single function, called **`main()`**. As a refresher, here's the source code from *main.c*:

```
#include <stdio.h>

int main (int argc, const char * argv[]) {
    // insert code here...
    printf("Hello, World!\n");
    return 0;
}
```

As you make your way through the code in this book, you'll notice that most of my code follows a slightly different style than Xcode's sample program. I tend to put my open curly brace ("**{**") on its own line, and I tend to sprinkle a few more spaces throughout my code. That's just my personal style. Adopt my style or develop one of your own. Find a style that works for you and be consistent!

At first blush, even this starter program can seem intimidating, but no worries, mate. There's really only one line in this code that you really need to focus on at this point in the book, and that's this function call:

```
    printf("Hello, World!\n");
```

Though this program has lots of complicated looking elements all around, at its heart is a single function call. As far as all the other dangly bits, you can read the tech block that follows for a sneak preview, or just ignore them and know that we'll get to them as we go along.

The source code above can be broken into five basic pieces. Here's the first piece:

```
#include <stdio.h>
```

In C, any line that starts off with a pound sign ("#") is known as a **compiler directive**, an instruction that asks the compiler to do something special. This particular directive is called a `#include` (pronounced "pound include"). It asks the compiler to include code from another file on your hard drive as if that code was in this file in the first place. As it turns out, the file `stdio.h` contains all kids of goodies that we'll use throughout the book. Just ignore this line for now.

Here's the second piece:

```
int main (int argc, const char *
argv[]) {

}
```

As we discussed a bit earlier, this is the function specifier for the function named `main()`. The curly-braces ("`{`" and "`}`") surround the body of the function.

The third piece of this puzzle is this line:

```
// insert code here...
```

Any time the compiler encounters two slashes ("`//`") in a row, it ignores the slashes and anything else on that line. This line of code is called a comment. Its only purpose is to document your code and to help make clear what's going on at this point in the program. Comments are a good thing.

The fourth piece is the call to the function printf(), which we'll focus on in a bit:

```
printf("Hello, World!\n");
```

The fifth and final piece of our program is this line of code:

```
return 0;
```

A return statement in a function tells the compiler that you are done with this function and you want to return. In this case, you want the function to return a value of 0.

Again, don't get hung up on the specifics. It'll all become clear as you go.

So what does "calling a function" really mean? Basically, whenever your source code calls a function, each of the statements in the *called* function is executed before the next statement of the *calling* function is executed.

Confused? Look at Figure 4.2. In this example, `main()` starts with a call to the function `MyFunction()`. This call to `MyFunction()` will cause each of the statements inside `MyFunction()` to be executed. Once the last statement in **`MyFunction()`** is executed, control is returned to **`main()`**. Now, **`main()`** can call **`AnotherFunction()`**.

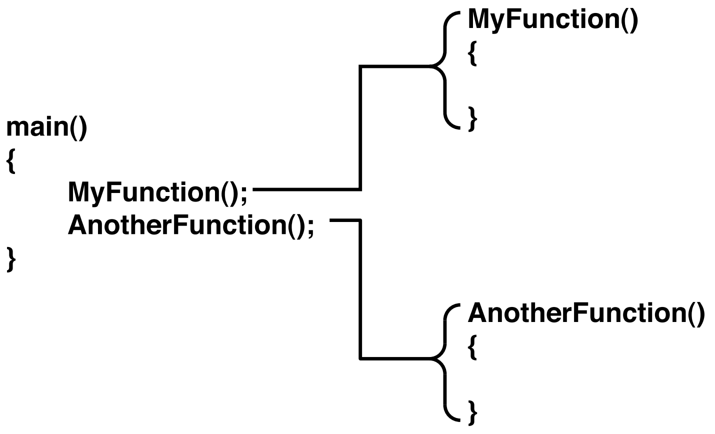


Figure 4.2 *main() calls MyFunction(), then calls AnotherFunction().*

Every C program you write will have a **`main()`** function. Your program will start running with the first line in **`main()`** and, unless something unusual happens, end with the last line in **`main()`**. Along the way **`main()`** may call other functions which may, in turn, call other functions and so on.

ISO C and the Standard Library

The American National Standards Institute (**ANSI**) established a national standard for the C programming language. This standard became known as **ANSI C**. Later, the International Standards Organization (**ISO**) adopted this standard, and ANSI C evolved into the international standard known as **ISO C**. Part of this standard is a specific definition of the syntax of the C language.

Occasionally, you'll still hear C programmers refer to the ANSI C standard. The main difference between the two standards is that ISO C has extra functions in its Standard Library to handle multibyte and wide characters. ISO C, ANSI C, either term is fine. The important thing is to be aware that a strict C standard does exist.

As we stated earlier, the syntax of a language gives programmers a set of rules that rigidly defines the format of their source code. For example, ISO C tells you when you can and can't use a semicolon. ISO C tells you to use a pair of curly braces to surround the body of each function. You get the idea. The greatest benefit to having an international standard for C is portability. With a minimum of tinkering, you can get an ISO C program written on one computer up and running on another computer. When you finish with this book, you'll be able to program in C on any computer that has an ISO C compiler.

Another part of the ISO C standard is the **Standard**

Library. The Standard Library is a set of functions available to every ISO C programmer. As you may have guessed, the **`printf()`** function you've seen in our sample source code is part of the Standard Library.

There are tons of great functions in the Standard Library. You'll learn some of the more popular ones as we make our way through the book. Once you get comfortable with the Standard Library functions presented here, dig through some of the Standard Library documentation that you'll find on the web, just to get a sense of what else is in there.

There are a number of great sites that discuss the Standard Library. One of my favorite resources on the net is Wikipedia (http://www.wikipedia.org), an open-content, collaborative encyclopedia. If you've never played with Wikipedia, here's an excellent link to get you started:

> http://en.wikipedia.org/wiki/ANSI_C_standard_library

Yeah, it's a bit techie, but an invaluable reference resource once you start developing your own code, or if you encounter a function in this book and want to know more.

Another great page (also referenced at the bottom of the Wikipedia page) is the detailed C Standard Library reference maintained by our friends at the University of Tasmania:

> http://www.infosys.utas.edu.au/info/documentation/C/CStdLib.html

Enjoy!

Same Program, Two Functions

As you start writing your own programs, you'll find yourself designing many individual functions. You might need a function that puts a form up on the screen for the user to fill out. You might need a function that takes a list of numbers as input, providing the average of those numbers in return. Whatever your needs, you will definitely be creating a lot of functions. Let's see how it's done.

Our first program contained a function named **`main()`** that passed the text string **`"Hello, world!\n"`** to **`printf()`**. Our next program, **`hello2`**, captures that functionality in a new function, called **`SayHello()`**.

You've probably been wondering why the characters "**`\n`**" keep appearing at the end of all our text strings. Don't worry, there's nothing wrong with your copy of the book. The "**`\n`**" is perfectly normal. It tells **`printf()`** to move the cursor to the beginning of the next line in the text window, sort of like hitting the return key in a text editor.

The sequence "**`\n`**" is frequently referred to as a newline character, a carriage return, or just plain return. By including a return at the end of a **`printf()`**, we know that the next line we print will appear at the beginning of the next line in the text window.

Opening hello2.xcode

In the Finder, open the `Learn C Projects` folder, open the subfolder named `04.01 - hello2` and double-click on the project file `hello2.xcode`. A project window with the title **`hello2`** will appear, as shown in Figure 4.3.

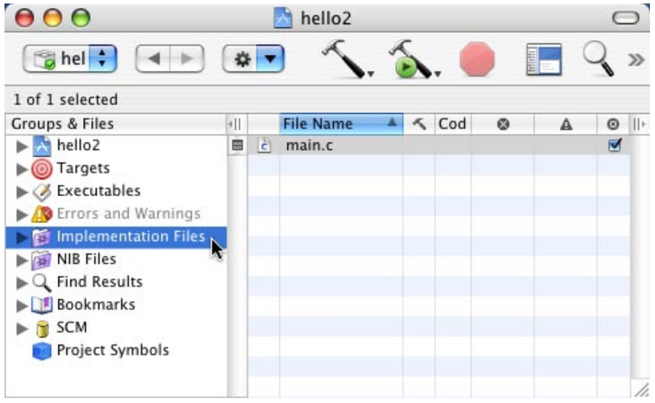
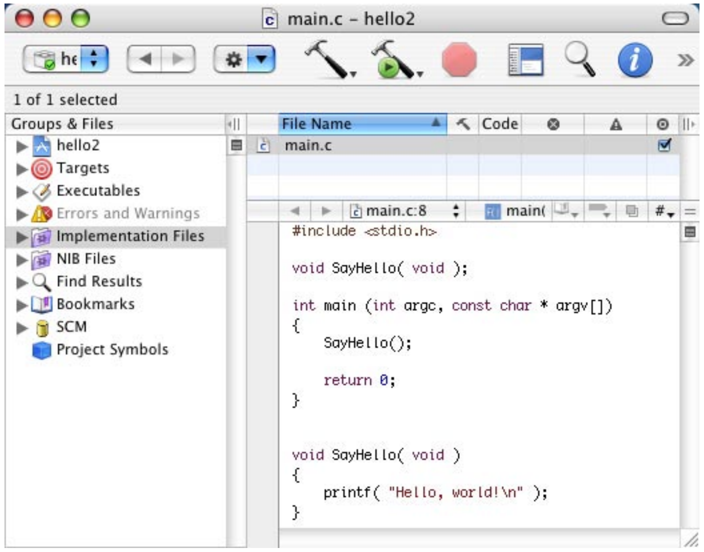


Figure 4.3 *The hello2 project window.*

Notice the left column, labeled *Groups & Files*. This is also known as the **Groups & Files pane** or the **smart groups pane**, a pane being a sub-area of a window. In Figure 4.3, the *Implementation Files* group is selected, and the main area of the project window lists the files in this group. In this case, *main.c* is the only source code file in this project. Later in the book, we'll see projects with multiple source code files. They'll all be listed in the *Implementation Files* group.

If you double-click on the name *main.c* in the main area, a new source code editing window will appear, allowing you to edit your source code. Alternatively, you can edit your source code right inside your project window by clicking on the **Show/Hide Editor icon** in the row of icons at the top of the project window. The icon is just to the right of the stop sign icon and looks like a miniature version of the project window. When you first click the editor, a source code editing pane appears in the project window (see Figure 4.4).



```
#include <stdio.h>

void SayHello( void );

int main (int argc, const char * argv[])
{
    SayHello();

    return 0;
}


void SayHello( void )
{
    printf( "Hello, world!\n" );
}
```

***Figure 4.4** Click on the Show/Hide Editor icon at the top of the window to open a source code editing pane in the project window.*

Here's the source code from *main.c*:

```
#include <stdio.h>

void SayHello( void );

int main (int argc, const char * argv[])
{
 SayHello();

        return 0;
}


void SayHello( void )
{
 printf( "Hello, world!\n" );
}
```

Let's walk through this, line-by-line. **`hello2`** starts off with this line of source code:

```
#include <stdio.h>
```

You'll find this line (or a slight variation) at the beginning of each one of the programs in this book. It tells the compiler to include the source code from the file *stdio.h* as it compiles *main.c*. *stdio.h* contains information we'll need if we are going to call **`printf()`** in this source code file. You'll see the **`#include`** mechanism used throughout this book.

We'll talk about it in detail later in the book. For now, get used to seeing this line of code at the top of each of our source code files.

The line following the **`#include`** is blank. This is completely cool. Since the C compiler ignores all blank lines, you can use them to make your code a little more readable. I like to leave a few blank lines (at least) between each of my functions.

This line of code appears next:

```
void SayHello( void );
```

While this line might look like a function specifier, don't be fooled! If this were a function specifier, it would not end with a semi-colon and it would be followed by a left-curly-brace ("{") and the rest of the function. This line is known as a **function prototype** or **function declaration**. You'll include a function prototype for every function, other than **`main()`**, in your source code file.

To understand why, it helps to know that a compiler reads your source code file from the beginning to the end, a line at a time. By placing a complete list of function prototypes at the beginning of the file, you give the compiler a preview of the functions it is about to compile. The compiler uses this information to make sure that calls to these functions are made correctly.

This will make a lot more sense to you once we get into the subject of parameters later on. For now, get used to seeing function prototypes at the beginning of all your source code files.

Next comes the function **`main()`**. **`main()`** first calls the function **`SayHello()`**:

```
int main (int argc, const char * argv[])
{
 SayHello();
```

At this point, the lines of the function **`SayHello()`** get run. When **`SayHello()`** is finished, **`main()`** can move on to its next line of code. The keyword **`return`** tells the compiler to return from the current function, without executing the remainder of the function. We'll talk about **`return`** later on. Until then, the only place you'll see this line is at the end of **`main()`**.

```
        return 0;
}
```

Following **`main()`** is another pair of blank lines, followed by the function **`SayHello()`**. **`SayHello()`** prints the string "Hello, world!", followed by a return, in a window, then returns control to **`main()`**.

```
void SayHello( void )
{
 printf( "Hello, world!\n" );
}
```

Let's step back for a second and compare our first program to **`hello2`**. In our first program, **`main()`** called **`printf()`** directly. In **`hello2`**, **`main()`** calls a function which then calls **`printf()`**. This extra layer demonstrates a basic C programming technique, taking code from one function and using it to create a new function. This example took this line of code:

```
printf( "Hello, world!\n" );
```

and used it to create a new function called **`SayHello()`**. This function is now available for use by the rest of the program. Every time we call the function **`SayHello()`**, it's as if we executed the line of code:

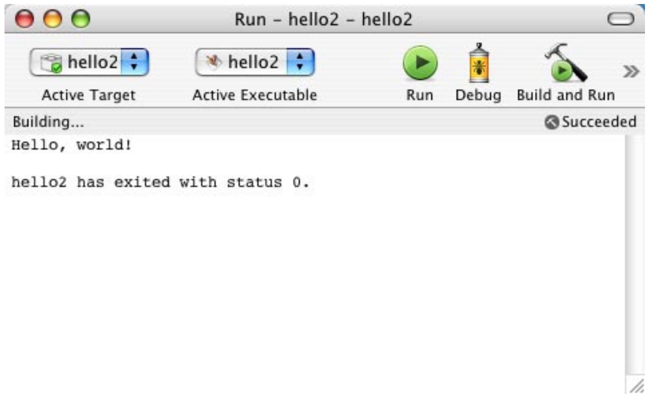
```
printf( "Hello, world!\n" );
```

`SayHello()` may be a simple function, but it demonstrates an important concept. Wrapping a chunk of code in a single function is a powerful technique. Suppose you create an extremely complex function, say, 100 lines of code in length. Now, suppose you call this function in five different places in your program. With 100 lines of code, plus the five function calls, you are essentially achieving 500 lines' worth of functionality. That's a pretty good return on your investment!

Let's watch **`hello2`** in action.

Running hello2

Select Build and Run from the Project menu. You'll see a window similar to the one shown in Figure 4.5. Gee, this looks just like the output from Chapter 2's test program. Of course, that was the point. Even though we embedded our **`printf()`** inside the function **`SayHello()`**, **`hello2`** produced the same results.



```
Hello, world!

hello2 has exited with status 0.
```

Figure 4.5 *The output from hello2.*

Before we move on to our next program, let's revisit a little terminology we first touched on at the beginning of the chapter. The window that appeared when we ran **hello2** is referred to as the run window and, less formally, as the console window or just plain console. There are a number of Standard Library functions designed to send text to the console window. The text that appears in the console window is known as **output**. After you run a program, you're likely to check the output that appears in the console to make sure your program ran correctly.

Text based programming is fine and good, but eventually, you'll want to expand your horizons and learn how to add graphical elements like buttons, scrollbars, windows, and menus to your programs. Have patience, stick with the program, and you'll get there. Start with *Learn C*. Once you feel comfortable with C, move on to Mark Dalrymple's excellent *Learn Objective C*, which will teach you how to add object programming to your C code. Once you have a handle on Objective C, you'll be ready to add Cocoa to the mix. Learn C to learn the basics of programming, add objects to the mix with Objective C, then bring the Mac-specific user interface widgets to life with Cocoa.

Again, have patience, stick with the program, and be sure to send me pictures of you, on the beach in Hawaii, celebrating the release of your brand new, best selling application!

Another Example

Imagine what would happen if you changed **hello2**'s version of **main()** so that it read:

```
int main (int argc, const char * argv[])
{
 SayHello();
 SayHello();
 SayHello();

 return 0;
}
```

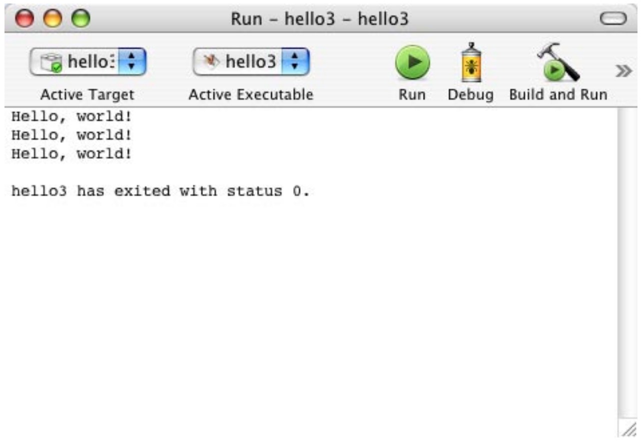
What's different? In this version, we've added two more calls to **SayHello()**. Can you picture what the console will look like after we run this new version?

To find out, close the **hello2** project window, then select Open… from Xcode's File menu. Note that as soon as you close the project window, Xcode will close all the other project-related windows automatically.

When Xcode prompts you to open a project, navigate into the *Learn C Projects* folder, then into the *04.02 – hello3* subdirectory and open the *hello3.xcode* project file.

When you run **hello3**, the run window shown in Figure 4.6 will appear. Take a look at the output. Does it make sense to you? Each call to **SayHello()** generates the text “Hello, world!”

followed by a carriage return.



```
Hello, world!
Hello, world!
Hello, world!

hello3 has exited with status 0.
```

Figure 4.6 *The output from hello3.*

Generating Some Errors

Before we move on to the next chapter, let's see how the compiler responds to errors in our source code. In the **`hello3`** project window, use your favorite method to edit the *main.c* source code file. Remember, you can click the *Show Editor* icon to edit the source file in the project window itself, or you can locate *main.c* in the *Implementation Files* group and double-click the name to open a new *main.c* editing window. Either method is fine.

In the source code window, find the line of source code containing the function specifier for **`main()`**. The line should read:

```
int main (int argc, const char * argv[])
```

Click at the end of the line, so the blinking cursor appears at the right end of the line. Now type a semicolon, so that the line now reads:

```
int main (int argc, const char * argv[]);
```

Here's the entire file, showing the tiny change you just made:

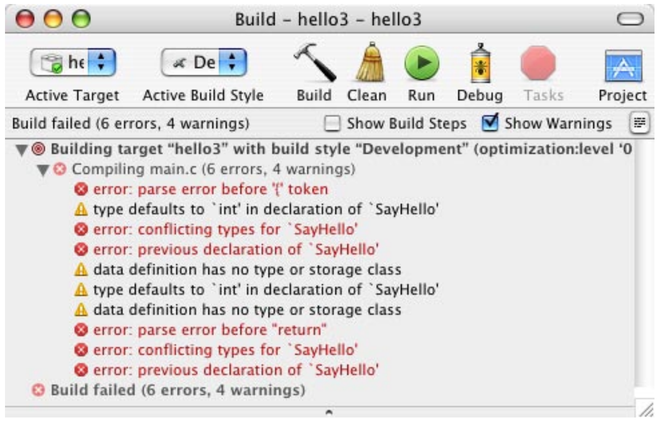
```
#include <stdio.h>

void SayHello( void );
```

```
int main (int argc, const char * argv[]);
{
 SayHello();
 SayHello();
 SayHello();

 return 0;
}


void SayHello( void )
{
 printf( "Hello, world!\n" );
}
```

Keep in mind that you only added a single semicolon to the source code and select Build and Run from the Build menu. Xcode knows that you changed your source code since the last time it was compiled and it will try to recompile *main.c*. Figure 4.7 shows the error window that appears, telling you that you've got a problem with your source code. Yikes! All that, just because you added a measly semicolon!

Sometimes, the compiler will give you a perfectly precise message that exactly describes the error it encountered. In this case, however, the compiler got so confused by the extra semicolon, it reported 6 errors instead of just one. Notice, however, that the very first error message gives you a pretty good idea of what is going on. It complains about a parse error before the "**{**" token. The compiler is reading your source code, making its way down *main.c*, when it encounters what it thinks is a function specifier. But then, just when it expects an open curly brace, it finds a semicolon. Hrm. That's not right. Better report an error.

In the build window, if you double-click on the first error line (the line that says "error: parse error before { token"), Xcode will take you to the offending line in the *main.c* editing window. In general, when you encounter an error compiling your code, you'll double-click on the error message, figure out what's wrong, fix it, then move on to the next error. Sometimes, I fix one error and immediately recompile, just on the off-chance that this one error actually was the cause of all the other error message, as is the case with our errant semicolon.



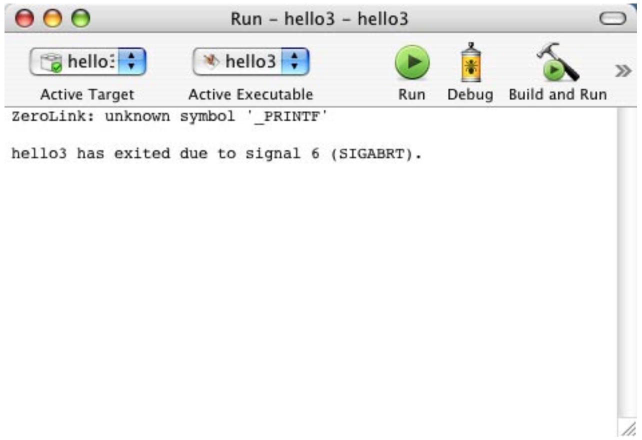
```
Build failed (6 errors, 4 warnings)
▼ Building target "hello3" with build style "Development" (optimization:level '0
  ▼ Compiling main.c (6 errors, 4 warnings)
      error: parse error before '{' token
      type defaults to `int' in declaration of `SayHello'
      error: conflicting types for `SayHello'
      error: previous declaration of `SayHello'
      data definition has no type or storage class
      type defaults to `int' in declaration of `SayHello'
      data definition has no type or storage class
      error: parse error before "return"
      error: conflicting types for `SayHello'
      error: previous declaration of `SayHello'
  Build failed (6 errors, 4 warnings)
```

***Figure 4.7** What? All these errors just from adding a simple semicolon? Yup.*

Go back to your *main.c* editing window, delete the extra semicolon, then select Build and Run from the Build menu. Xcode will recompile your code and rerun the program, proving that you have indeed fixed the error. Good.

C is Case Sensitive

There are many different types of errors possible in C programming. One of the most common results from the fact that C is a **case-sensitive** language. In a case-sensitive language, there is a big difference between lower- and upper-case letters. This means you can't refer to `printf()` as `Printf()` or even `PRINTF()`. Figure 4.8 shows the warning message you'll get if you change your call of `printf()` to `PRINTF()`. Basically, this message is telling you that Xcode couldn't find a function named `PRINTF()` and will do its best to run the program anyway, assuming it will find the appropriate function at run-time (when the program runs). To fix this problem, just change `PRINTF()` back to `printf()` and recompile.



```
ZeroLink: unknown symbol '_PRINTF'

hello3 has exited due to signal 6 (SIGABRT).
```

Figure 4.8 The warning message you get when you change **`printf()`** *to* **`PRINTF()`**.

What's Next?

Congratulations! You've made it through basic training. You know how to open a project, how to compile your code, and even how to create an error message or two. You've learned about the most important function: **`main()`**. You've also learned about **`printf()`** and the Standard Library.

Now you're ready to dig into the stuff that gives a C program life: variables and operators.

Exercises

Open the project *hello2.xcode*, edit *hello2.c* as described in each exercise, and describe the error that results:

1) Change the line:

```
SayHello();
```

to say:

```
SayHello(;
```

2) Change things back. Now change the line:

```
int main (int argc, const char * argv[])
```

to say:

```
int MAIN (int argc, const char * argv[])
```

3) Change things back. Now delete the "**{**" after the line:

```
int main (int argc, const char * argv[])
```

4) Change things back. Now delete the semicolon at the end of this line:

```
printf( "Hello, world!\n" );
```

so it reads:

```
printf( "Hello, world!\n" )
```

Chapter 5 C Basics: Variables and Operators

At this point, you should feel pretty comfortable using Xcode. You should know how to open a project and how to edit a project's source code. You should also feel comfortable running a project and (heaven forbid) fixing any syntax errors that may have occurred along the way.

On the programming side, you should recognize a function when you see one. When you think of a function you should first think of `main()`, the most important function. You should remember that functions are made up of statements, each of which is followed by a semicolon.

With these things in mind, we're ready to explore the foundation of C programming: **variables** and **operators**. Variables and operators are the building blocks you'll use to construct your program's statements.

An Introduction to Variables

A large part of the programming process involves working with data. You might need to add together a column of numbers or sort a list of names alphabetically. The tricky part of this process is representing your data in a program. This is where variables come in.

Variables can be thought of as containers for your program's data. Imagine a table with three containers sitting on it. Each container is labeled. One container is labeled **`cup1`**, one labeled **`cup2`**, and the third **`cup3`**. Now imagine you have three pieces of paper. Write a number on each piece of paper and place one piece inside each of the three containers. Figure 5.1 shows a picture of what this might look like.

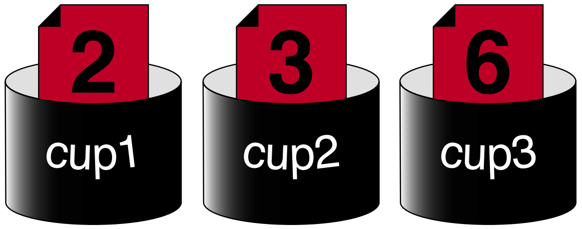


Figure 5.1 Three cups, each with its own value.

Now imagine asking a friend to reach into the three cups, pull out the number in each one, and add the three values together. You can ask your friend to place the sum of the three values in a fourth container created just for this purpose. The fourth container is labeled **`sum`** and can be seen in Figure 5.2.

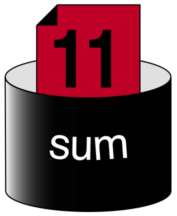


Figure 5.2 A fourth container, containing the sum of the other three containers.

This is exactly how variables work. Variables are containers for your program's data. You create a variable and place a value in it. You then ask the computer to do something with the value in your variable. You can ask the computer to add three variables together, placing the result in a fourth variable. You can even ask the computer to take the value in a variable, multiply it by 2, and place the result back into the original variable.

Getting back to our example, now imagine that you changed the values in **`cup1`**, **`cup2`**, and **`cup3`**. Once again, you could call on your friend to add the three values, updating the value in the container sum. You've reused the same variables, using the same formula, to achieve a different result. Here's the C version of this formula:

```
sum = cup1 + cup2 + cup3;
```

Every time you execute this line of source code, you place the sum of the variables **`cup1`**, **`cup2`**, and **`cup3`** into the variable named **`sum`**. At this point, it's not important to understand exactly how this line of C source code works. What *is* important is to understand the basic idea behind variables. Each variable in your program is like a container with a value in it. This chapter will teach you how to create your own variables and how to place a value in a variable.

Working With Variables

Variables come in a variety of flavors, called **types**. A variable's type determines the type of data that can be stored in that variable. You determine a variable's type when you create the variable. (We'll discuss creating variables in just a second.) Some variable types are useful for working with numbers. Other variable types are designed to work with text. In this chapter, we'll work strictly with variables of one type, a numerical type called **`int`** (eventually, we'll get into other variable types). A variable of type **`int`** can hold a numerical value, such as 27 or -589.

Working with variables is a two-stage process. First you *create* a variable, then you *use* the variable. In C, you create a variable by **declaring** it. Declaring a variable tells the compiler, "Create a variable for me. I need a container to place a piece of data in." When you declare a variable, you have to specify the variable's type as well as its name. In our earlier example, we created four containers. Each container had a label. In the C world, this would be the same as creating four variables with the names **`cup1`**, **`cup2`**, **`cup3`**, and **`sum`**. In C, if we want to use the value stored in a variable, we use the variable's name. We'll show you how to do this later in the chapter.

Here's an example of a variable declaration:

```
int  myVariable;
```

This declaration tells the compiler to create a variable of type **`int`** (remember, **`int`**s are useful for working with numbers) with the name **`myVariable`**. The type of the variable (in this case, **`int`**) is extremely important. As you'll see, variable type determines the type and range of values a variable can be assigned.

Variable Names

Here are a few rules to follow when you create your own variable names:

- Variable names must always start with an upper or lower-case letter (A, B, ..., Z or a, b, ..., z) or with an underscore ("_").
- The remainder of the variable name must be made up of upper or lower-case letters, numbers (0, 1, ..., 9), or the underscore.

These two rules yield variable names like **`myVariable`**, **`THIS_NUMBER`**, **`VaRiAbLe_1`**, and **`A1234_4321`**. Note that a C variable may never include a space, or a character like "**`&`**" or "**`*`**". These two rules *must* be followed.

On the other hand, these rules do leave a fair amount of room for inventiveness. Over the years, different groups of programmers came up with additional guidelines (also known as **conventions**) that made variable names more consistent and a bit easier to read.

As an example of this, Unix programmers tended to use all lower case letters in their variable names. When a variable name consisted of more than one word, the words were separated by an underscore.

This yielded variable names like **`my_variable`** or **`number_of_puppies`**.

Another popular convention stems from a programming language named SmallTalk. Instead of limiting all variable names to lower case and separating words with an underscore ("_"), SmallTalk used a convention known as **InterCap**, where all the words in a variable or function name are stuck together. Rather than include a special, separating character, each new word added to the first word starts with a capital letter. For example, instead of **`number_of_puppies`**, you'd use **`numberOfPuppies`**. Instead of **`my_variable`**, you'd use **`myVariable`**. Function names follow the same convention, but start with a capital letter, giving us function names such as **`SmellTheFlowers()`** or **`HowMuchChangeYouGot()`**.

Which convention should you use? For now, we'll follow the InterCap SmallTalk convention described in the previous paragraph. But as you make your way through the programming universe, you'll encounter different naming conventions that vary with each programming environment you encounter.

As mentioned in Chapter 4, C is a case-sensitive language. The compiler will cough out an error if you sometimes refer to **`myVariable`** and other times refer to **`myvariable`**. Adopt a variable naming convention and stick with it - Be consistent!

The Size of a Type

When you declare a variable, the compiler reserves a section of memory for the exclusive use of that variable. When you assign a value to a variable, you are actually modifying the variable's dedicated memory to reflect that value. The number of bytes assigned to a variable is determined by the variable's type. You should check your compiler's documentation to see how many bytes go along with each of the standard C types.

Some Macintosh compilers assign 2 bytes to each **`int`**. Others assign 4 bytes to each **`int`**. By default, Xcode uses 4 byte **`int`**s.

It's important to understand that the size of a type can change, depending on factors such as your computer's processor type, operating system (Mac OS X vs. Windows, for example), and your development environment. Remember, read the documentation that comes with your compiler.

Let's continue with the assumption that Xcode is using 4 byte ints. The variable declaration:

```
int  myInt;
```

reserves memory (in our case, 4 bytes) for the exclusive use of the variable **`myInt`**. If you later assign a value to **`myInt`**, that value is stored in the 4 bytes allocated for **`myInt`**. If you ever refer to `myInt`'s value, you'll be referring to the value stored in `myInt`'s 4 bytes.

If your compiler used 2 byte `int`s, the preceding declaration would allocate 2 bytes of memory for the exclusive use of `myInt`. As you'll see, it is important to know the size of the types you are dealing with.

Why is the size of a type important? The size of a type determines the range of values that type can handle. As you might expect, a type that's 4 bytes in size can hold a wider range of values than a type that's only 1 byte in size. Here's how all this works…

Bytes and Bits

Each byte of computer memory is made up of 8 **bits**. Each bit has a value of either 1 or 0. Figure 5.3 shows a byte holding the value 00101011. The value 00101011 is said to be the **binary** representation of the value of the byte. Look closer at Figure 5.3. Notice that each bit is numbered (the bit numbers are above each bit in the figure), with bit 0 on the extreme right side to bit 7 on the extreme left. This is a standard bit-numbering scheme used in most computers.

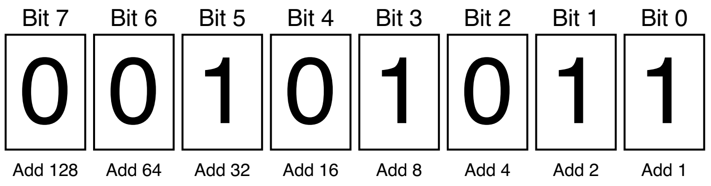


Figure 5.3 *A byte holding the binary value 00101011.*

Notice also the labels that appear beneath each bit in the figure ("Add 1", "Add 2", etc.). These labels are the key to binary numbers. Memorize them. (It's easy — each bit is worth twice the value of its right neighbor.) These labels are used to calculate the value of the entire byte. Here's how it works:

- Start with a value of 0.
- For each bit with a value of 1, add the label value below the bit.

That's all there is to it! In the byte pictured in Figure 5.3, you'd calculate the byte's value by adding 1 + 2 + 8 + 32 = 43. Where did we get the 1, 2, 8, and 32? They're the bottom labels of the only bits with a value of 1. Try another one.

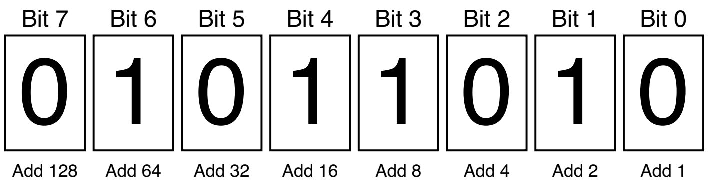


Figure 5.4 *What's the value of this byte?*

What's the value of the byte pictured in Figure 5.4? Easy, right? 2 + 8 + 16 + 64 = 90. Right! How about the byte in Figure 5.5?

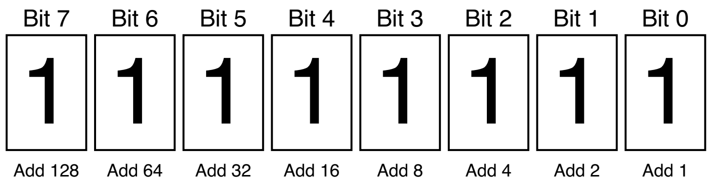


Figure 5.5 *Last one: What's the value of this byte?*

This is an interesting one: 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255. This example demonstrates the largest value that can fit in a single byte. Why? Because every bit is turned on. We've added everything we can add to the value of the byte.

The smallest value a byte can have is 0 (00000000). Since a byte can range in value from 0 to 255, a byte can have 256 possible values.

Actually, this is just one of several ways to represent a number using binary. This approach is fine if you want to represent integers that are always greater than or equal to 0 (known as **unsigned** integers). Computers use a different technique, known as **two's complement notation**, when they want to represent integers that might be either negative or positive.

To represent a negative number using two's complement notation:

- Start with the binary representation of the positive version of the number
- Complement all the bits (turn the 1s into 0s and the 0s into 1s)
- Add 1 to the result.

For example, the binary notation for the number 9 is 00001001. To represent -9 in two's complement notation, flip the bits (11110110) then add 1. The two's complement for -9 is 11110110 + 1 = 11110111.

The binary notation for the number 2 is 00000010. The two's complement for -2 would be 11111101 + 1 = 11111110. Notice that in binary addition, when you add 01 + 01 you get 10. Just as in regular addition, you carry the 1 to the next column.

Don't worry about the details of binary representation and arithmetic. What's important to remember is that the computer uses one notation for positive-only numbers and a different notation for numbers that can be positive or negative. Both notations allow a byte to take on one of 256 different values. The positives-only scheme allows values ranging from 0 to 255. The two's complement scheme allows a byte to take on values ranging from -128 to 127. Note that both of these ranges contain exactly 256 values.

Going From 1 to 2 Bytes

So far, we've discovered that 1 byte (8 bits) of memory can hold one of $2^8 = 256$ possible values. By extension, 2 bytes (16 bits) of memory can hold one of $2^{16} = 65{,}536$ possible values. If the 2 bytes are **unsigned** (never allowed to hold a negative value) they can hold values ranging from 0 to 65,535. If the 2 bytes are **signed** (allowed to hold both positive and negative values) they can hold values ranging from -32,768 to 32,767.

A 4 byte **`int`** can hold $2^{32} = 4{,}294{,}967{,}296$ possible values. Wow! An unsigned 4 byte int can hold values ranging from –2,147,483,648 to 2,147,483,647, while a signed 4 byte int can hold values from 0 to 4,294,967,295.

To declare a variable as **`unsigned`**, precede its declaration with the **`unsigned`** qualifier. Here's an example:

```
unsigned int       myInt;
```

Now that you've defined the type of variable your program will use (in this case, **`int`**), you can assign a value to your variable.

Operators

One way to assign a value to a variable is with the **=** **operator**, also known as the **assignment operator**. An **operator** is a special character (or set of characters) that represents a specific computer operation. The assignment operator tells the computer to compute the value of the right side of the **=** and assign that value to the left side of the **=**. Take a look at this line of source code:

```
myInt = 237;
```

This statement causes the value 237 to be placed in the memory allocated for **`myInt`**. In this line of code, **`myInt`** is known as an **l-value** (stands for left-value), because it appears on the *left* side of the **=** operator. A variable makes a fine l-value. A number (like 237) makes a terrible l-value. Why? Because values are copied *from the right* side *to the left* side of the **=** operator. In this line of code:

```
237 = myInt;
```

you are asking the compiler to copy the value in **`myInt`** to the number 237. Since you can't change the value of a number, the compiler will report an error when it encounters this line of code (most likely, the error message will say something about an "invalid lvalue" – go ahead, try this yourself).

As we just illustrated, you can use numerical **constants** (such as 237) directly in your code. In the programming world, these constants are called **literals**. Just as there are different types of variables, there are also different types of literals. You'll see more on this topic later in the book.

Look at this example:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int myInt, anotherInt;

 myInt = 503;
 anotherInt = myInt;

 return 0;
}
```

Notice we've declared two variables in this program. One way to declare multiple variables is the way we did here, separating the variables by a comma ("`,`"). There's no limit to the number of variables you can declare using this method.

We could have declared these variables using two separate declaration lines:

```
int  myInt;
int  anotherInt;
```

Either way is fine. As you'll see, C is an extremely flexible language. Let's look at some other operators.

The +, -, ++, and -- Operators

The **+** and **-** operators each take two values and reduce them to a single value. For example, the statement:

```
myInt = 5 + 3;
```

will first resolve the right side of the **=** by adding the numbers 5 and 3 together. Once that's done, the resulting value (8) is assigned to the variable on the left side of the **=**. This statement assigns the value 8 to the variable **`myInt`**. Assigning a value to a variable means copying the value into the memory allocated to that variable.

Here's another example:

```
myInt = 10;
anotherInt = 12 - myInt;
```

The first statement assigns the value 10 to **`myInt`**. The second statement subtracts 10 from 12 to get 2, then assigns the value 2 to **`anotherInt`**.

The **`++`** and **`--`** operators operate on a single value only. **`++` increments** (raises) the value by 1 and **`--` decrements** (lowers) the value by 1. Take a look:

```
myInt = 10;
myInt++;
```

The first statement assigns **`myInt`** a value of 10. The second statement changes **`myInt`**'s value from 10 to 11. Here's a **`--`** example:

```
myInt = 10;
--myInt;
```

This time the second line of code left **`myInt`** with a value of 9. You may have noticed that the first example showed the **`++`** following **`myInt`**, while the second example showed the **`--`** preceding **`myInt`**.

The position of the **`++`** and **`--`** operators determines when their operation is performed in relation to the rest of the statement. Placing the operator on the right side of a variable or expression (**postfix notation**) tells the compiler to resolve all values before performing the increment (or decrement) operation. Placing the operator on the left side of the variable (**prefix notation**) tells the compiler to increment (or decrement) first, then continue evaluation. Confused? The following examples should make this point clear:

```
myInt = 10;
anotherInt = myInt--;
```

The first statement assigns **`myInt`** a value of 10. In the second statement, the **`--`** operator is on **`myInt`**'s right side. This use of postfix notation tells the compiler to assign **`myInt`**'s value to **`anotherInt`** before decrementing **`myInt`**. This example leaves **`myInt`** with a value of 9 and **`anotherInt`** with a value of 10.

Here's the same example, written using prefix notation:

```
myInt = 10;
anotherInt = --myInt;
```

This time, the **`--`** is on the left side of **`myInt`**. In this case, the value of **`myInt`** is decremented before being assigned to **`anotherInt`**. The result? **`myInt`** and **`anotherInt`** are both left with a value of 9.

This use of prefix and postfix notation shows both a strength and a weakness of the C language. On the plus side, C allows you to accomplish a lot in a small amount of code. In the previous examples, we changed the value of two different variables in a single statement. C is powerful.

On the down side, C code written in this fashion can be extremely cryptic, difficult to read for even the most seasoned C programmer.

Write your code carefully.

The += and -= Operators

In C, you can place the same variable on both the left and right sides of an assignment statement. For example, the statement:

```
myInt = myInt + 10;
```

increases the value of **`myInt`** by 10. The same results can be achieved using the **`+=`** operator:

```
myInt += 10;
```

is the same as:

```
myInt = myInt + 10;
```

In the same way, the **`-=`** operator can be used to decrement the value of a variable. The statement:

```
myInt -= 10;
```

decrements the value of **`myInt`** by 10.

The *, /, *=, and /= Operators

The **`*`** and **`/`** operators each take two values and reduce them to a single value, much the same as the **`+`** and **`-`** operators do. The statement:

```
myInt = 3 * 5;
```

multiplies 3 and 5, leaving **`myInt`** with a value of 15. The statement:

```
myInt = 5 / 2;
```

divides 5 by 2 and, assuming **`myInt`** is declared as an **`int`** (or any other type designed to hold whole numbers), assigns the integral (truncated) result to **`myInt`**. The number 5 divided by 2 is 2.5. Since **`myInt`** can only hold whole numbers, the value 2.5 is truncated and the value 2 is assigned to **`myInt`**.

Math alert! Numbers like -37, 0, and 22 are known as **whole numbers** or **integers**. Numbers like 3.14159, 2.5, and .0001 are known as **fractional** or **floating point numbers**.

The **`*=`** and **`/=`** operators work much the same as their **`+=`** and **`-=`** counterparts. The statement:

```
myInt *= 10;
```

is identical to the statement:

```
myInt = myInt * 10;
```

The statement:

```
myInt /= 10;
```

is identical to the statement:

```
myInt = myInt / 10;
```

The `/` operator doesn't perform its truncation automatically. The accuracy of the result is limited by the data type of the operands. As an example, if the division is performed using **`int`**s, the result will be an **`int`**, and is truncated to an integer value.

There are several data types (such as **`float`**) which support floating point division using the `/` operator.

Using Parentheses ()

Sometimes the expressions you create can be evaluated in several ways. Here's an example:

```
myInt = 5 + 3 * 2;
```

You can add 5 + 3, then multiply the result by 2 (giving you 16). Alternatively, you can multiply 3 * 2 and add 5 to the result (giving you 11). Which is correct?

C has a set of built-in rules for resolving the order of operators. As it turns out, the **`*`** operator has a higher precedence than the **`+`** operator, so the multiplication will be performed first, yielding a result of 11.

Though it helps to understand the relative precedence of the C operators, it is hard to keep track of them all. That's why the C gods gave us parentheses! Use parentheses in pairs to define the order in which you want your operators performed. The statement:

```
myInt = ( 5 + 3 ) * 2;
```

will leave **`myInt`** with a value of 16. The statement:

```
myInt = 5 + ( 3 * 2 );
```

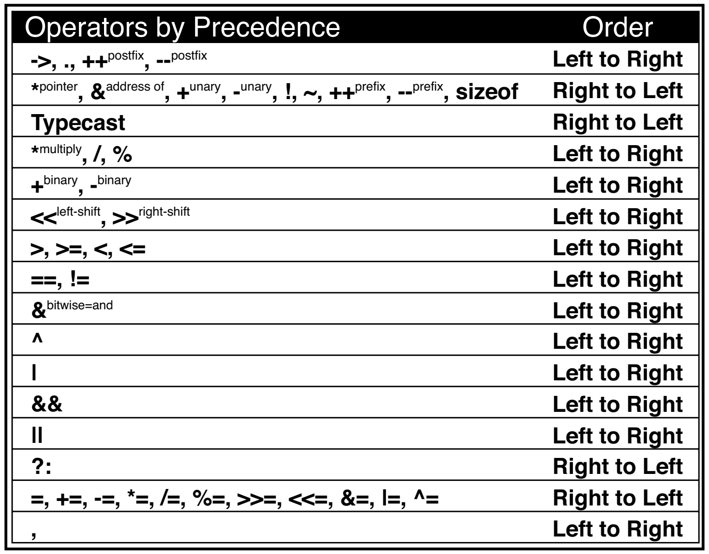
will leave **`myInt`** with a value of 11. You can use more than one set of parentheses in a statement, as long as they occur in pairs — one left parenthesis associated with each right parenthesis. The statement:

```
myInt = ( ( 5 + 3 ) * 2 );
```

will leave **`myInt`** with a value of 16.

Operator Precedence

In the previous section I referred to C's built in rules for resolving operator precedence. If you have a question about which operator has a higher precedence, look it up in the chart in Figure 5.6. Here's how the chart works.



Operators by Precedence	Order
->, ., ++$^{\text{postfix}}$, --$^{\text{postfix}}$	Left to Right
*$^{\text{pointer}}$, &$^{\text{address of}}$, +$^{\text{unary}}$, -$^{\text{unary}}$, !, ~, ++$^{\text{prefix}}$, --$^{\text{prefix}}$, sizeof	Right to Left
Typecast	Right to Left
*$^{\text{multiply}}$, /, %	Left to Right
+$^{\text{binary}}$, -$^{\text{binary}}$	Left to Right
<<$^{\text{left-shift}}$, >>$^{\text{right-shift}}$	Left to Right
>, >=, <, <=	Left to Right
==, !=	Left to Right
&$^{\text{bitwise=and}}$	Left to Right
^	Left to Right
|	Left to Right
&&	Left to Right
||	Left to Right
?:	Right to Left
=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=	Right to Left
,	Left to Right

Figure 5.6 *The relative precedence of C's built-in operators. The higher the position in the chart, the higher the precedence.*

The higher an operator is in the chart, the higher its precedence. For example, suppose you are trying to predict the result of this line of code:

```
myInt = 5 * 3 + 7;
```

First, look up the operator **`*`** in the chart. Hmmm... **`*`** seems to be in the chart twice, once with label **`pointer`** and once with the label **`multiply`**. You can tell just by looking at this line of code that we want the **`multiply`** version. The compiler is pretty smart. Just like you, it can tell that this is the **`multiply`** version of **`*`**.

OK, now look up **`+`**. Yup - it's in there twice also, once as **`unary`** and once as **`binary`**. A **`unary +`** or **`-`** is the sign that appears before a number, like +147 or -32768. In our line of code, the **`+`** operator has two operands, so clearly **`binary +`** is the one we want.

Now that you've figured out which operator is which, you can see that the **`multiply *`** is higher up on the chart than the **`binary +`**, and thus has a higher precedence. This means that the **`*`** will get evaluated before the **`+`**, as if the expression were written as:

```
myInt = (5 * 3) + 7;
```

So far so good. Now what about this line of code:

```
myInt = 27 * 6 % 5;
```

Both of these operators are on the fourth line in the chart. Which one gets evaluated first? If both operators under consideration are on the same line in the chart, the order of evaluation is determined by the entry in the chart's right-most column. In this case, the operators are evaluated from left to right. In the current example, **`%`** will get evaluated before **`*`**, as if the line of code were written:

```
myInt = 27 * (6 % 5);
```

What about this line of code:

```
myInt = 27 % 6 * 5;
```

In this case, the **`*`** will get evaluated before the **`%`**, as if the line of code were written:

```
myInt = 27 % (6 * 5);
```

Of course, you can avoid this exercise altogether with a judicious sprinkling of parentheses. As you look through the chart, you'll definitely notice some operators that you haven't learned about yet. As you read through the book and encounter new operators, check back with the chart to see where it fits in.

Sample Programs

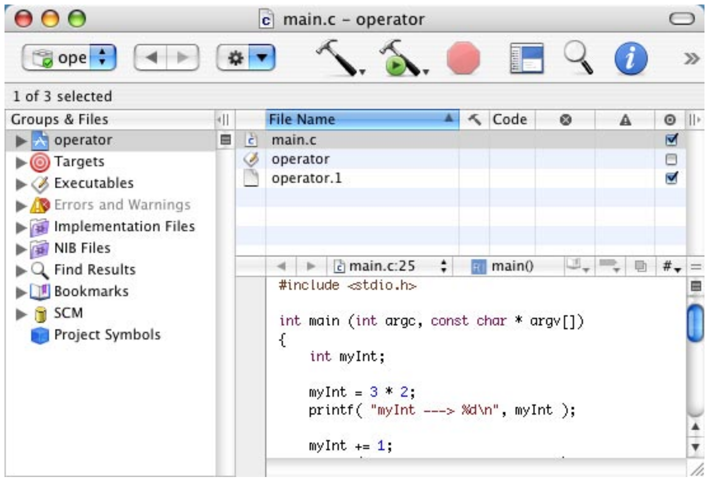
So far in this chapter, we've discussed variables (mostly of type **`int`**) and operators (mostly mathematical). The program examples on the following pages combine variables and operators into useful C statements. We'll also learn about a powerful part of the Standard Library, the **`printf()`** function.

Opening operator.xcode

Our next program, **`operator`**, provides a testing ground for some of the operators covered in the previous sections. *main.c* declares a variable (**`myInt`**) and uses a series of statements to change the value of the variable. By including a **`printf()`** after each of these statements, *main.c* makes it easy to follow the variable, step by step, as its value changes.

In Xcode, close any project windows that may be open. In the Finder, locate the *Learn C Projects* folder and the *05.01 – operator* subfolder, then double-click the file *operator.xcode*. The operator project window should appear (Figure 5.7).

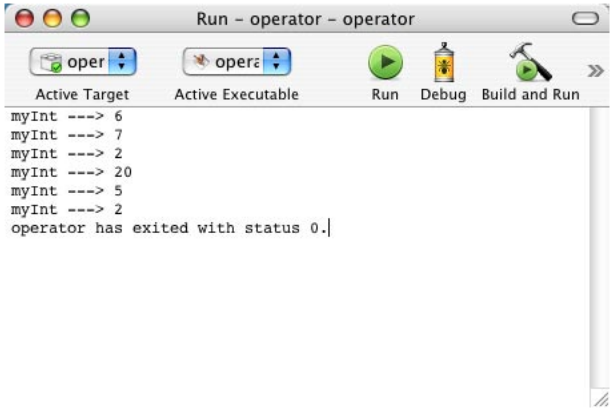
Remember, you can double-click on the source code file name to open a new editing window, or you can click the *Show/Hide Editor* icon to open an editing pane within the project window.



```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int myInt;

    myInt = 3 * 2;
    printf( "myInt ---> %d\n", myInt );

    myInt += 1;
```

Figure 5.7 *The operator project window.*

Run **`operator`** by selecting Build and Run from the Build menu. Xcode will first attempt to compile *main.c*, then run it. Compare your output to that shown in Figure 5.8. They should be about the same.



```
myInt ---> 6
myInt ---> 7
myInt ---> 2
myInt ---> 20
myInt ---> 5
myInt ---> 2
operator has exited with status 0.
```

Figure 5.8 *The operator's output.*

Stepping Through the Source Code

Before we walk through the source code in *main.c*, you might want to bring the source code up on your screen (double-click on the name *main.c* in the project window, or click on the *Show/Hide Editor* icon).

main.c starts off with a **#include** statement that gives us access to a bunch of Standard Library functions, including **printf()**:

```
#include <stdio.h>
```

main() starts out by **defining** an **int** named **myInt**.

```
int main (int argc, const char * argv[])
{
    int myInt;
```

Note that earlier I used the term "declaring a variable" and now I'm using the term "defining". What's the difference? A variable *declaration* is any statement that specifies a variable's name and type. The line:

```
int myInt;
```

certainly does that. A variable *definition* is a declaration that causes memory to be allocated for the variable. Since the previous statement does cause memory to be allocated for myInt, it does qualify as a definition. Later in the book, you'll see some declarations that don't qualify as definitions. For now, just remember, a definition causes memory to be allocated.

At this point in the program (after **myInt** has been declared but before any value has been assigned to it), **myInt** is said to be **uninitialized**. In computerese, the term **initialization** refers to the process of establishing a variable's value for the first time. A variable that has been declared, but that has not had a value assigned to it, is said to be uninitialized. You **initialize** a variable the first time you assign a value to it.

Since **myInt** was declared to be of type **int**, and since Xcode is currently set to use 4 byte **int**s, 4 bytes of memory were reserved for **myInt**. Since we haven't placed a value in those 4 bytes yet, they could contain any value at all. Some compilers place a value of 0 in a newly allocated variable, but there are some compilers that do not. The key is, don't depend on a variable being preset to some specific value. If you want a variable to contain a specific value, assign the value to the variable yourself!

Later in the book, you'll learn about global variables. Global variables are always set to 0 by the compiler. All the variables used in this chapter are local variables, not global variables. Local variables are not guaranteed to be initialized by the compiler.

The next line of code uses the `*` operator to assign a value of 6 to **`myInt`**. Following that, we use **`printf()`** to display the value of **`myInt`** in the console window.

```
myInt = 3 * 2;
printf( "myInt ---> %d\n", myInt );
```

The code between **`printf()`**'s left and right parentheses is known as a **parameter list**. The **parameters** in a parameter list (also known as **arguments**) are automatically provided to the function you are calling (in this case, **`printf()`**). The receiving function can use the parameters passed to it to determine its next course of action. We'll get into the specifics of parameter passing in Chapter 7. For the moment, let's talk about **`printf()`** and the parameters used by this Standard Library function.

The first parameter passed to **`printf()`** defines what will be drawn in the console window. The simplest call to **`printf()`** uses a quoted text string as its only parameter. A quoted text string consists of a pair of double-quote characters (**"**) with zero or more characters between them. For example, this call of **`printf()`**:

```
printf( "Hello!" );
```

will draw the characters **`Hello!`** in the console window. Notice that the double-quote characters are not part of the text string.

You can request that **`printf()`** draw a variable's value in the midst of the quoted string. In the case of an **`int`**, do this by embedding the two characters **`%d`** within the first parameter and by passing the **`int`** as a second parameter. **`printf()`** will replace the **`%d`** with the value of the **`int`**.

In these two lines of code, we first set **`myInt`** to 6, use **`printf()`** to print the value of **`myInt`** in the console window.

```
myInt = 3 * 2;
printf( "myInt ---> %d\n", myInt );
```

This code produces this line of output in the console window:

```
myInt ---> 6
```

The two characters "**`\n`**" in the first parameter represent a carriage return and tell **`printf()`** to move the cursor to the beginning of the next line before it prints any more characters.

The **`%d`** is known as a **format specifier**. The **`d`** in the format specifier tells **`printf()`** that you are printing an integer variable, such as an **`int`**.

You can place any number of **`%`** specifications in the first parameter, as long as you follow the first parameter by the appropriate number of variables. Here's another example:

```
int  var1, var2;

var1 = 5;
var2 = 10;
printf( "var1 = %d\n\nvar2 = %d\n", var1, var2
 );
```

will draw the text

```
var1 = 5

var2 = 10
```

in the console window. Notice the blank line between the two lines of output. It was caused by the "**`\n\n`**" in the first **`printf()`** parameter. The first carriage return placed the cursor at the beginning of the next console line (directly under the **`v`** in **`var1`**). The second carriage return moved the cursor down one more line, leaving a blank line in its path.

Let's get back to our source code. The next line of *main.c* increments **`myInt`** from 6 to 7, and prints the new value in the console window.

```
myInt += 1;
printf( "myInt ---> %d\n", myInt );
```

The next line decrements **`myInt`** by 5, and prints its new value of 2 in the console window.

```
myInt -= 5;
printf( "myInt ---> %d\n", myInt );
```

Next, **`myInt`** is multiplied by 10, and its new value of 20 is printed in the console window.

```
myInt *= 10;
printf( "myInt ---> %d\n", myInt );
```

Next, **`myInt`** is divided by 4, resulting in a new value of 5.

```
    myInt /= 4;
    printf( "myInt ---> %d\n", myInt );
```

Finally, **`myInt`** is divided by 2. Since 5 divided by 2 is 2.5 (not a whole number), a truncation is performed and **`myInt`** is left with a value of 2.

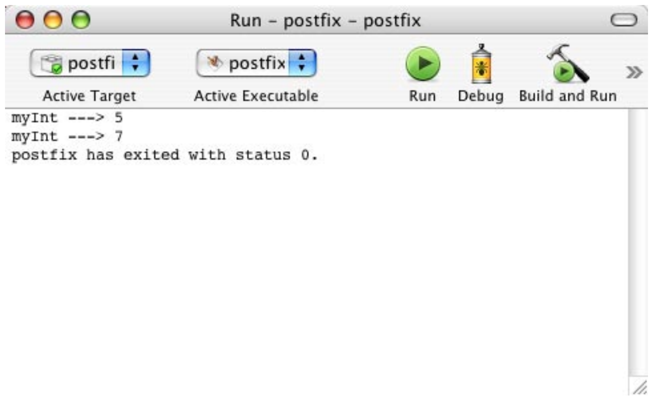
```
    myInt /= 2;
    printf( "myInt ---> %d", myInt );

    return 0;
  }
```

Opening postfix.xcode

Our next program demonstrates the difference between postfix and prefix notation (remember the **`++`** and **`--`** operators defined earlier in the chapter?) If you have a project open in Xcode, close it. In the Finder, go into the *Learn C Projects* folder, then into the *05.02 - postfix* subfolder, and double-click on the project file *postfix.xcode*.

Take a look at the source code in the file *main.c* and try to predict the result of the two **`printf()`** calls before you run the program. Careful, this one's tricky.

Once your guesses are locked in, select Build and Run from the Build menu. How'd you do? Compare your two guesses with the output in Figure 5.9. Let's look at the source code.



```
myInt ---> 5
myInt ---> 7
postfix has exited with status 0.
```

Figure 5.9 *The output generated by postfix.*

Stepping Through the Source Code

The first half of *main.c* is pretty straightforward. The variable **`myInt`** is defined to be of type **`int`**. Then, **`myInt`** is assigned a value of 5. Next comes the tricky part.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int        myInt;

 myInt = 5;
```

The first call to **`printf()`** actually has a statement embedded in it. This is another great feature of the C language. Where there's room for a variable, there's room for an entire statement. Sometimes it's convenient to perform two actions within the same line of code. For example, this line of code:

```
printf( "myInt ---> %d\n", myInt = myInt * 3
 );
```

first triples the value of **`myInt`**, then passes the result (the tripled value of **`myInt`**) on to **`printf()`**. The same could have been accomplished using two lines of code:

```
myInt = myInt * 3;
printf( "myInt ---> %d\n", myInt );
```

In general, when the compiler encounters an assignment statement where it expects a variable, it first completes the assignment, then passes on the result of the assignment as if it were a variable. Let's see this technique in action.

In *main.c*, our friend the postfix operator emerges again. Just prior to the two calls of **`printf()`**, **`myInt`** has a value of 5. The first of the two **`printf()`**'s increments the value of **`myInt`** using postfix notation:

```
  printf( "myInt ---> %d\n", myInt++ );
```

The use of postfix notation means that the value of **`myInt`** will be passed on to **`printf()`** before **`myInt`** is incremented. This means that the first **`printf()`** will accord **`myInt`** a value of 5. However, when the statement is finished, **`myInt`** will have a value of 6.

The second **`printf()`** acts in a more rational (and preferable) manner. The prefix notation guarantees that **`myInt`** will be incremented (from 6 to 7) before its value is passed on to **`printf()`**.

```
  printf( "myInt ---> %d", ++myInt );

  return 0;
}
```

Can you break each of these `printf()`s into two separate statements? Give it a try, then read on...

The first `printf()` looks like this:

```
printf( "myInt ---> %d\n", myInt++ );
```

Here's the two statement version:

```
printf( "myInt ---> %d\n", myInt );

myInt++;
```

Notice that the statement incrementing `myInt` was placed after the `printf()`. Do you see why? The postfix notation makes this necessary. Run through both versions and verify this for yourself.

The second `printf()` looks like this:

```
printf( "myInt ---> %d", ++myInt );
```

Here's the two statement version:

```
++myInt;

printf( "myInt ---> %d\n", myInt );
```

This time the statement incrementing `myInt` came before the `printf()`. This time, it's the prefix notation that makes this necessary. Again, go through both versions and verify this for yourself.

The purpose of demonstrating the complexity of the postfix and prefix operators is twofold. On one hand, it's extremely important that you understand exactly how these operators work from all angles. This will allow you to write code that works and will aid you in making sense of other programmers' code.

On the other hand, embedding prefix and postfix operators within function parameters may save you lines of code but, as you can see, may prove a bit confusing. So what's a coder to do? Clarity before brevity. Make sure your code is readable. After all, you will likely have to go back and edit it at some point. Readable code is *much* easier to maintain.

Backslash Combinations

The last program in Chapter 5, **`slasher`**, demonstrates several different **backslash combinations**. A backslash combination combines a backslash character ("\") and a second character to produce a specific result when the combination is printed in the console window. One backslash combination you've seen a lot of in this book is "\n", which produces a new line in the console.

C allows you to embed any number of backslash combinations in a text string. For example, this line of code:

```
printf( "Hello\n\nGoodbye" );
```

produces this output in the console window:

```
Hello

Goodbye
```

The single blank line between "Hello" and "Goodbye" was caused by the two "`\n`" characters. The first "`\n`" would have caused the console to put "Goodbye" on the line immediately below "Hello". The second "`\n`" moved the "Goodbye" down one more line.

There are a number of backslash combinations. We'll discuss a few of the more interesting ones.

"`\r`" causes the cursor to move to the beginning of the *same* line. This allows you to draw some text then go back and overwrite the same text.

"`\b`" is a backspace character. This has the same affect as if you hit the delete key while you were typing, erasing the last character typed.

"`\\`" allows you to place a backslash character in a string. Think about this for a moment. If you simply embedded a backslash character in your string, the compiler would attempt to combine the backslash with the very next character, producing some unpredictable results. Unpredictable is bad.

"`\"`" allows you to place a quote character in a string. When the compiler first sees a double-quote character in your code, it assumes you are starting a text string. It keeps reading, reading, reading, until it encounters a second, matching double-quote character. The second quote tells the compiler that it has reached the end of the string. So how do you place a quote character inside a string without ending the string? Easy. Use the "`\"`" where you want the quote to appear.

"`\t`" allows you to place a tab in a string.

"`\a`" embeds a single beep in the string.

Support for Backslash Combinations

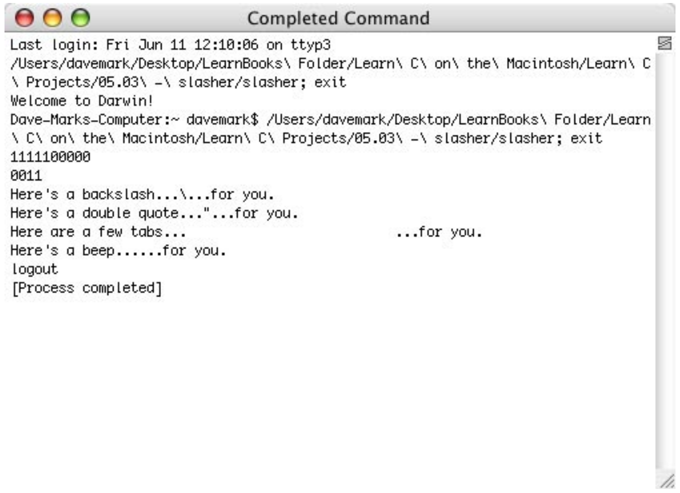
Backslash combinations stem from the olden days, where all programs ran on video displays with a fixed number of rows and columns. The backslash combinations helped programmers overcome the limitations of these displays, giving them a bit more control. As computers evolved, many of these backslash combinations became unnecessary. Over time, many development environments stopped supporting all but the most basic of these.

To see this for yourself, we'll run our next program, **`slasher`**, using the Terminal application that ships with Mac OS X. The Terminal app implements a classic console window that supports all the well known backslash combinations, just like an old video display terminal. We'll use the built in Unix tools that you installed when you installed Xcode at the beginning of the book to compile the program as well.

Though Xcode doesn't support many of the backslash combinations that we used in **`slasher`**, I built a project file for it anyway. After you are done playing with the Unix version of slasher, take the Xcode version for a spin. You'll find it in the *Learn C Projects* folder, in the *05.03 – slasher* subdirectory.

Running slasher

In the Finder, go to the *Learn C Projects* folder, into the *05.03 – slasher* subdirectory, and double-click on the file named *slasher*. The Terminal application will launch and a new window will appear, similar to the one shown in Figure 5.10. If you've never used the Terminal before, this may look a bit cryptic. The Terminal is similar to Xcode's console window. No graphics, just a scrolling series of lines of text.



```
Last login: Fri Jun 11 12:10:06 on ttyp3
/Users/davemark/Desktop/LearnBooks\ Folder/Learn\ C\ on\ the\ Macintosh/Learn\ C
\ Projects/05.03\ -\ slasher/slasher; exit
Welcome to Darwin!
Dave-Marks-Computer:~ davemark$ /Users/davemark/Desktop/LearnBooks\ Folder/Learn
\ C\ on\ the\ Macintosh/Learn\ C\ Projects/05.03\ -\ slasher/slasher; exit
1111100000
0011
Here's a backslash...\...for you.
Here's a double quote..."...for you.
Here are a few tabs...                              ...for you.
Here's a beep......for you.
logout
[Process completed]
```

Figure 5.10 *Running slasher using the Terminal application.*

You can ignore the first six lines of text in the Terminal window. The key lines of output to pay attention to are these six:

```
1111100000
0011
Here's a backslash...\...for you.
Here's a double quote..."...for you.
Here are a few tabs...
   ...for you.
Here's a beep......for you.
```

As we step through the source code, you'll see a series of six **`printf()`**s, each of which corresponds to one of these lines of output. Once we finish going through the source code, we'll take a shot at compiling the source using the Unix compiler and the Terminal, instead of using Xcode.

Stepping Through the Source Code

main.c consists of a series of **`printf()`**s, each of which demonstrates a different backslash combination. The first **`printf()`** prints a series of ten zeros, followed by the characters **`\r`** (also known as the backslash combination **`\r`**). The **`\r`** backslash combination generates a carriage return without a line feed, leaving the cursor at the beginning of the current line (unlike **`\n`**, which leaves the cursor at the beginning of the next line down).

```
#include <stdio.h>


int main (int argc, const char * argv[])
{
 printf( "0000000000\r" );
```

The next **`printf()`** prints five 1s over the first five 0s, as if someone had printed the text string **`"1111100000"`**. The **`\n`** at the end of this **`printf()`** moves the cursor to the beginning of the next line in the console window.

```
   printf( "11111\n" );
```

The next **`printf()`** demonstrates **`\b`**, the backspace backslash combination. **`\b`** tells **`printf()`** to back up one character so that the next character printed replaces the last character printed. This **`printf()`** sends out four 0s, backspaces over the last two, then prints two 1s. The result is as if you had printed the string **`"0011"`**.

```
   printf( "0000\b\b11\n" );
```

The **`\`** can also be used to cancel a character's special meaning within a quoted string. For example, the backslash combination **`\\`** generates a single **`\`** character. The difference is, this **`\`** loses its special backslash powers. It doesn't affect the character immediately following it.

The backslash combination **`\"`** generates a **`"`** character, taking away the special meaning of the **`"`**. As we said earlier, without the **`\`** before it, the **`"`** character would mark the end of the quoted string. The **`\`** allows you to include a **`"`** inside a quoted string.

The backslash combinations **`\\`** and **`\"`** are demonstrated in the next two **`printf()`**s:

```
   printf( "Here's a backslash...\\...for you.\
n" );
   printf( "Here's a double quote...\"...for
you.\n" );
```

The **`\t`** combination generates a single tab character. The console window has a tab stop every eight spaces. Here's a **`printf()`** example:

```
   printf( "Here's a few tabs...\t\t\t\t...for
you.\n" );
```

While the Mac offers a host of sound options, most text-based computer consoles offer one: the beep. While a beep isn't quite as interesting as a Clank! or a Boing!, it can still serve a useful purpose. The **`\a`** backslash combination provides a simple way to make your Mac beep.

```
   printf( "Here's a beep...\a...for you.\n" );

   return 0;
}
```

Building slasher

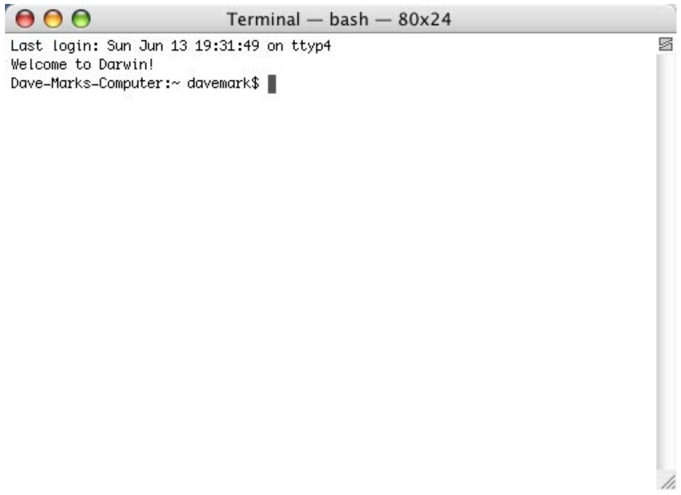
This section is completely optional. You can skip it entirely, or scan it to follow along, or do every darned step along the way. We're going to use the Terminal to compile the **`slasher`** source code into a runnable Unix application. In effect, we're going to rebuild the **`slasher`** app that you just ran.

In your home directory, create a new folder called *slasher*. Your home directory is the directory with the house icon, named with your login name. For example, my home directory is in the *Users* folder and is called *davemark*.

Next, locate the folder containing the **`slasher`** project. You'll find it in the *Learn C Projects* directory, in the *05.03 – slasher* subdirectory. Inside that folder, you'll find a file named *main.c* which contains the **`slasher`** source code. Use the Finder to drag a copy of *main.c* into the new *slasher* folder you created in your home directory.

If Terminal is running, open a new window by selecting New Shell from the File menu. If Terminal is *not* running, launch it. You'll find it in the *Applications* folder, in the *Utilities* subfolder.

At this point, you should have a *slasher* folder in your home folder containing a copy of slasher's *main.c* file and a new Terminal window which looks like the one shown in Figure 5.11.



```
Last login: Sun Jun 13 19:31:49 on ttyp4
Welcome to Darwin!
Dave-Marks-Computer:~ davemark$
```

Figure 5.11 *A brand new Terminal window.*

We're now going to type some Unix commands into the Terminal window. Our first goal is to make sure we can see the new *slasher* folder we just created in our home directory. Type this command, followed by a carriage return:

```
cd ~
```

Note that there is a space in between the **`cd`** and the tilde character ("**`~`**"). This command tells Unix to change your directory (cd) to the tilde directory. In Unix-speak, the tilde directory is always your home directory.

Next, you'll type the command:

```
ls
```

followed by a carriage return. This command asks Unix to list all the visible files in the current directory which, in this case, is your home directory. Here's the list I got:

```
Desktop    Library    Music      Public
 slasher
Documents  Movies     Pictures   Sites
```

Note that our newly created *slasher* directory is in this list. If you don't see *slasher* in your list, chances are good that you created the directory in the wrong place. Go find the folder, drag it into your home directory, then go back to Terminal and do another **`ls`**.

Next, let's go into the *slasher* directory and make sure the *main.c* file is there. Issue these two commands:

```
cd slasher
ls
```

Remember to type a carriage return after each command. The first command changes directories to the *slasher* directory. The second command lists the visible files in that directory. Here's the results of my **`ls`**:

```
main.c
```

If your *slasher* directory is empty, you did not successfully copy *main.c* into the *slasher* folder you created. Go fix that.

Once **`ls`** shows *main.c* in the *slasher* directory, you are ready to do a compile. Type this command:

```
cc -o slasher main.c
```

Be sure to end it with a carriage return. You've just asked Unix to compile the C code in the file *main.c* and link the resulting object code into an executable file named *slasher*. The "**`-o`**" tells the cc command that you want to name the output, the word "**`slasher`**" tells it the name to use. If you left out the "**`-o slasher`**" from the command, **`cc`** would put the output in a file named **`a.out`**.

To see the results of your compile, do another **`ls`**. Here's the results you should see:

```
main.c  slasher
```

Notice that a new file named *slasher* has been created. You can run this program by typing this command:

```
./slasher
```

Note the "**./**" before the word **slasher**. This tells Unix to run the **slasher** in the current directory, as opposed to some other file named **slasher** that might be elsewhere in its search path.

Here's the output I saw when I ran my copy of slasher:

```
1111100000
0011
Here's a backslash...\...for you.
Here's a double quote..."...for you.
Here are a few tabs...
    ...for you.
Here's a beep......for you.
```

Feel free to quit Terminal. Your work here is done.

Those are all the sample programs for this chapter. Before we move on, however, I'd like to talk to you about something personal. It's about your coding habits.

Sprucing Up Your Code

You are now in the middle of your C learning curve. You've learned about variables, types, functions, and bytes. You've learned about an important part of the Standard Library, the function **printf()**. It's at this point in the learning process that programmers start developing their coding habits.

Coding habits are the little things programmers do that make their code a little bit different (and hopefully better!) than anyone else's. Before you get too set in your ways, here are a few coding habits you can, and should, add to your arsenal.

Source Code Spacing

You may have noticed the tabs, spaces, and blank lines scattered throughout the sample programs. These are known in C as **white space**. With a few exceptions, white space is ignored by C compilers. Believe it or not, as far as the C compiler goes, this program:

```
#include <stdio.h>
int main (int argc,
const char * argv[]){
 int myInt;myInt

=
5
;
printf("myInt=",myInt);}
```

is equivalent to this program:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int myInt;

 myInt = 5;
 printf( "myInt =", myInt );
}
```

The C compiler doesn't care if you put five statements per line, or if you put 20 carriage returns between your statements and your semicolons. One thing the compiler won't let you do is place white space in the middle of a word, such as a variable or function name. For example, this line of code:

```
my  Int = 5;
```

won't compile. Instead of a single variable named **`myInt`**, the compiler sees two items, one named **`my`** and the other named **`Int`**. Too much white space can confuse the compiler.

Too little white space can also confuse the compiler. For example, this line of code won't compile:

```
intmyInt;
```

The compiler needs at least one piece of white space to tell it where the type ends and where the variable begins. On the other hand, as you've already seen, this line compiles just fine:

```
myInt=5;
```

Since a variable name can't contain the character "**`=`**", the compiler has no problem telling where the variable ends and where the operator begins.

As long as your code compiles properly, you're free to develop your own white-space style. Here are a few hints...

- Place a blank line between your variable declarations and the rest of your function's code. Also, use blank lines to group related lines of code.

- Sprinkle single spaces throughout a statement. Compare this line:

```
printf("myInt=",myInt);
```

  with this line:

```
printf( "myInt =", myInt );
```

The spaces make the second line easier to read.

- When in doubt, use parentheses. Compare this line:

```
myInt=var1+2*var2+4;
```

with this line:

```
myInt = var1 + (2*var2) + 4;
```

What a difference parentheses and spaces make!

- Always start variable names with a lower-case letter, using an upper-case letter at the start of each subsequent word in the name. This yields variable names such as **`myVar`**, **`areWeDone`**, and **`employeeName`**.

- Always start function names with an upper-case letter, using an upper-case letter at the start of each subsequent word in the name. This yields function names such as **`DoSomeWork()`**, **`HoldThese()`**, and **`DealTheCards()`**.

These hints are merely suggestions. Use a set of standards that make sense for you and the people with whom you work. The object here is to make your code as readable as possible.

Comment Your Code

One of the most critical elements in the creation of a computer program is clear and comprehensive documentation. When you deliver your award-winning graphics package to your customers, you'll want to have two sets of documentation. One set is for your customers, who'll need a clear set of instructions that guide them through your wonderful new creation.

The other set of documentation consists of the comments you'll weave throughout your code. Source code comments act as a sort of narrative, guiding a reader through your source code. You'll include comments that describe how your code works, what makes it special, and what to look out for when changing it. Well-commented code includes a comment at the beginning of each function that describes the function, the function parameters, and the function's variables. It's also a good idea to sprinkle individual comments among your source code statements, explaining the role each line plays in your program's algorithm. How do you add a comment to your source code? Take a look...

All C compilers recognize the sequence **`/*`** as the start of a comment and will ignore all characters until they hit the sequence **`*/`** (the end of comment characters). Here's some commented code:

```
int main (int argc, const char * argv[])
{
 int numPieces;/* Number of pieces of pie left
 */

 numPieces = 8;   /* We started with 8 pieces
 */

 numPieces--;     /* Marge had a piece  */
 numPieces--;     /* Lisa had a piece  */
 numPieces -= 2;/* Bart had two pieces!!  */
 numPieces -= 4;/* Homer had the rest!!!  */

 printf( "Slices left = %d", numPieces );
                       /* How about
                              some cake
                              instead?  */
 return 0;
}
```

Notice that, although most of the comments fit on the same line, the last comment was split between three lines. The above code will compile just fine.

Most modern C compilers will also accept the C++ commenting convention. C++ ignores the remainder of a line of code, once it encounters the characters "//". For example, this line of code combines both comment styles:

```
printf( "Hello" /* C comment */ ); //
C++ comment!!!
```

Use the C++ comment mechanism if you are sure you won't be porting your code to a C compiler that doesn't understand the C++ mechanism.

Since each of the programs in this book are examined in detail, line by line, the comments were left out. This was done to make the examples as simple as possible. In this instance, do as we say, not as we do. Comment your code. No excuses!

What's Next?

This chapter introduced the concepts of variables and operators, tied together in C statements, separated by semicolons. We looked at several examples, each of which made heavy use of the Standard Library function **`printf()`**. We learned about the console window, quoted strings, and backslash combinations.

Chapter 6 will increase our programming options significantly, introducing C control structures such as the **`for`** loop and the **`if ... then ... else`** statement. Get ready to expand your C-programming horizons. See you in Chapter 6.

Exercises

1) Find the error in each of the following code fragments:

```
a. printf( Hello, world );

b. int    myInt  myOtherInt;

c. myInt =+ 3;

d. printf( “myInt = %d” );

e. printf( “myInt = “, myInt );

f. printf( “myInt = %d\”, myInt );

g. myInt + 3 = myInt;

h. int main (int argc, const char * argv[])
   {
          int   myInt;
          myInt = 3;
          anotherInt = myInt;

          return 0;
   }
```

2) Compute the value of **`myInt`** after each code fragment is executed:

```
a. myInt = 5;
   myInt *= (3+4) * 2;

b. myInt = 2;
   myInt *= ( (3*4) / 2 ) - 9;

c. myInt = 2;
   myInt /= 5;
   myInt--;

d. myInt = 25;
   myInt /= 3 * 2;

e. myInt = (3*4*5) / 9;
   myInt -= (3+4) * 2;

f. myInt = 5;
   printf( “myInt = %d”, myInt = 2 );

g. myInt = 5;
   myInt = (3+4) * 2;

h. myInt = 1;
   myInt /= (3+4) / 6;
```

Chapter 6 Controlling Your Program's Flow

So far, you’ve learned quite a bit about the C language. You know about functions (especially one named **`main()`**), which are made up of statements, each of which is terminated by a semicolon. You know about variables, which have a name and a type. Up to this point, you’ve dealt with variables of type **`int`**.

You also know about operators, such as **`=`**, **`+`**, and **`+=`**. You’ve learned about postfix and prefix notation, and the importance of writing clear, easy-to-understand code. You’ve learned about the Standard Library, a set of functions that comes as standard equipment with every C programming environment. You’ve also learned about **`printf()`**, an invaluable component of the Standard Library.

Finally, you’ve learned a few housekeeping techniques to keep your code fresh, sparkling, and readable. Comment your code, because your memory isn’t perfect, and insert some white space to keep your code from getting too cramped.

Flow Control

One thing you haven’t learned about the C language is **flow control**. The programs we’ve written so far have all consisted of a straightforward series of statements, one right after the other. Every statement is executed in the order it occurred.

Flow control is the ability to control the order in which your program’s statements are executed. The C language provides several keywords you can use in your program to control your program’s flow. One of these is the **`if`** keyword.

The if Statement

The **`if`** keyword allows you to choose between several options in your program. In English, you might say something like this:

```
If it’s raining outside I’ll bring my
 umbrella;
otherwise I won’t.
```

In this sentence, you’re using **`if`** to choose between two options. Depending on the weather, you'll do one of two things. You'll bring your umbrella or you won't bring your umbrella. C's **`if`** statement gives you this same flexibility. Here's an example:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int myInt;

 myInt = 5;

 if ( myInt == 0 )
    printf( "myInt is equal to zero." );
 else
    printf( "myInt is not equal to zero." );

 return 0;
}
```

This program declares **`myInt`** to be of type **`int`** and sets the value of **`myInt`** to 5. Next, we use the **`if`** statement to test whether **`myInt`** is equal to 0. If **`myInt`** is equal to 0 (which we know is not true), we'll print one string. Otherwise, we'll print a different string. As expected, this program prints the string **`"myInt is not equal to zero"`**.

`if` statements come two ways. The first, known as plain old **`if`**, fits this pattern:

```
if ( expression )
 statement
```

An **`if`** statement will always consist of the word **`if`**, a left parenthesis, an expression, a right parenthesis, and a statement. (We'll define both expression and statement in a minute.) This first form of **`if`** executes the statement if the expression in parentheses is true. An English example of the plain **`if`** might be:

```
If it's raining outside, I'll bring my
 umbrella.
```

Notice that this statement only tells us what will happen if it's raining outside. No particular action will be taken if it is not raining.

The second form of **`if`**, known as **`if-else`**, fits this pattern:

```
if ( expression )
 statement
else
 statement
```

An **`if-else`** statement will always consist of the word **`if`**, a left parenthesis, an expression, a right parenthesis, a statement, the word **`else`**, and a second statement. This form of **`if`** executes the first statement if the expression is true, and executes the second statement if the expression is false. An English example of an **`if-else`** statement might be:

```
If it’s raining outside, I’ll bring my
 umbrella,
otherwise I won’t.
```

Notice that this example tells us what will happen if it is raining outside (I’ll bring my umbrella) and if it isn’t raining outside (I won’t bring my umbrella). The example programs presented later in the chapter demonstrate the proper use of both **`if`** and **`if-else`**.

Our next step is to define the terms **expression** and **statement**.

Expressions

In C, an expression is anything that has a value. For example, a variable is a type of expression, since variables always have a value. (Even uninitialized variables have a value—we just don’t know what the value is!) The following are all examples of expressions:

- `myInt + 3`
- `( myInt + anotherInt ) * 4`
- `myInt++`

An assignment statement is also an expression. Can you guess the value of an assignment statement? Think back to Chapter 5. Remember when we included an assignment statement as a parameter to **`printf()`**? The value of an assignment statement is the value of its left side. Check out the following code fragment:

```
myInt = 5;
myInt += 3;
```

Both of these **`statement`**s qualify as expressions. The value of the first expression is 5. The value of the second expression is 8 (because we added 3 to

myInt's previous value).

Literals can also be used as expressions. The number 8 has a value. Guess what? Its value is 8. All expressions, no matter what their type, have a numerical value.

Technically, there is an exception to this rule. The expression **(void) 0** has no value. In fact, any value or variable cast to type **void** has no value. Ummm, but Dave, what's a cast? What is type **void**? We'll get to both of these topics later in the book. For the moment, when you see **void**, think "no value".

True Expressions

Earlier, we defined the **if** statement as follows:

```
if ( expression )
 statement
```

We then said the statement gets executed if the expression is true. Let's look at C's concept of truth.

Everyone has an intuitive understanding of the difference between true and false. I think we'd all agree that the statement:

```
5 equals 3
```

is false. We'd also agree that the statement:

```
5 and 3 are both greater than 0
```

is true. This intuitive grasp of true and false carries over into the C language. In the case of C, however, both true and false have numerical values. Here's how it works.

In C, any expression that has a value of 0 is said to be false. Any expression with a value other than 0 is said to be true. As stated earlier, an **if** statement's statement gets executed if its expression is true. To put this more accurately:

- An **if** statement's statement gets executed if (and only if) its expression has a value other than 0.

Here's an example:

```
myInt = 27;

if ( myInt )
 printf( "myInt is not equal to 0" );
```

The **if** statement in this piece of code first tests the value of **myInt**. Since **myInt** is not equal to 0, the **printf()** gets executed.

Comparative Operators

C expressions have a special set of operators, called **comparative operators.** Comparative operators compare their left sides with their right sides and produce a value of either `1` or `0`, depending on the relationship of the two sides.

For example, the operator `==` determines whether the expression on the left is equal in value to the expression on the right. The expression:

```
myInt == 5
```

evaluates to 1 if **`myInt`** is equal to 5, and to 0 if **`myInt`** is not equal to 5. Here's an example of the `==` operator at work:

```
if ( myInt == 5 )
 printf( "myInt is equal to 5" );
```

If **`myInt`** is equal to 5, the expression **`myInt == 5`** evaluates to 1 and **`printf()`** gets called. If **`myInt`** wasn't equal to 5, the expression evaluates to 0 and the **`printf()`** is skipped. Just remember, the key to triggering an **`if`** statement is an expression that resolves to a value other than 0.

Figure 6.1 shows some of the other comparative operators. You'll see some of these operators in the example programs later in the chapter.

Operator	Resolves to 1 if...
==	left side is equal to right
<=	left side is less than or equal to right
>=	left side is greater than or equal to right
>=	left side is less than right
>=	left side is greater than right
>=	left side is not equal to right

***Figure 6.1** Comparitive Operators*

Logical Operators

The C standard provides a pair of constants that really come in handy when dealing with our next set of operators. The constant **`true`** has a value of 1, while the constant **`false`** has a value of 0. You can use these constants in your programs to make them a little easier to read. Read on, and you'll see why.

In addition to **`true`** and **`false`**, most C environments also provide the constants **`TRUE`** and **`FALSE`** (with values of 1 and 0 respectively). Some people prefer **`TRUE`** and **`FALSE`**, others prefer **`true`** and **`false`**. Pick a pair and stick with them. We'll work with **`true`** and **`false`** throughout the rest of the book.

When you get to the **`truthTester`** program in just a bit, you'll find a **`#include`** of the file **`<c.h>`** at the beginning of the file. This is where true and false are defined.

Our next set of operators are known, collectively, as **logical operators.** The set of logical operators are modeled on the mathematical concept of truth tables. If you don't know much about truth tables (or are just frightened by mathematics in general), don't panic. Everything you need to know is outlined in the next few paragraphs.

The first of the set of logical operators is the **`!`** operator. The **`!`** operator turns **`true`** into **`false`** and **`false`** into **`true`**. Figure 6.2 shows the truth table for the **`!`** operator. In this table, **`T`** stands for **`true`** and **`F`** stands for **`false`**. The letter **`A`** in the table represents an expression. If the expression **`A`** is **`true`**, applying the **`!`** operator to **`A`** yields the value **`false`**. If the expression **`A`** is **`false`**, applying the **`!`** operator to **`A`** yields the value **`true`**. The **`!`** operator is commonly referred to as the **`NOT`** operator. **`!A`** is pronounced "**`NOT A`**".

A	!A
T	F
F	T

***Figure 6.2** The truth table for the* **`!`** *operator.*

Here's a piece of code that demonstrates the **`!`** operator:

```
int  myFirstInt, mySecondInt;

myFirstInt = false;
mySecondInt = ! myFirstInt;
```

First, we declare two **`int`**s. We assign the value **`false`** to the first **`int`**, then use the **`!`** operator to turn the **`false`** into a **`true`** and assign it to the second **`int`**. This is really important. Take another look at Figure 6.2. The **`!`** operator converts **`true`** into **`false`** and **`false`** into **`true`**. What this really means is that **`!`** converts 1 to 0 and 0 to 1. This really comes in handy when you are working with an if statement's expression, like this one:

```
if ( mySecondInt )
 printf( "mySecondInt must be true" );
```

The previous chunk of code translated **`mySecondInt`** from **`false`** to **`true`**, which is the same thing as saying that **`mySecondInt`** has a value of 1. Either way, **`mySecondInt`** will cause the **`if`** to fire, and the **`printf()`** will get executed.

Take a look at this piece of code:

```
if ( ! mySecondInt )
 printf( "mySecondInt must be false" );
```

This **`printf()`** will get executed if **`mySecondInt`** is **`false`**. Do you see why? If **`mySecondInt`** is **`false`**, then **`!mySecondInt`** must be **`true`**.

The **`!`** operator is a **unary** operator. Unary operators operate on a single expression (the expression to the right of the operator). The other two logical operators, **`&&`** and `||`, are **binary** operators. Binary operators, such as the **`==`** operator presented earlier, operate on two expressions, one on the left side and one on the right side of the operator.

The **`&&`** operator is commonly referred to as the **`and`** operator. The result of an **`&&`** operation is **`true`** if, and only if, both the left side and the right side are **`true`**. Here's an example:

```
int  hasCar, hasTimeToGiveRide;

hasCar = true;
hasTimeToGiveRide = true;

if ( hasCar && hasTimeToGiveRide )
 printf( "Hop in - I'll give you a ride!\n" );
else
 printf( "I've either got no car, no time, or
 neither!\n" );
```

This example uses two variables. One indicates whether the program has a car, the other whether the program has time to give us a ride to the mall. All philosophical issues aside (can a program have a car?), the question of the moment is, which of the two **`printf()`**'s will fire? Since both sides of the **`&&`** were set to **`true`**, the first **`printf()`** will be called. If either one (or both) of the variables were set to **`false`**, the second **`printf()`** would be called. Another way to think of this is that we'll only get a ride to the mall if our friendly program has a car *and* has time to give us a ride. If either of these is not true, we're not getting a ride. By the way, notice the use here of the second form of **`if`**, the **`if-else`** statement.

The `||` operator is commonly referred to as the **`or`** operator. The result of a `||` operation is **`true`** if either the left side or the right side, or both sides, of the `||` are **`true`**. Put another way, the result of a `||` is **`false`** if, and only if, both the left side and the right side of the `||` are **`false`**. Here's an example:

```
int  nothingElseOn, newEpisode;

nothingElseOn = true;
newEpisode = true;

if ( newEpisode || nothingElseOn )
 printf( "Let's watch Star Trek!\n" );
else
 printf( "Something else is on or I've seen
 this one.\n" );
```

This example uses two variables to decide whether or not we should watch Star Trek (your choice - TOS, TNG, DS9, STV, or STE). One variable indicates whether anything else is on right now, and the other tells you whether this episode is a rerun. If this is a brand new episode, *or* if nothing else is on, we'll watch Star Trek.

Here's a slight twist on the previous example:

```
int  nothingElseOn, itsARerun;

nothingElseOn = true;
itsARerun = false;

if ( (! itsARerun) || nothingElseOn )
 printf( "Let's watch Star Trek!\n" );
else
 printf( "Something else is on or I've seen
 this one.\n" );
```

This time, we've replaced the variable **`newEpisode`** with its exact opposite, **`itsARerun`**. Look at the logic that drives the **`if`** statement. Now we're combining **`itsARerun`** with the **`!`** operator. Before, we cared whether the episode was a **`newEpisode`**. This time we are concerned that the episode is not a rerun. See the difference?

truthTester.xcode

Both the **`&&`** and the `||` operators are summarized in the table in Figure 6.3. If you look in the folder *`Learn C Projects`*, you'll find a subfolder named *06.01 - truthTester*. *main.c* contains the three examples we just went through. Take some time to play with the code. Take turns changing the variables from **`true`** to **`false`** and back again. Use this code to get a good feel for the **`!`**, **`&&`**, and `||` operators.

You might also try commenting out the line **`#include <c.h>`** towards the top of the file. To do this, just insert the characters **`//`** at the very beginning of the line. When you compile, you'll get an error telling you that "true is undeclared". Worth remembering this! As you write your own programs, be sure to **`#include <c.h>`** if you want to use **`true`** and **`false`**.

A	B	A && B	A ll B
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Figure 6.3 *Truth table for the* **`&&`** *and* `||` *operators.*

On most keyboards, you type an **`&`** character by holding down the shift key and typing a 7. You type a | character by holding down the shift key and typing a \ (backslash). Don't confuse the | with the letter **`l`**, **`i`**, or with the **`!`** character.

Compound Expressions

All of the examples presented so far have consisted of relatively simple expressions. Here's an example that combines several different operators:

```
int  myInt;

myInt = 7;

if ( (myInt >= 1) && (myInt <= 10) )
 printf( "myInt is between 1 and 10" );
else
 printf( "myInt is not between 1 and 10" );
```

This example tests whether a variable is in the range between 1 and 10. The key here is the expression:

```
(myInt >= 1) && (myInt <= 10)
```

that lies between the **`if`** statement's parentheses. This expression uses the **`&&`** operator to combine two smaller expressions. Notice that the two smaller expressions were each surrounded by parentheses to avoid any ambiguity. If we left out the parentheses, like so:

```
myInt >= 1 && myInt <= 10
```

the expression might not be interpreted as we intended. Once again, use parentheses for safe computing.

Statements

At the beginning of the chapter, we defined the **`if`** statement as:

```
if ( expression )
 statement
```

We've covered expressions pretty thoroughly. Now, we'll turn our attention to the statement.

At this point in the book, you probably have a pretty intuitive model of the statement. You'd probably agree that this:

```
myInt = 7;
```

is a statement. But is this:

```
if ( isCold )
 printf( "Put on your sweater!" );
```

one statement or two? Actually, the previous code fragment is a statement within another statement. The **`printf()`** is one statement, residing within a larger statement, the **`if`** statement.

The ability to break your code out into individual statements is not a critical skill. Getting your code to compile, however, *is* critical. As new types of statements are introduced (like the **`if`** and **`if-`**

`else` introduced in this chapter) pay attention to the statement syntax. And pay special attention to the examples. Where do the semicolons go? What distinguishes this type of statement from all other types?

As you build up your repertoire of statement types, you'll find yourself using one type of statement within another. That's perfectly acceptable in C. In fact, every time you create an **`if`** statement, you'll use at least two statements, one within the other. Take a look at this example:

```
if ( myVar >= 1 )
 if ( myVar <= 10 )
    printf( "myVar is between 1 and 10" );
```

This example used an **`if`** statement as the statement for another **`if`** statement. This example calls the **`printf()`** if both **`if`** expressions are **`true`**; that is, if **`myVar`** is greater than or equal to 1 and less than or equal to 10. You could have accomplished the same result with this piece of code:

```
if ( ( myVar >= 1 ) && ( myVar <= 10 ) )
    printf( "myVar is between 1 and 10" );
```

The second piece of code is a little easier to read. There are times, however, when the method demonstrated in the first piece of code is preferred. Take a look at this example:

```
if ( myVar != 0 )
 if ( ( 1 / myVar ) < 1 )
    printf( "myVar is in range" );
```

One thing you don't want to do in C is divide a number by 0. Any number divided by zero is infinity, and infinity is a foreign concept to the C language. If your program ever tries to divide a number by 0, your program is likely to crash. The first expression in this example tests to make sure **`myVar`** is not equal to zero. If **`myVar`** is equal to zero, the second expression won't even be evaluated! The sole purpose of the first **`if`** is to make sure the second **`if`** never tries to divide by zero. Make sure you understand this point. Imagine what would happen if we wrote the code this way:

```
if ( (myVar != 0) && ((1 / myVar) < 1) )
    printf( "myVar is in range" );
```

As it turns out, if the left half of the **`&&`** operator evaluates to **`false`**, the right half of the expression will never be evaluated and the entire expression will evaluate to **`false`**. Why? Because if the left operand is **`false`**, it doesn't matter what the right operand is – **`true`** or **`false`**, the expression will evaluate to **`false`**. Be aware of this as you construct your expressions.

The Curly Braces { }

Earlier in the book, you learned about the curly braces that surround the body of every function. These braces also play an important role in statement construction. Just as parentheses can be used to group terms of an expression together, curly braces can be used to group multiple statements together. Here's an example:

```
onYourBack = TRUE;

if ( onYourBack )
{
 printf( "Flipping over" );
 onYourBack = FALSE;
}
```

In the example, if **onYourBack** is **true**, both of the statements in curly braces will be executed. A pair of curly braces can be used to combine any number of statements into a single super-statement, also known as a **block**. You can use this technique anywhere a statement is called for.

Curly braces can be used to organize your code, much as you'd use parentheses to ensure that an expression is evaluated properly. This concept is especially appropriate when dealing with nested statements. Consider this code, for example:

```
if ( myInt >= 0 )
 if ( myInt <= 10 )
    printf( "myInt is between 0 and 10.\n" );
else
 printf( "myInt is negative.\n" ); /* <---
 Error!!! */
```

Do you see the problem with this code? Which **if** does the **else** belong to? As written (and as formatted), the **else** looks like it belongs to the first **if**. That is, if **myInt** is greater than or equal to 0, the second **if** is executed, otherwise the second **printf()** is executed. Is this right?

Nope. As it turns out, an **else** belongs to the **if** closest to it (the second **if**, in this case). Here's a slight rewrite:

```
if ( myInt >= 0 )
 if ( myInt <= 10 )
    printf( "myInt is between 0 and 10.\n" );
 else
    printf( "myInt is not between 0 and 10.\
 n" );
```

One point here is that formatting is nice, but it won't fool the compiler. More importantly, this example shows how easy it is to make a mistake. Check out this version of the code:

```
if ( myInt >= 0 )
{
 if ( myInt <= 10 )
    printf( "myInt is between 0 and 10.\n" );
```

```
  }
  else
   printf( "myInt is negative.\n" );
```

Do you see how the curly braces help? In a sense, they act to hide the second **`if`** inside the first **`if`** statement. There is no chance for the **`else`** to connect to the hidden **`if`**.

No one I know ever got fired for using too many parentheses or too many curly braces.

Where to Place the Semicolon

So far, the statements we've seen fall into two categories. Function calls, such as calls to **`printf()`**, and assignment statements are called **simple statements**. Always place a semicolon at the end of a simple statement, even if it is broken over several lines, like this:

```
  printf( "%d%d%d%d", var1,
                      var2,
                      var3,
                      var4 );
```

Statements made up of several parts, including, possibly, other statements, are called **compound statements**. Compound statements obey some pretty strict rules of syntax. The **`if`** statement, for example, always looks like this:

```
  if ( expression )
   statement
```

Notice there are no semicolons in this definition. The statement part of the **`if`** can be a simple statement or a compound statement. If the statement is simple, follow the semicolon rules for simple statements and place a semicolon at the end of the statement. If the statement is compound, follow the semicolon rules for that particular type of statement.

Notice that using "curlies" to build a super-statement or block out of smaller statements does not require the addition of a semicolon.

The Loneliest Statement

Guess what? A single semicolon qualifies as a statement, albeit a somewhat lonely one. For example, this code fragment:

```
  if ( bored )
   ;
```

is a legitimate (and thoroughly useless) **`if`** statement. If **`bored`** is **`true`**, the semicolon statement gets executed. The semicolon by itself doesn't do anything but fill the bill where a statement is needed. There are times where the semicolon by itself is exactly what you need.

The while Statement

The **`if`** statement uses the value of an expression to decide whether to execute or skip over a statement. If the statement is executed, it is executed just once. Another type of statement, the **`while`** statement, repeatedly executes a statement as long as a specified expression is **`true`**. The **`while`** statement follows this pattern:

```
while ( expression )
 statement
```

The **`while`** statement is also known as the **while loop**, because once the statement is executed, the **`while`** loops back to reevaluate the expression. Here's an example of the **`while`** loop in action:

```
int  i;

i=0;

while ( ++i < 3 )
 printf( "Looping: %d\n", i );

printf( "We are past the while loop." );
```

This example starts by declaring a variable, **`i`**, to be of type **`int`**. **`i`** is then initialized to 0. Next comes the **`while`** loop. The first thing the **`while`** loop does is evaluate its expression. The **`while`** loop's expression is:

```
++i < 3
```

Before this expression is evaluated, **`i`** has a value of 0. The prefix notation used in the expression (**`++i`**) increments the value of **`i`** to 1 before the remainder of the expression is evaluated. The evaluation of the expression results in **`true`** since 1 is less than 3. Since the expression is **`true`**, the **`while`** loop's statement, a single **`printf()`** is executed. Here's the output after the first pass through the loop:

```
Looping: 1
```

Next, the **`while`** loops back and reevaluates its expression. Once again, the prefix notation increments **`i`**, this time to a value of 2. Since 2 is less than 3, the expression evaluates to **`true`**, and the **`printf()`** is executed again. Here's the output after the second pass through the loop:

```
Looping: 1
Looping: 2
```

Once the second **`printf()`** completes, it's back to the top of the loop to reevaluate the expression. Will this never end? Once again, **`i`** is incremented, this time to a value of 3. Aha! This time, the expression evaluates to **`false`**, since 3 is not less than 3. Once the expression evaluates to **`false`**, the **`while`** loop ends and control passes to the next statement, the second **`printf()`** in our example:

```
printf( "We are past the while loop." );
```

The **`while`** loop was driven by three factors: **initialization**, **modification,** and **termination**. Initialization is any code that affects the loop, but occurs before the loop is entered. In our example, the critical initialization occurred when the variable **`i`** was set to 0.

Frequently, you'll use a variable in a loop that changes value each time through the loop. In our example, the variable **`i`** was incremented by 1 each time through the loop. The first time through the loop, **`i`** had a value of 1. The second time, **`i`** had a value of 2. Variables that maintain a value based on the number of times through a loop are known as **counters**.

In the interest of clarity, some programmers use names like **`counter`**, or **`loopCounter`**. The nice thing about names like **`i`**, **`j`**, and **`k`** is that they don't get in the way, they don't take up a lot of space on the line. On the other hand, your goal should be to make your code as readable as possible, so it would seem that a name like **`counter`** would be better than the uninformative **`i`**, **`j`**, or **`k`**.

Once again, pick a style you are comfortable with and stick with it!

Modification is any code within the loop that changes the value of the loop's expression. In our example, the modification occurred within the expression itself when the counter, **`i`**, was incremented.

Termination is any condition that causes the loop to terminate. In our example, termination occurs when the expression has a value of **`false`**. This occurs when the counter, **`i`**, has a value that is not less than 3. Take a look at this example:

```
int  i;

i=1;

while ( i < 3 )
{
 printf( "Looping: %d\n", i );
 i++;
}

printf( "We are past the while loop." );
```

This example produces the same results as the previous example. This time, however, the initialization and modification conditions have changed slightly. In this example, **`i`** starts with a value of 1 instead of 0. In the previous example, the **`++`** operator was used to increment **`i`** at the very *top of the loop*. This example modifies **`i`** at the *bottom of the loop*.

Both of these examples show different ways to accomplish the same end. The phrase, "There's more than one way to eat an Oreo," sums up the situation perfectly. There will always be more than one solution to any programming problem. Don't be afraid to do things your own way. Just make sure your code works properly and is easy to read.

The for Statement

Nestled inside the C toolbox, right next to the **`while`** statement, is the **`for`** statement. The **`for`** statement is similar to the **`while`** statement, following the basic model of initialization, modification, and termination. Here's the pattern for a **`for`** statement:

```
for ( expression1 ; expression2 ; expression3
 )
 statement
```

The first expression represents the **`for`** statement's initialization. Typically, this expression consists of an assignment statement, setting the initial value of a counter variable. This first expression is evaluated once, at the beginning of the loop.

The second expression is identical in function to the expression in a **`while`** statement, providing the termination condition for the loop. This expression is evaluated each time through the loop, before the statement is executed.

Finally, the third expression provides the modification portion of the **`for`** statement. This expression is evaluated at the bottom of the loop, immediately following execution of the statement.

All three of these expressions are optional and may be left out entirely. For example, here's a **`for`** loop that leaves out all three expressions:

```
for ( ; ; )

DoSomethingForever();
```

Since this loop has no terminating expression, it is known as an **infinite loop**. Infinite loops are generally considered bad form and should be avoided like the plague!

The **`for`** loop can also be described in terms of a **`while`** loop:

```
expression1;
while ( expression2 )
{
 statement
 expression3;
}
```

Since you can always rewrite a **`for`** loop as a **`while`** loop, why introduce the **`for`** loop at all? Sometimes, a programming idea fits more naturally into the pattern of a **`for`** statement. If the **`for`** loop makes for more readable code, why not use it? As you write more and more code, you'll develop a sense for when to use the **`while`** and when to use the **`for`**.

Here's an example of a **`for`** loop:

```
int  i;

for ( i = 1; i < 3; i++ )
 printf( "Looping: %d\n", i );

printf( "We are past the for loop." );
```

This example is identical in functionality to the **`while`** loops presented earlier. Note the three expressions on the first line of the **`for`** loop. Before the loop is entered, the first expression is evaluated (remember, assignment statements make great expressions):

```
i = 1
```

Once the expression is evaluated, **`i`** has a value of 1. We are now ready to enter the loop. At the top of each pass through the loop, the second expression is evaluated:

```
i < 3
```

If the expression evaluates to **`true`**, the loop continues. Since **`i`** is less than 3, we can proceed. Next, the statement is executed:

```
printf( "Looping: %d\n", i );
```

Here's the first line of output:

```
Looping: 1
```

Having reached the bottom of the loop, the **`for`** evaluates its third expression:

```
i++
```

This changes the value of **`i`** to 2. Back to the top of the loop. Evaluate the termination expression:

```
i < 3
```

Since **`i`** is still less than 3, the loop continues. Once again, the **`printf()`** does its thing. The console window looks like this:

```
Looping: 1
Looping: 2
```

Next, the **`for`** evaluates **`expression3`**:

```
i++
```

incrementing the value of **`i`** to 3. Back to the top of the loop. Evaluate the termination expression:

```
i < 3
```

Lo and behold! Since **`i`** is no longer less than 3, the loop ends and the second **`printf()`** in our example is executed:

```
printf( "We are past the for loop." );
```

As was the case with **`while`**, **`for`** can take full advantage of a pair of curly braces:

```
for ( i = 0; i < 10; i++ )
{
 DoThis();
 DoThat();
 DanceALittleJig();
}
```

In addition, both **`while`** and **`for`** can take advantage of the loneliest statement, the lone semicolon. This example:

```
for ( i = 0; i < 1000; i++ )
 ;
```

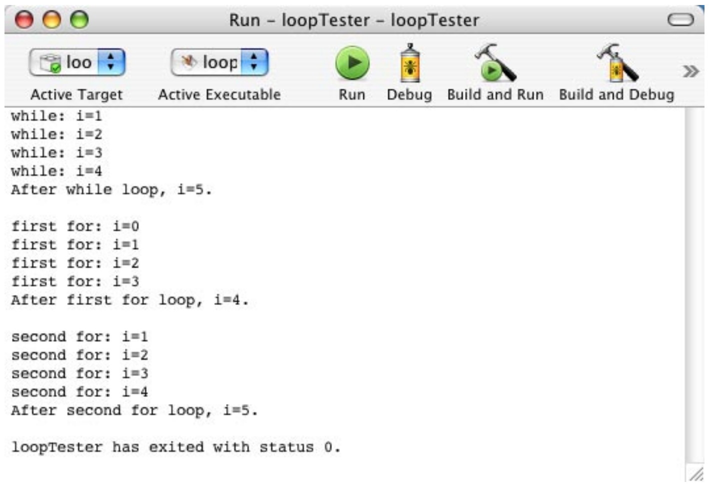
does nothing 1,000 times. Actually, the example does take some time to execute. The initialization expression is evaluated once, and the modification and termination expressions are each evaluated 1,000 times. Here's a **`while`** version of the loneliest loop:

```
i = 0;

while ( i++ < 1000 )
 ;
```

Some compilers will eliminate this loop and just set **`i`** to its terminating value (the value it would have if the loop executed normally). This is known as **code optimization**. The nice thing about code optimization is that it can make your code run faster and more efficiently. The down side is that an optimization pass on your code can sometimes have unwanted side-effects, like eliminating the **`while`** loop just discussed. It's a good idea to get to know your compiler's optimization capabilities and tendencies. Read the documentation!

loopTester.xcode

Interestingly, there is an important difference between the **`for`** and **`while`** loops you just saw. Take a minute to look back and try to predict the value of **`i`** the first time through each loop and after each loop terminates. Were the results the same for the while and for loops? Hmmm... You might want to take another look. Here's a sample program that should clarify the difference between these two loops. Look in the folder *Learn C Projects*, inside the subfolder named *06.02 - loopTester*, and open the project *loopTester.xcode*. *main.c* implements a **`while`** loop and two slightly different **`for`** loops. Run the project. Your output should look like that shown in Figure 6.4.



```
while: i=1
while: i=2
while: i=3
while: i=4
After while loop, i=5.

first for: i=0
first for: i=1
first for: i=2
first for: i=3
After first for loop, i=4.

second for: i=1
second for: i=2
second for: i=3
second for: i=4
After second for loop, i=5.

loopTester has exited with status 0.
```

Figure 6.4 *The output from* **`loopTester`**, *showing the output from 3 different loops.*

`loopTester` starts off with the standard **`#include`**. **`main()`** defines a counter variable, **`i`**, sets **`i`** to 0, then enters a **`while`** loop:

```
    while ( i++ < 4 )
        printf( "while: i=%d\n", i );
```

The loop executes 4 times, resulting in this output:

```
while: i=1
while: i=2
while: i=3
while: i=4
```

Do you see why? If not, go through the loop yourself, calculating the value for **`i`** each time through the loop. Remember, since we are using postfix notation (**`i++`**), **`i`** gets incremented *after* the test is made to see if it is less than 4. The test and the increment happen at the top of the loop, before the loop is entered.

Once the loop completes, we print the value of **`i`** again:

```
printf( "After while loop, i=%d.\n\n", i );
```

Here's the result:

```
After while loop, i=5.
```

Here's how we got that value. The last time through the loop (with **`i`** equal to 4), we go back to the top of the **`while`** loop, test to see if **`i`** is less than 4 (it no longer is), then do the increment of **`i`**, bumping it from 4 to 5.

OK, one loop down, two to go. This next loop looks like it should accomplish the same thing. The difference is, we don't do the increment of **`i`** till the bottom of the loop, till we've been through the loop once already.

```
for ( i = 0; i < 4; i++ )
    printf( "first for: i=%d\n", i );
```

As you can see by the output, **`I`**

ranges from 0 to 3 instead of from 1 to 4.

```
first for: i=0
first for: i=1
first for: i=2
first for: i=3
```

Once we drop out of the **`for`** loop, we once again print the value of **`i`**:

```
printf( "After first for loop, i=%d.\n\n", i
);
```

Here's the result:

```
After first for loop, i=4.
```

As you can see, the **`while`** loop ranged **`i`** from 1 to 4, leaving **`i`** with a value of 5 at the end of the loop. The **`for`** loop ranged **`i`** from 0 to 3, leaving **`i`** with a value of 4 at the end of the loop. So how do we fix the **`for`** loop so it works the same way as the **`while`** loop? Take a look:

```
for ( i = 1; i <= 4; i++ )
    printf( "second for: i=%d\n", i );
```

This **for** loop started **i** at 1 instead of 0. It tests to see if **i** is *less than or equal to* 4 instead of just less than 4. We could also have used the terminating expression **i < 5** instead. Either one will work. As proof, here's the output from this loop:

```
second for: i=1
second for: i=2
second for: i=3
second for: i=4
```

Once again, we print the value of **i** at the end of the loop:

```
 printf( "After second for loop, i=%d.\n", i
 );

 return 0;
}
```

Here's the last piece of output:

```
After second for loop, i=5.
```

This second **for** loop is the functional equivalent to the **while** loop. Take some time to play with this code. You might try to modify the **while** loop to match the first **for** loop.

By far, the **while** and **for** statements are the most common types of C loops. For completeness, however, we'll cover the remaining loop, a little-used gem called the **do** statement.

The do Statement

The **do** statement is a **while** statement that evaluates its expression at the bottom of its loop, instead of at the top. Here's the pattern a **do** statement must match:

```
do
 statement
while ( expression ) ;
```

Here's a sample:

```
i = 1;

do
{
 printf( "%d\n", i );
 i++;
}
while ( i < 3 );

printf( "We are past the do loop." );
```

The first time through the loop, **i** has a value of 1. The **printf()** prints a 1 in the console window, then the value of **i** is bumped to 2. It's not until this point that the expression **(i < 3)** is evaluated.

Since 2 is less than 3, a second pass through the loop occurs.

During this second pass, the **`printf()`** prints a 2 in the console window, then the value of **`i`** is bumped to 3. Once again, the expression **`( i < 3 )`** is evaluated. Since 3 is not less than 3, we drop out of the loop to the second **`printf()`**.

The important thing to remember about **`do`** loops is this: Since the expression is not evaluated until the bottom of the loop, the body of the loop (the statement) is always executed at least once. Since **`for`** and **`while`** loops both check their expressions at the top of the loop, it's possible for either to drop out of the loop before the body of the loop is executed.

Let's move on to a completely different type of statement, known as the switch.

The switch

The **`switch`** statement uses the value of an expression to determine which of a series of statements to execute. Here's an example that should make this concept a little clearer:

```
switch ( theYear )
{
 case 1066:
    printf( "Battle of Hastings" );
    break;
 case 1492:
    printf( "Columbus sailed the ocean blue"
 );
    break;
 case 1776:
    printf( "Declaration of Independence\n"
 );
    printf( "A very important document!!!" );
    break;
 default:
    printf( "Don't know what happened during
 this year" );
}
```

The **`switch`** is constructed of a series of **`case`**s, each based on a specific value of **`theYear`**. If **`theYear`** has a value of 1066, execution continues with the statement following that case's colon, in this case, the line:

```
printf( "Battle of Hastings" );
```

Execution continues, line after line, until either the bottom of the **`switch`** (the right curly-brace) or a **`break`** statement is reached. In this case, the next line is a **`break`** statement.

The **`break`** statement comes in handy when you are working with **`switch`**es and loops. The **`break`** tells the computer to jump immediately to the next statement after the end of the loop or **`switch`**.

Continuing with the example, if **`theYear`** has a value of 1492, the **`switch`** jumps to the lines:

```
printf( "Columbus sailed the ocean blue" );
break;
```

A value of 1776 jumps to the lines:

```
printf( "Declaration of Independence\n" );
printf( "A very important document!!!" );
break;
```

Notice that this **`case`** has two statements before the break. There is no limit to the number of statements a **`case`** can have. One is OK, 653 is OK. You can even have a **`case`** with no statements at all.

The original example also contains a **`default case`**. If the **`switch`** can't find a **`case`** that matches the value of its expression, the **`switch`** looks for a **`case`** labeled **`default`**. If the **`default`** is present, its statements are executed. If no **`default`** is present, the **`switch`** completes without executing any of its statements.

Here's the pattern the **`switch`** tries to match:

```
switch ( expression )
{
 case constant:
    statements
 case constant:
    statements
 default:
    statements
}
```

Why would you want a **`case`** with no statements? Here's an example:

```
switch ( myVar )

{

case 1:

case 2:

DoSomething();

break;

case 3:

DoSomethingElse();

}
```

In this example, if **`myVar`** has a value of 1 or 2, the function **`DoSomething()`** is called. If **`myVar`** has a value of 3, the function **`DoSomethingElse()`** is called. If **`myVar`** has any other value, nothing happens. Use a **`case`** with no statements when you want two different **`case`**s to execute the same statements.

Think about what happens with this example:

```
switch ( myVar )

{

case 1:

DoSometimes();

case 2:

DoFrequently();

default:

DoAlways();

}
```

If **`myVar`** is 1, all three functions will get called. If **`myVar`** is 2, **`DoFrequently()`** and **`DoAlways()`** will get called. If **`myVar`** has any other value, **`DoAlways()`** gets called by itself. This is a good example of a **`switch`** without **`break`**s.

At the heart of each **`switch`** is its expression. Most **`switch`**es are based on single variables but, as we mentioned earlier, assignment statements make perfectly acceptable expressions.

Each **`case`** is based on a **constant**. Numbers (like 47 or -12,932) are valid constants. Variables, such as **`myVar`**, are not. As you'll see later, single-byte characters (like **`'a'`** or **`'\n'`**) are also valid constants. Multiple-byte character strings (like **`"Gummy-bear"`**) are not.

If your **`switch`** uses a **`default case`**, make sure you use it as shown in the pattern above. Don't include the word **`case`** before the word **`default`**.

Breaks in Other Loops

The **`break`** statement has other uses besides the **`switch`** statement. Here's an example of a **`break`** used in a **`while`** loop:

```
i=1;

while ( i <= 9 )
{
 PlayAnInning( i );
 if ( ItIsRaining() )
    break;
 i++;
}
```

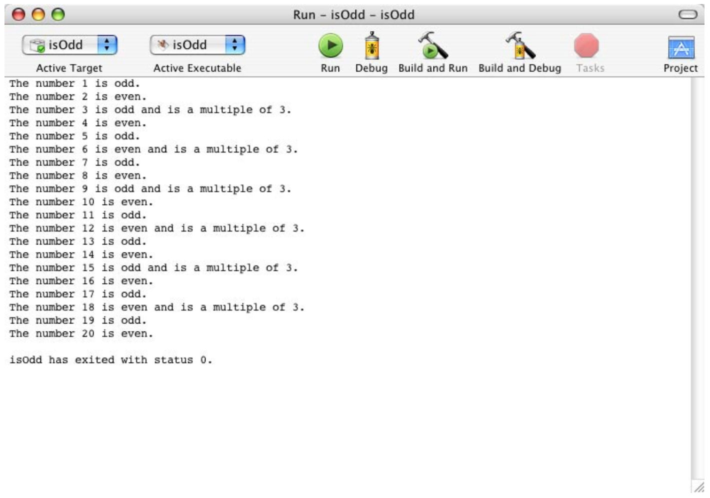
This sample tries to play nine innings of baseball. As long as the function **`ItIsRaining()`** returns with a value of **`false`**, the game continues uninterrupted. If **`ItsRaining()`** returns a value of **`true`**, the **`break`** statement is executed and the program drops out of the loop, interrupting the game.

The **`break`** statement allows you to construct loops that depend on multiple factors. The termination of the loop depends on the value of the expression found at the top of the loop, as well as on any outside factors that might trigger an unexpected **`break`**.

isOdd.xcode

This next program combines **`for`** and **`if`** statements to tell you whether the number 1 through 20 are odd or even, and if they are an even multiple of 3. It also introduces a brand new operator: the **`%`** operator. Go into the *Learn C Projects* folder, into the *06.03 - isOdd* subfolder, and open the project *isOdd.xcode*.

Run *isOdd.xcode*. You should see something like the console window shown in Figure 6.5. You should see a line for each number from 1 through 20. Each of the numbers will be described as either odd or even. Each of the multiples of 3 will have additional text describing them as such. Here's how the program works:



```
The number 1 is odd.
The number 2 is even.
The number 3 is odd and is a multiple of 3.
The number 4 is even.
The number 5 is odd.
The number 6 is even and is a multiple of 3.
The number 7 is odd.
The number 8 is even.
The number 9 is odd and is a multiple of 3.
The number 10 is even.
The number 11 is odd.
The number 12 is even and is a multiple of 3.
The number 13 is odd.
The number 14 is even.
The number 15 is odd and is a multiple of 3.
The number 16 is even.
The number 17 is odd.
The number 18 is even and is a multiple of 3.
The number 19 is odd.
The number 20 is even.

isOdd has exited with status 0.
```

Figure 6.5 *Running* **`isOdd`**.

Stepping Through the Source Code

main.c starts off with the usual **`#include`** and the beginning of **`main()`**. **`main()`** starts off by declaring a counter variable named **`i`**.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int i;
```

Our goal here is to step through each of the numbers from 1 to 20. For each number, we want to check to see if the number is odd or even. We also want to check whether the number is evenly divisible by 3.

Once we've analyzed a number, we'll use **`printf()`** to print a description of the number in the console window.

As we mentioned in Chapter 4, the scheme that defines the way a program works is called the program's algorithm. It's a good idea to try to work out the details of your program's algorithm before writing even one line of source code.

As you might expect, the next step is to set up a **`for`** loop using **`i`** as a counter. **`i`** is initialized to 1. The loop will keep running as long as the value of **`i`** is less than or equal to 20. This is the same as saying the loop will exit as soon as the value of **`i`** is found to be greater than 20. Every time the loop reaches the bottom, the third expression, **`i++`**, will be evaluated, incrementing the value of **`i`** by 1. This is a classic **`for`** loop.

```
for ( i = 1; i <= 20; i++ )
{
```

Now we're inside the **`for`** loop. Our goal is to print a single line for each number (i.e., one line each time through the **`for`** loop). If you check back to Figure 6.4, you'll notice that each line starts with the phrase:

```
The number x is
```

where **`x`** is the number being described. That's the purpose of this first **`printf()`**:

```
printf( "The number %d is ", i );
```

Notice that this **`printf()`** wasn't part of an **`if`** statement. We want this **`printf()`** to print its message every time through the loop. The next sequence of **`printf()`**s are a different story altogether.

The next chunk of code determines whether **`i`** is even or odd, then uses **`printf()`** to print the appropriate word in the console window. Because the last **`printf()`** didn't end with a newline character (**`'\n'`**), the word "even" or "odd" will appear immediately following:

```
The number x is
```

on the *same line* in the console window.

This next chunk of code introduces a brand new operator. **`%`** is a binary operator that returns the remainder when the left operand is divided by the right operand. For example, **`i % 2`** divides **`2`** into **`i`** and returns the remainder. If **`i`** is even, this remainder will be 0. If **`i`** is odd, this remainder will be 1.

```
      if ( (i % 2) == 0 )
            printf( "even" );
      else
            printf( "odd" );
```

In the expression **`i % 3`**, the remainder will be 0 if **`i`** is evenly divisible by 3, and either 1 or 2 otherwise.

```
  if ( (i % 3) == 0 )
   printf( " and is a multiple of 3" );
```

If **`i`** is evenly divisible by 3, we'll add the phrase:

```
  " and is a multiple of 3"
```

to the end of the current line. Finally, we add a period and a newline **`".\n"`** to the end of the current line, placing us at the beginning of the next line of the console window.

```
   printf( ".\n" );
```

The loop ends with a curly brace. **`main()`** ends with our normal **`return`** and curly brace.

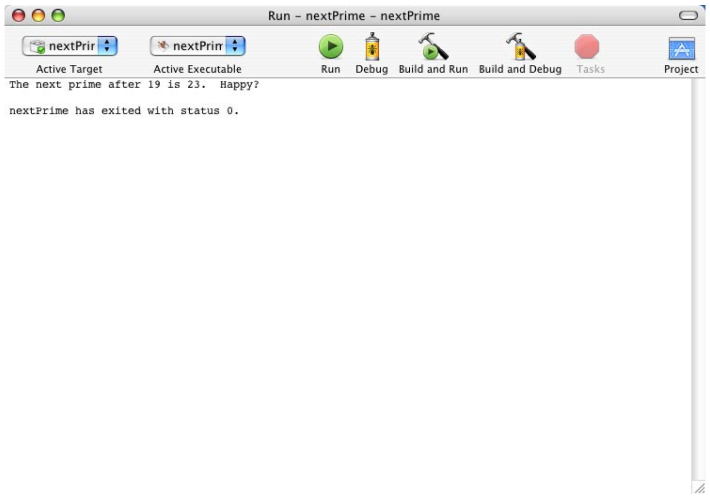
```
   }

   return 0;
  }
```

nextPrime.xcode

Our next program focuses on the mathematical concept of **prime numbers**. A prime number is any number whose only factors are 1 and itself. For example, 6 is not a prime number because its factors are 1, 2, 3, and 6. The number 5 is prime because its factors are limited to 1 and 5. The number 12 isn't prime — its factors are 1, 2, 3, 4, 6, and 12.

Our next program will find the next prime number greater than a specified number. For example, if we set our starting point to 14, the program would find the next prime, 17. We have the program set up to check for the next prime after 19. Know what that is?

Go into the folder *Learn C Projects*, into the subfolder *06.04 - nextPrime*, and open the project *nextPrime.xcode*. Run the project. You should see something like the console window shown in Figure 6.6. As you can see, the next prime number after 19 is (drum roll, please...) 23. Here's how the program works.



```
The next prime after 19 is 23.  Happy?

nextPrime has exited with status 0.
```

Figure 6.6 Running **`nextPrime`**.

Stepping Through the Source Code

In addition to our **`#include`** of **`<stdio.h>`** and **`<c.h>`** (the latter to include the definition of **`true`** and **`false`**), we've added a third **`#include`** to our stable. The new **`#include`**, **`<math.h>`**, gives us access to a series of math functions, most notably the function **`sqrt()`**. **`sqrt()`** takes a single parameter and returns the square root of that parameter. You'll see how this works in a minute.

```
#include <stdio.h>
#include <math.h>
#include <c.h>

int main (int argc, const char * argv[])
{
```

We're going to need a boatload of variables. They're all defined as **`int`**s:

```
	int		startingPoint, candidate, last, i;
	int		isPrime;
```

`startingPoint` is the number we want to start off with. We'll find the next prime after **`startingPoint`**. **`candidate`** is the current candidate we are considering. Is **`candidate`** the lowest prime number greater than **`startingPoint`**? By the time we are done, it will be!

```
	startingPoint = 19;
```

Since 2 is the lowest prime number, if **`startingPoint`** is less than 2, we know that the next prime is 2. By setting **`candidate`** to 2, our work is done.

```
		if ( startingPoint < 2 )
		{
			candidate = 2;
		}
```

If **`startingPoint`** is 2, the next prime is 3 and we'll set **`candidate`** accordingly.

```
else if ( startingPoint == 2 )
{
    candidate = 3;
}
```

If we got this far, we know that **`startingPoint`** is greater than 2. Since 2 is the only even prime number, and since we've already checked for **`startingPoint`** being equal to 2, we can now limit our search to odd numbers only. We'll start **`candidate`** at **`startingPoint`**, then make sure that **`candidate`** is odd. If not, we'll decrement **`candidate`**. Why decrement instead of increment? If you peek ahead a few lines, you'll see we're about to enter a **`do`** loop, and that we bump **`candidate`** to the next odd number at the *top* of the loop. By decrementing **`candidate`** now, we're preparing for the bump at the top of the loop, which will take **`candidate`** to the next odd number greater than **`startingPoint`**.

```
else
{
    candidate = startingPoint;

    if (candidate % 2 == 0)
        candidate--;
```

This loop will continue stepping through consecutive odd numbers until we find a prime number. We'll start **`isPrime`** off as **`true`**, then check the current **`candidate`** to see if we can find a factor. If we do find a factor, we'll set **`isPrime`** to **`false`**, forcing us to repeat the loop.

```
do
{
    isPrime = true;
    candidate += 2;
```

Now we'll check to see if **`candidate`** is prime. This means verifying that **`candidate`** has no factors other than 1 and **`candidate`**. To do this, we'll check the numbers from 3 to the square root of **`candidate`** to see if any of them divide evenly into **`candidate`**. If not, we know we've got ourselves a prime!

```
    last = sqrt( candidate );
```

So why don't we check from 2 up to **`candidate`** - 1? Why start with 3? Since **`candidate`** will never be even, we know that 2 will never be a factor. For the same reason, we know that no even number will ever be a factor.

Why stop at the square root of **`candidate`**? Good question! To help understand this approach, consider the factors of 12, other than 1 and 12. They are 2, 3, 4, and 6. The square root of 12 is approximately 3.46. Notice how this fits nicely in the middle of the list of factors. Each of the factors less than the square root will have a matching factor greater than the square root. In this case, 2 matches with 6 (2*6=12) and 3 matches with 4 (3*4=12). This will always be true. If we don't find a factor by the time we hit the square root, there won't be a factor, and the candidate is prime.

Take a look at the top of the **`for`** loop. We start **`i`** at 3. Each time we hit the top of the loop (including the first time through the loop) we'll check to make sure we haven't passed the square root of **`candidate`**, and that **`isPrime`** is still **`true`**. If **`isPrime`** is **`false`**, we can stop searching for a factor, since we've just found one! Finally, each time we complete the loop, we bump **`i`** to the next odd number.

```
for ( i = 3; (i <= last) && isPrime; i += 2 )
{
```

Each time through the loop, we'll check to see if **`i`** divides evenly into **`candidate`**. If so, we know it is a factor and we can set **`isPrime`** to **`false`**.

```
            if ( (candidate % i) == 0 )
               isPrime = false;
         }
      } while ( ! isPrime );
   }
```

Once we drop out of the **`do`** loop, we use **`printf()`** to print both the starting point and the first prime number greater than the starting point.

```
   printf( "The next prime after %d is %d.
      Happy?\n", startingPoint, candidate );
   return 0;
}
```

If you are interested in prime numbers, play around with this program. See if you can modify the code to print all the prime numbers from 1 to 100. How about the first 100 prime numbers?

What’s Next?

Congratulations! You’ve made it through some tough concepts. You’ve learned about the C statements that allow you to control your program’s flow. You’ve learned about C expressions and the concept of **`true`** and **`false`**. You’ve also learned about the logical operators based on the values **`true`** and **`false`**. You’ve learned about the **`if`**, **`if-else`**, **`for`**, **`while`**, **`do`**, **`switch`**, and **`break`** statements. In short, you’ve learned a lot!

Our next chapter introduces the concept of **pointers**.

A pointer to a variable is really the address of the variable in memory. If you pass the value of a variable to a function, the function can make use of the variable’s value, but can’t *change* the variable’s value. If you pass the address of the variable to the function, the function can also change the value of the variable. Chapter 7 will tell you why.

Chapter 7 will also discuss function parameters in detail. As usual, plenty of code fragments and sample applications will be presented to keep you busy. See you there.

Exercises

1) What's wrong with each of the following code fragments?

```
a.  if     i
           i++;

b.  for    ( i=0; i<20; i++ )
           i--;

c.  while  ( )
           i++;

d.  do     ( i++ )
           until ( i == 20 );

e.  switch ( i )
    {
           case "hello":
           case "goodbye":
                  printf( "Greetings." );
                  break;
           case default:
                  printf( "Boring." );
}

f.  if     ( i < 20 )
           if     ( i == 20 )
                  printf( "Lonely..." );

g.  while  ( done = TRUE )
           done = ! done;

h.  for    ( i=0; i<20; i*20 )
           printf( "Modification..." );
```

2) Modify **`nextPrime.c`** to compute the prime numbers from 1 to 100.

3) Modify **`nextPrime.c`** to compute the first 100 prime numbers.

Chapter 7 Pointers and Parameters

You've come a long way. You've mastered variable basics, operators, and statements. You're about to add some powerful, new concepts to your programming toolbox.

For starters, we'll introduce the concept of **pointers**, also known as variable addresses. From now on, you'll use pointers in almost every C program you write. Pointers allow you to implement complex data structures, opening up a world of programming possibilities.

What is a Pointer?

In programming, pointers are references to other things. When someone calls your name to get your attention, they're using your name as a pointer. Your name is one way people refer to you.

Your name and address can combine to serve as a pointer, telling the mail carrier where to deliver the new Sears catalog. Your address distinguishes your house from all the other houses in your neighborhood and your name distinguishes you from the rest of the people living in your house.

When you declare a variable in C, memory is allocated to the variable. This memory has an address. C pointers are special variables, specifically designed to hold one of these addresses. Later in the chapter, you'll learn how to create a pointer, how to make it point to a specific variable, and how to use the pointer to change the variable's value.

Why Use Pointers?

Pointers can be extremely useful, allowing you to access your data in ways that ordinary variables just don't allow. Here's a real-world example of "pointer flexibility."

When you go to the library in search of a specific title, chances are you start your search in a card catalog. Card catalogs contain thousands of index cards, one for every book in the library. Each index card contains information about a specific book, including such information as the author's name, the book's title, and the copyright date.

Most libraries have three card catalogs. Each lists all the books, sorted alphabetically by subject, author, or by title. In the subject card catalog, a book can be listed more than once. For example, a book about Thomas Jefferson might be listed under "Presidents, U.S.," "Architects," or even under "Inventors" (Jefferson was quite an inventor).

Figure 7.1 shows a catalog card for Albert Einstein's famous book on relativity, called *The Meaning of Relativity.* The card was listed in the subject catalog under the subject "RELATIVITY (PHYSICS)." Take a minute to look the card over. Pay special attention to the catalog information located on the left side of the card. The catalog number for this book is 530.1. This number tells you exactly where to find the book among all the other books on the shelves. The books are ordered numerically, so you'll find this book in the 500 shelves, between 530 and 531.

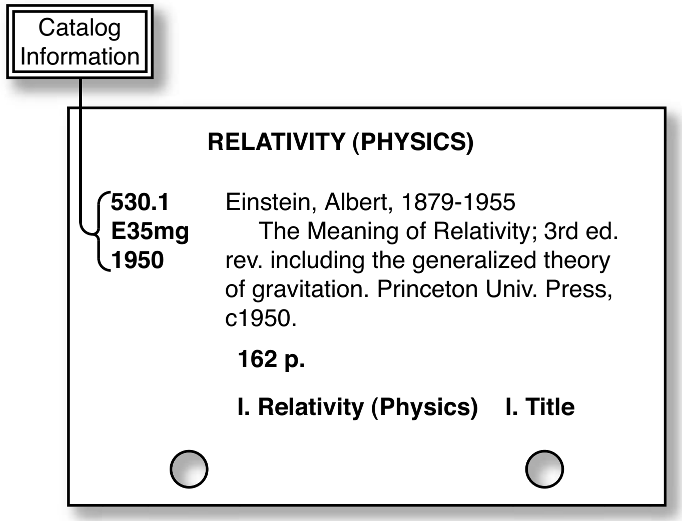

Catalog Information
RELATIVITY (PHYSICS)
530.1
E35mg
1950
Einstein, Albert, 1879-1955
The Meaning of Relativity; 3rd ed. rev. including the generalized theory of gravitation. Princeton Univ. Press, c1950.
162 p.
I. Relativity (Physics)
I. Title

Figure 7.1 *Catalog card for a rather famous book. Note the catalog information on the left side of the card.*

In this example, the library bookshelves are like your computer's memory, with the books acting as data. The catalog number is the address of your data (a book) in memory (on the shelf).

As you might have guessed, the catalog number acts as a pointer. The card catalogs use these pointers to rearrange all the books in the library, without moving a single book. Think about it. In the subject card catalog, all the books are arranged by subject. Physically, the book arrangements have nothing to do with subject. Physically, the books are arranged numerically, by catalog number. By adding a layer of pointers between you and the books, the librarians achieve an extra layer of flexibility.

In the same way, the author and title card catalogs use a layer of pointers to arrange all the books by author and by title. By using pointers, all the books in the library are arranged four different ways without ever leaving the shelves. The books are arranged physically (sorted by catalog number) and logically (sorted in one catalog by author, in another by subject, and in another by title). Without the support of a layer of pointers, these logical book arrangements would be impossible.

Adding a layer of pointers is also known as "adding a **level of indirection**." The number of levels of indirection is the number of pointers you have to use to get to your library book (or to your data).

Checking Out of the Library

So far, we've talked about pointers in terms of library catalog numbers. The use of pointers in your C programs is not much different from this model. Each card catalog number points out the location of a book on the library shelf. In the same way, each pointer in your program will point out the location of a piece of data in computer memory.

If you wrote a program to keep track of your compact-disc collection, you might maintain a list of pointers, each one of which might point to a block of data that describes a single CD. Each block of data might contain such info as the name of the artist, the name of the album, the year of release, and a category (jazz, rock, blues). If you got more ambitious, you could create several pointer lists. One list might sort your CDs alphabetically by artist name. Another might sort them chronologically by year of release. Yet another list might sort your CDs by musical category. You get the picture.

There's a lot you can do with pointers. By mastering the techniques presented in these next few chapters, you'll be able to create programs that take full advantage of pointers.

Our goal for this chapter is to master pointer basics. We'll talk about C pointers and C pointer operations. You'll learn how to create a pointer and how to make the pointer point to a variable. You'll also learn how to use a pointer to change the value of the variable the pointer points to.

Pointer Basics

Pointers are variable addresses. Instead of an address such as:

```
1313 Mockingbird Lane
Raven Heights, California  90263
```

a variable's address refers to a memory location within your computer. As we discussed in Chapter 3, your computer's memory, also known as **random access memory**, or **RAM**, consists of a sequence of bytes. One megabyte of RAM has exactly 2^{20} (or 1,048,576) bytes of memory. Eight megabytes of RAM has exactly $8 \times 2^{20} = 2^{23} = 8{,}388{,}608$ bytes of memory. One gigabyte of RAM has exactly 2^{30} bytes = 1,024 megabytes = 1,073,741,824 bytes of memory. Whew!

Every one of those bytes has its own unique address. Computer addresses typically start with 0 and continue up, one at a time, until they reach the highest address. The first byte has an address of 0, the next byte has an address of 1, and so on. Figure 7.2 shows the addressing scheme for a computer with a gigabyte of RAM. A gigabyte is 1,024 megabytes. Notice that the addresses run from 0 (the lowest address) all the way up to 1,073,741,823 (the highest address). The same scheme would hold true for ten gigabytes, or even one terabyte (1,024 gigabytes).

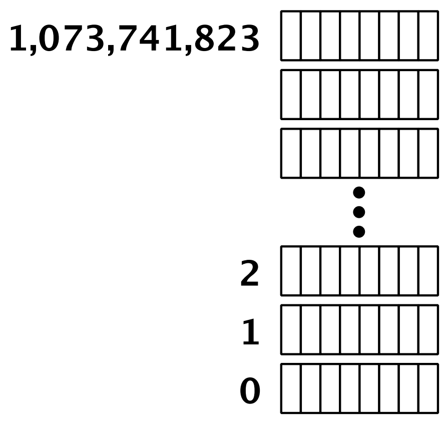


Figure 7.2 *A gigabyte worth of bytes.*

Variable Addresses

When you run a program, one of the first things the computer does is allocate memory for your program's variables. When you declare an **`int`** in your code, like this:

```
int  myVar;
```

the compiler reserves memory for the exclusive use of **`myVar`**.

As mentioned earlier in the book, the amount of memory allocated for an **`int`** depends on your development environment. Xcode defaults to using 4-byte **`int`**s.

Each of **`myVar`**'s bytes has a specific address. Figure 7.3 shows a one gigabyte chunk of memory with 4 bytes allocated to the variable **`myVar`**. In this picture, the 4 bytes allocated to **`myVar`** have the addresses 836, 837, 838, and 839.

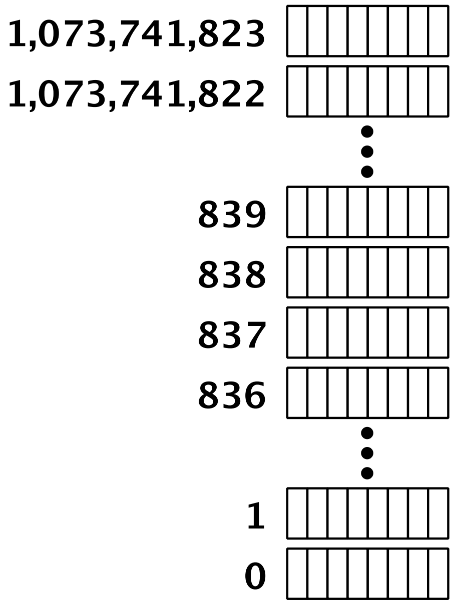


***Figure** 7.3 Four bytes allocated for the* **`int`** *named* **`myVar`**.

By convention, a variable's address is said to be the address of its first byte (the first byte is the byte with the lowest-numbered address). If a variable uses memory locations 836 through 839 (as **`myVar`** does), its address is 836 and its length is 4 bytes.

When more than 1 byte is allocated to a variable, the bytes will always be consecutive (next to each other in memory). You will never see an **`int`** whose byte addresses are 508, 509, 510, and 695. A variable's bytes are like family—they stick together!

As we showed earlier, a variable's address is a lot like the catalog number on a library catalog card. Both act as pointers, one to a book on the library shelf, and the other to a variable. From now on, when we use the term pointer with respect to a variable, we are referring to the variable's address.

Now that you understand what a pointer is, your next goal is to learn how to use pointers in your programs. The next few sections will teach you some valuable pointer-programming skills. You'll learn how to create a pointer to a variable. You'll also learn how to use that pointer to access the variable it points to.

The C language provides you with a few key tools to help you. These tools come in the form of two special operators: **`&`** and **`*`**.

The & Operator

The **`&`** operator (also called the **address-of operator**) pairs with a variable name to produce the variable's address. The expression:

```
&myVar
```

refers to **`myVar`**'s address in memory. If **`myVar`** owned memory locations 836 through 839 (as in Figure 7.3), the expression:

```
&myVar
```

would have a value of 836. The expression **`&myVar`** is a pointer to the variable **`myVar`**.

As you start programming with pointers, you'll find yourself using the **`&`** operator frequently. An expression like **`&myVar`** is a common way to represent a pointer. Another way to represent a pointer is with a **pointer variable**. A pointer variable is a variable specifically designed to hold the address of another variable.

Declaring a Pointer Variable

C supports a special notation for declaring pointer variables. This line:

```
int  *myPointer;
```

declares a variable called **`myPointer`**. Notice that the **`*`** is not part of the variable's name. Instead, it tells the compiler that the associated variable is a pointer, specifically designed to hold the address of an **`int`**. If there were a data type called **`bluto`**, you could declare a variable designed to point to a **`bluto`** like this:

```
bluto      *blutoPointer;
```

For now, we'll limit ourselves to pointers that point to **`int`**s. Look at this code:

```
int  *myPointer, myVar;

myPointer = &myVar;
```

The assignment statement puts **`myVar`**'s address in the variable **`myPointer`**. If **`myVar`**'s address is 836, this code will leave **`myPointer`** with a value of 836. Note that this code has absolutely no effect on the value of **`myVar`**.

There will be times in your coding when you have a pointer to a variable, but do not have access to the variable itself. This happens a lot. You can actually use the pointer to manipulate the value of the variable it points to. Observe:

```
int  *myPointer, myVar;

myPointer = &myVar;
*myPointer = 27;
```

As before, the first assignment statement places **`myVar`**'s address in the variable **`myPointer`**. The second assignment introduces the **`*`** operator. The **`*`** operator (called the **star** operator) converts a pointer variable to the item the pointer points to.

> The **`*`** that appears in the declaration statement isn't really an operator. It's only there to designate the variable **`myPointer`** as a pointer.

If **`myPointer`** points to **`myVar`**, as is the case in our example, **`*myPointer`** refers to the variable **`myVar`**. In this case, the line:

```
*myPointer = 27;
```

is the same as saying:

```
myVar = 27;
```

Confused? These memory pictures should help. Figure 7.4 joins our program in progress, just after the variables **`myVar`** and **`myPointer`** were declared:

```
int  *myPointer, myVar;
```

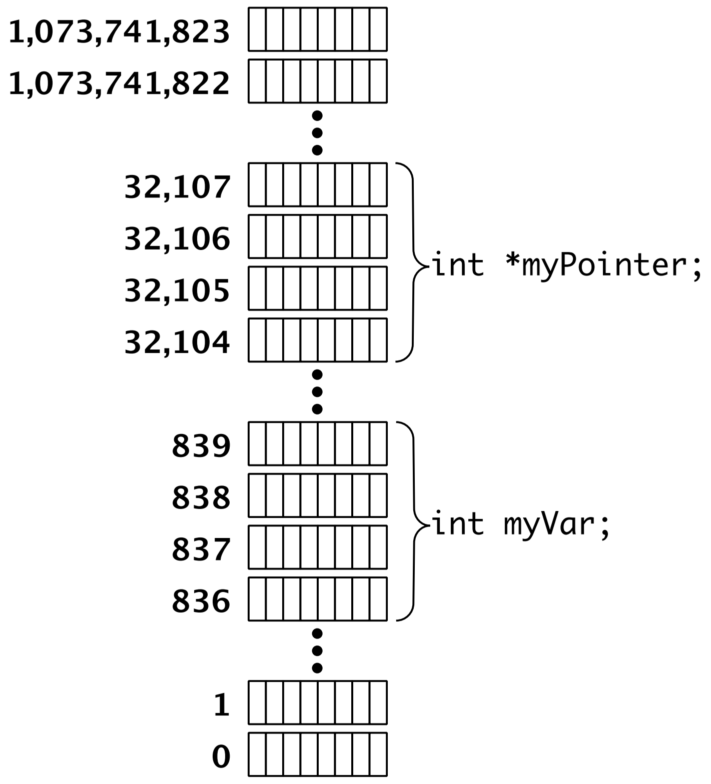


***Figure** 7.4 Memory allocated for* **`myVar`** *and* **`myPointer`**.

Notice that 4 bytes were allocated for the variable **`myVar`** and an additional 4 bytes were allocated for **`myPointer`**. Why? Because **`myVar`** is an **`int`** and **`myPointer`** is a pointer, designed to hold a 4-byte address.

Why a 4-byte address? Good question! 4 bytes is equal to 32 bits. Since memory addresses start at 0 and can never be negative, a 4-byte memory address can range from 0 up to $2^{32} - 1 = 4,294,967,295$. That means that a 32-bit computer can address a maximum of 4 gigabytes (4096 megabytes) of memory.

While 4 gigs of memory might seem more than adequate for most folks, there are already a number of applications that require more RAM than this. After all, it was just a few years ago that 32 megs of RAM was the standard. Soon, we will look back and wonder just how we managed to live with that pesky 4 gig limit!

So how do we address more than 4 gigabytes in a 32-bit computer? The short answer is, we don't. When Apple released the G5 back in 2003, they introduced their first 64-bit computer. Instead of a 4-byte address, the G5 supports an 8-byte address. An 8-byte address can hold values from 0 to $2^{64} - 1$. That is one giant number.

The point here is to be aware that the size of an address can change and the number of bytes used to represent an **`int`** can change.

Older computers (like the Apple IIe, for example) represented an address using 2 bytes (16-bits) of memory, yielding a range of addresses from 0 to $2^{16} - 1 = 65,535$. Imagine having to fit your operating system, as well as all your applications in a mere 64K of RAM (1K = 1024 bytes).

When the Mac first appeared, it came with 128K of RAM and used 24-bit memory addresses, yielding a range of addresses from 0 to $2^{24} - 1 = 16,777,215$ (also known as 16 megabytes). In those days, no one could imagine a computer that actually included 16 entire megabytes of memory!

Of course, these days we are much smarter. We absolutely know for a fact that we'll never exceed the need for 64-bit addresses. I mean, there's no way that a computer could ever make use of 4 gigabytes of RAM, right? Hmmm... Better not count on that. In fact, if you are a betting person, I'd wager that someday we'll see 16-byte addresses. Really!

Once memory is allocated for **`myVar`** and **`myPointer`**, we move on to the statement:

```
myPointer = &myVar;
```

The 4-byte address of the variable **`myVar`** is written to the 4 bytes allocated to **`myPointer`**. In our example, **`myVar`**'s address is 836. Figure 7.5 shows the value 836 stored in **`myPointer`**'s 4 bytes. Now **`myPointer`** is said to "point-to" **`myVar`**.

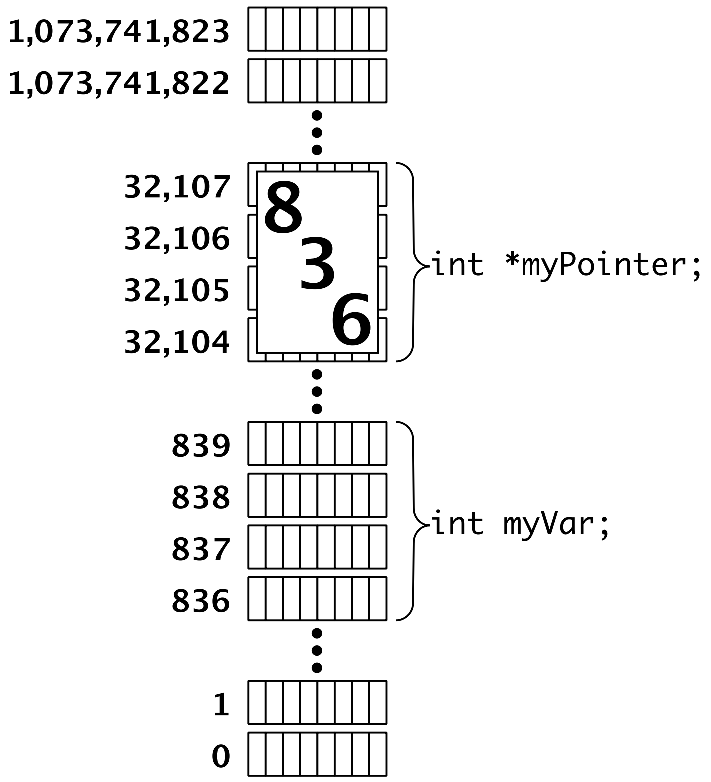


***Figure 7.5** The address of* **`myVar`** *is assigned to* **`myPointer`**.

OK, we're almost there. The next line of our example writes the value 27 to the location pointed to by **`myPointer`**.

```
*myPointer = 27;
```

Without the **`*`** operator, the computer would place the value 27 in the memory allocated to **`myPointer`**. The **`*`** operator **dereferences** **`myPointer`**. Dereferencing a pointer turns the pointer into the variable it points to. Figure 7.6 shows the end results.

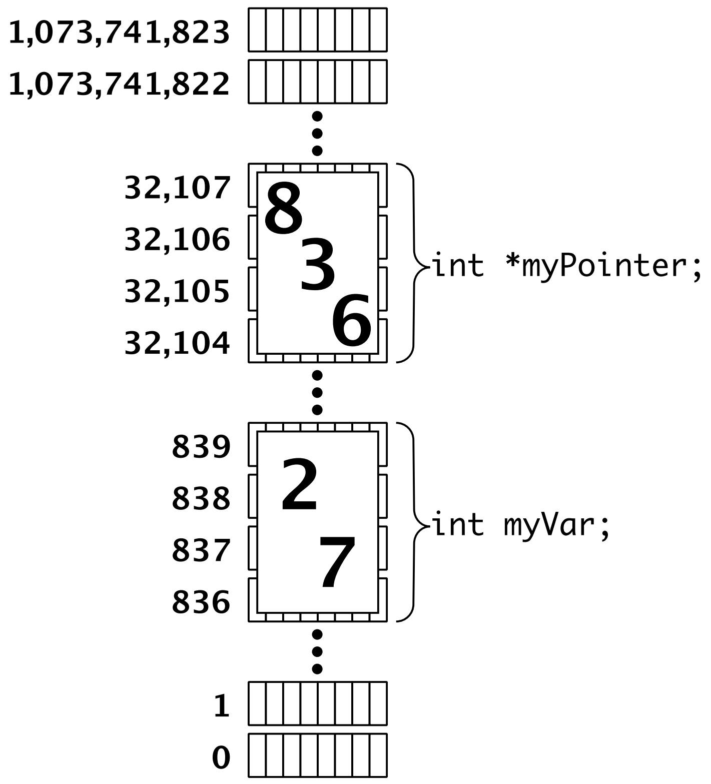


***Figure 7.6** Finally, the value 27 is assigned to* **`*myPointer`**.

If the concept of pointers seems alien to you, don't worry. You are not alone. Programming with pointers is one of the most difficult topics you'll ever take on. Just keep reading, and make sure you follow each of the examples line by line. By the end of the chapter, you'll be a pointer expert!

Function Parameters

One of the most important uses of pointers (and perhaps the easiest to understand) lies in the implementation of **function parameters.** In this section, we'll focus on parameters and, at the same time, have a chance to check out pointers in action.

What Are Function Parameters?

A function parameter is your chance to share a variable between a calling function and the called function.

Suppose you wanted to write a function called **`AddTwo()`** that took two numbers, added them together, and returned the sum of the two numbers. How would you get the two original numbers into **`AddTwo()`**? How would you get the sum of the two numbers back to the function that called **`AddTwo()`**?

As you might have guessed, the answer to both questions lies in the use of parameters. Before you can learn how to use parameters, however, you'll have to first understand the concept of **scope**.

Variable Scope

In C, every variable is said to have a scope, or range. A variable's scope defines where in the program you have access to a variable. In other words, if a variable is declared inside one function, can another function refer to that same variable?

C defines variable scope as follows:

- A variable declared inside a function is local to that function and may only be referenced inside that function.

This statement is important. It means you can't declare a variable inside one function, then refer to that same value inside another function. Here's an example that will never compile:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int numDots;

 numDots = 500;

 DrawDots();

 return 0;
}

void DrawDots( void )
{
 int i;

 for ( i = 1; i <= numDots; i++ )
     printf( "." );
}
```

The error in this code occurs when the function `DrawDots()` tries to reference the variable `numDots`. According to the rules of scope, `DrawDots()` doesn't even know about the variable `numDots`. If you tried to compile this program the compiler would complain that `DrawDots()` tried to use the variable `numDots` without declaring it.

The problem you are faced with is getting the value of `numDots` to the function `DrawDots()` so `DrawDots()` knows how many "dots" to draw. The answer to this problem is function parameters.

`DrawDots()` is another example of the value of writing functions. We've taken the code needed to perform a specific function (in this case, draw some dots) and embedded it in a function. Now, instead of having to duplicate the code inside `DrawDots()` every time we want to draw some dots in our program, all we'd need is a single line of code: a call to the function `DrawDots()`.

How Function Parameters Work

Function parameters are just like variables. Instead of being declared at the beginning of a function, function parameters are declared between the parentheses on the function's title line, like this:

```
void DrawDots( int numDots )
{
 /* function's body goes here */
}
```

When you call a function, you just match up the parameters, making sure you pass the function what it expects. To call the version of **`DrawDots()`** we just defined, make sure you place an **`int`** between the parentheses. The call to **`DrawDots()`** inside **`main()`**:

```
int main( void )
{
 DrawDots( 30 );

 return 0;
}
```

passes the value 30 into the function **`DrawDots()`**. When **`DrawDots()`** starts executing, it sets its parameter to the passed-in value. In this case, **`DrawDots()`** has one parameter, an **`int`** named **`numDots`**. When the call:

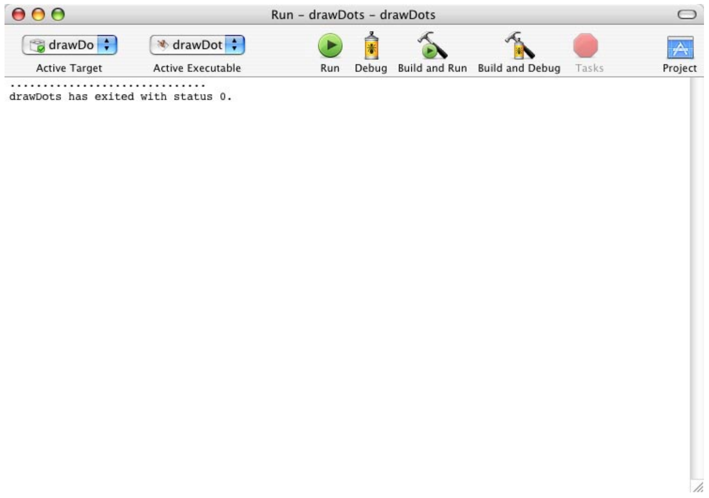
```
DrawDots( 30 );
```

executes, the function **`DrawDots()`** sets its parameter, **`numDots`**, to a value of 30. To make things a little clearer, here's a revised version of our example:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 DrawDots( 30 );

 return 0;
}

void DrawDots( int numDots )
{
 int i;

 for ( i = 1; i <= numDots; i++ )
    printf( "." );
}
```

This version of **`main()`** calls **`DrawDots()`**, passing as a parameter the constant 30. **`DrawDots()`** receives the value 30 in its **`int`** parameter, **`numDots`**. This means that the function **`DrawDots()`** starts execution with a variable named **`numDots`** having a value of 30.

Inside **`DrawDots()`**, the **`for`** loop behaves as you might expect, drawing 30 periods in the console window. Figure 7.7 shows a picture of this program in action. You can run this example yourself. The project file, **`drawDots.xcode`**, is located in the *Learn C Projects* folder in a subfolder named *07.01 - drawDots*.



```
.............................
drawDots has exited with status 0.
```

Figure 7.7 **`drawDots`** *in action.*

Parameters are Temporary

When you pass a value from a calling function to a called function, you are creating a temporary variable inside the called function. Once the called function exits (returns to the calling function), that variable ceases to exist.

In our example, we passed a value of 30 into **`DrawDots()`** as a parameter. The value came to rest in the parameter variable named **`numDots`**. Once **`DrawDots()`** exited, **`numDots`** ceased to exist.

Remember, a variable declared inside a function can only be referenced by that function.

It is perfectly acceptable for two functions to use the same variable names for completely different purposes. It's fairly standard, for example, to use a variable name like **`i`** as a counter in a **`for`** loop. What happens when, in the middle of just such a **`for`** loop, you call a function that also uses a variable named **`i`**? Here's an example:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int i;

 for ( i=1; i<=10; i++ )
 {
    DrawDots( 30 );
    printf( "\n" );
 }

 return 0;
}

void DrawDots( int numDots )
{
 int i;

 for ( i = 1; i <= numDots; i++ )
    printf( "." );
}
```

This code prints a series of 10 rows of dots, with 30 dots in each row. After each call to **`DrawDots()`**, a carriage return ("**`\n`**") is printed, moving the cursor in position to begin the next row of dots.

Notice that **`main()`** and **`DrawDots()`** each feature a variable named **`i`**. **`main()`** uses the variable **`i`** as a counter, tracking the number of rows of dots printed. **`DrawDots()`** also uses **`i`** as a counter, tracking the number of dots in the row it is printing. Won't **`DrawDots()`**'s copy of **`i`** mess up **`main()`**'s copy of **`i`**? No!

When **`main()`** starts executing, memory gets allocated for its copy of **`i`**. When **`main()`** calls **`DrawDots()`**, additional memory gets allocated for **`DrawDots()`**' copy of **`i`**. When **`DrawDots()`** exits, the memory for its copy of **`i`** is **deallocated**, freed up so it can be used again for some other variable. A variable declared within a specific function is known as a **local** variable. **`DrawDots()`** has a single local variable, the variable **`i`**.

What Does All This Have to Do with Pointers?

OK. Now we're getting to the crux of the whole matter. What do parameters have to do with pointers? To answer this question, you have to understand the two different methods of parameter passing.

Parameters are passed from function to function either by value or by address. Passing a parameter by value passes only the value of a variable or literal on to the called function. Take a look at this code:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int numDots;

 numDots = 30;

 DrawDots( numDots );

 return 0;
}

void DrawDots( int numDots )
{
 int i;

 for ( i = 1; i <= numDots; i++ )
    printf( "." );
}
```

Here's what happens when **`main()`** calls

`DrawDots()`. On the calling side, the expression passed as a parameter to **`DrawDots()`** is resolved to a single value. In this case, the expression is simply the variable **`numDots`**. The value of the expression is the value of **`numDots`**, which is 30.

On the receiving side, when **`DrawDots()`** gets called, memory is allocated for its parameters as well as for its local variables. This means that memory is allocated for **`DrawDots()`**'s copy of **`numDots`**, as well as for its copy of **`i`**. The value passed in to **`DrawDots()`** from **`main()`** (in this case, 30) is copied into the memory allocated to **`DrawDots()`**'s copy of **`numDots`**.

It is important to understand that whatever **`main()`** passes as a parameter to **`DrawDots()`** is *copied* into **`DrawDots()`**'s local copy of the parameter. Think of **`DrawDots()`**'s copy of **`numDots`** as just another local variable that will disappear when **`DrawDots()`** exits. **`DrawDots()`** can do whatever it likes to its copy of the parameter. Since it is just a local copy, any changes will have absolutely no affect on **`main()`**'s copy of the parameter.

Since passing parameters by value is a one-way operation, there's no way to get data back from the called function. Why would you ever want to? Several reasons. You might write a function that takes an employee number as a parameter. You might want that function to return the employee's salary in another parameter. How about a function that turns yards into meters? You could pass the number of yards as a value parameter, but how would you get back the number of meters?

Passing a parameter by address (instead of by value) solves this problem. If you pass the address of a variable, the receiving function can use the **`*`** operator to change the value of the original variable.

Here's an example:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int square;

 SquareIt( 5, &square );

 printf( "5 squared is %d.\n", square );

 return 0;
}


void SquareIt( int  number, int *squarePtr )
{
 *squarePtr = number * number;
}
```

In this example, **`main()`** calls the function **`SquareIt()`**. **`SquareIt()`** takes two parameters. As in our last example, both parameters are declared between the parentheses on the function's title line. Notice that we used a comma to separate the parameter declarations.

The first of **`SquareIt()`**'s two parameters is an **`int`**. The second parameter is a pointer to an **`int`**. **`SquareIt()`** squares the value passed in the first parameter, using the pointer in the second parameter to return the squared value.

If it's been ten or more years since your last math class, squaring a number is the same as multiplying the number by itself. The square of 4 is 16 and the square of 5 is 25.

Here's **`main()`**'s call of **`SquareIt()`**:

```
SquareIt( 5, &square );
```

Here's the function prototype of **`SquareIt()`**:

```
void SquareIt( int  number, int*squarePtr );
```

When **`SquareIt()`** gets called, memory is allocated for an **`int`** (**`number`**) and for a pointer to an **`int`** (**`squarePtr`**).

Once the local memory is allocated, the value 5 is copied into the local parameter **`number`**, and the address of **`square`** is copied into **`squarePtr`** (Remember, the **`&`** operator produces the address of a variable).

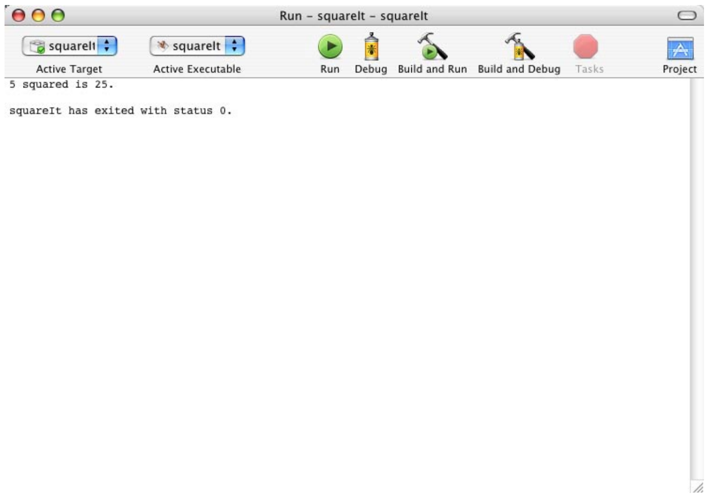
Inside the function **`SquareIt()`**, any reference to:

```
*squarePtr
```

is just like a reference to **`square`**. The assignment statement:

```
*squarePtr = number * number;
```

assigns the value 25 (since number has a value of 5) to the variable pointed to by **`squarePtr`**. This has the effect of assigning the value 25 to **`square`**. When **`SquareIt()`** returns control to **`main()`**, the value of **`square`** has been changed, as evidenced by the screen shot in Figure 7.8. If you'd like to give this code a try, you'll find it in the *Learn C Projects* folder, inside the *07.02 - squareIt* subfolder.



```
5 squared is 25.

squareIt has exited with status 0.
```

Figure 7.8 **`squareIt`** *in action.*

We'll see lots more pointer-wielding examples throughout the rest of the book.

Global Variables and Function Returns

The combination of pointers and parameters gives us one way to share variables between different functions. This section demonstrates two more techniques for doing the same.

Global variables are variables that are accessible from inside every function in your program. By declaring a global variable, two separate functions can access the same variable without passing parameters. We'll show you how to declare a global variable, then talk about when and when not to use global variables in your programs.

Another topic we'll discuss later in the chapter is a property common to all functions. All functions written in C have the ability to **return** a value to the function that calls them. You set this **return value** inside the function itself. You can use a function's return value in place of a parameter, use it to pass additional information to the calling function, or not use it at all. We'll show you how to add a return value to your functions.

Global Variables

Earlier in the chapter, you learned how to use parameters to share variables between two functions. Passing parameters between functions is great. You can call a function, pass it some data to work on, and when the function's done, it can pass you back the results.

Global variables provide an alternative to parameters. Global variables are just like regular variables, with one exception. Global variables are immune to C's scope rules. They can be referenced inside each of your program's functions. One function might initialize the global variable, another might change its value, and another function might print the value of the global variable in the console window.

As you design your programs, you'll have to make some basic decisions about data sharing between functions. If you'll be sharing a variable among a number of functions, you might want to consider making the variable a global. Globals are especially useful when you want to share a variable between two functions that are several calls apart.

Several calls apart? At times, you'll find yourself passing a parameter to a function, not because that function needs the parameter, but because the function calls another function that needs the parameter. Look at this code:

```
#include <stdio.h>

void PassAlong( int myVar );
void PrintMyVar( int myVar );

int main( void )
{
 int myVar;

 myVar = 10;
 PassAlong( myVar );

 return 0;
}

void PassAlong( int myVar )
{
 PrintMyVar( myVar );
}

void PrintMyVar( int myVar )
{
 printf( "myVar = %d", myVar );
}
```

Notice that **`main()`** passes **`myVar`** to the function **`PassAlong()`**. **`PassAlong()`** doesn't actually make use of **`myVar`**. Instead, it just passes **`myVar`** along to the function **`PrintMyVar()`**. **`PrintMyVar()`** prints **`myVar`**, then returns.

If **`myVar`** were a global, you could have avoided some parameter passing. **`main()`** and **`PrintMyVar()`** could have shared **`myVar`** without the use of parameters. When should you use parameters? When should you use globals? There's no easy answer. As you write more code, you'll develop your own coding style and, with it, your own sense of when to use globals versus parameters. For the moment, let's take a look at the proper way to add globals to your programs.

Adding Globals to Your Programs

Adding globals to your programs is easy. Just declare a variable at the beginning of your source code before the start of any of your functions. Here's the example we showed you earlier, using globals in place of parameters:

```
#include <stdio.h>

void PassAlong( void );
void PrintMyVar( void );

int  gMyVar;

int main (int argc, const char * argv[])
{
 gMyVar = 10;

 PassAlong();

 return 0;
}

void PassAlong( void )
{
 PrintMyVar();
}

void PrintMyVar( void )
{
 printf( "gMyVar = %d", gMyVar );
}
```

This example starts with a variable declaration, right at the top of the program. Because **`gMyVar`** was declared at the top of the program, **`gMyVar`** becomes a global variable, accessible to each of the program's functions. Notice that none of the functions in this version use parameters. As a reminder, when a function is declared without parameters, use the keyword **`void`** in place of a parameter list.

Did you notice that letter **`g`** at the beginning of the global's name? Many C programmers start each of their global variables with the letter **`g`** (for global). Doing this will distinguish your local variables from your global variables and will make your code much easier to read.

When to Use Globals

In general, you should try to minimize your use of globals. On one hand, global variables make programming easier, because you can access a global anywhere. With parameters, you have to pass the parameter from function to function, until it gets to where it will be used.

On the other hand, globals are expensive, memory-wise. Since the memory available to your program is finite, you should try to be memory conscious whenever possible. What makes global variables expensive where memory is concerned? Whenever a function is called, memory for the function's variables is allocated on a temporary basis. When the function exits, the memory allocated to the function is freed up (put back into the pool of available memory). Global variables, on the other hand, are around for the life of your program. Memory for each global is allocated when the program first starts running and isn't freed up until the program exits.

Try to minimize your use of globals, but don't be a miser. If using a global will make your life easier, go ahead and use it.

Function Returns

Before we get to our source code examples, there's one more subject to cover. In addition to passing a parameter and using a global variable, there's one more way to share data between two functions. Every function returns a value to the function that called it. You can use this return value to pass data back from a called function.

So far, all of our examples have ignored **function return values**. The return value only comes into play when you call a function in an expression, like this:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int sum;

 sum = AddTheseNumbers( 5, 6 );

 printf( "The sum is %d.", sum );

 return 0;
}
```

```
int  AddTheseNumbers( int num1, int num2 )
{
 return( num1 + num2 );
}
```

There are a few things worth noting in this example. First, take a look at the function specifier for **`AddTheseNumbers()`**. So far in this book, every single function other than **`main()`** has been declared using the keyword **`void`**. **`AddTheseNumbers()`**, like **`main()`**, starts with the keyword **`int`**. This keyword tells you the type returned by this function. A function declared with the **`void`** keyword doesn't return a value. A function declared with the **`int`** keyword returns a value of type **`int`**.

A function returns a value by using the **`return`** keyword, followed by an expression that represents the value you want returned. For example, take a look at this line of code from **`AddTheseNumbers()`**:

```
 return( num1 + num2 );
```

This line of code adds the two variables **`num1`** and **`num2`** together, then returns the sum. To understand what that means, take a look at this line of code from **`main()`** that calls **`AddTheseNumbers()`**:

```
 sum = AddTheseNumbers( 5, 6 );
```

This line of code first calls **`AddTheseNumbers()`**, passing in values of 5 and 6 as parameters. **`AddTheseNumbers()`** adds these numbers together and returns the value 11, which is then assigned to the variable **`sum`**.

When you use a function inside an expression, the computer makes the function call, then substitutes the function's return value for the function when it evaluates the rest of the expression.

There are several ways to use **`return`**. To immediately exit a function, without establishing a return value, use the statement:

```
return;
```

or

```
return();
```

The parentheses in a **`return`** statement are optional. You'd use the plain **`return`**, without an expression, to return from a function of type **`void`**. You might use this immediate **`return`** in case of an error, like this:

```
if ( OutOfMemory() )
 return;
```

What you'll want to remember about this form of return is that it does not establish the return value of the function. This works fine if your function is declared **`void`**:

```
void MyVoidFunction( int myParam );
```

but won't cut it if your function is declared to return a value:

```
int  AddTheseNumbers( int num1, int num2 )
```

If you forget to specify a return value, some compilers will say nothing, some will print warnings, and others will report errors.

`AddTheseNumbers()` is declared to return a value of type **`int`**. Here are two different versions of the **`AddTheseNumbers() return`** statement:

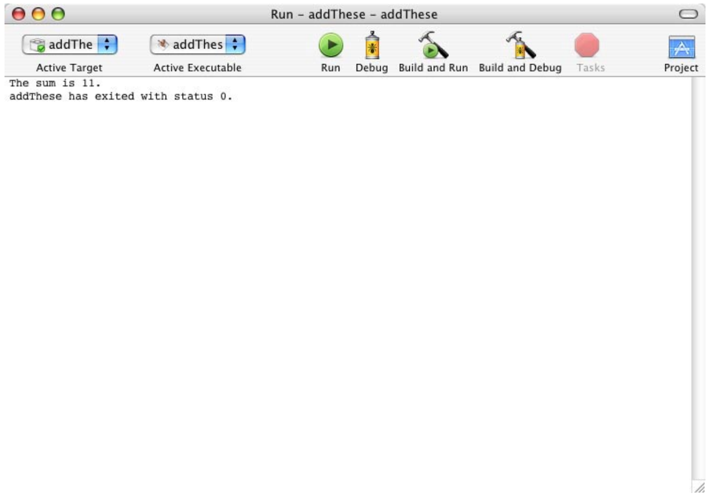
```
return( num1 + num2 );
```

and

```
return num1 + num2;
```

Notice that the second version did not include any parentheses. Since **`return`** is a keyword and not a function call, either of these forms is fine.

You can find a version of this program on your hard drive. Look in the folder *Learn C Projects*, in the subfolder *07.03 - addThese*. Figure 7.9 shows the output of this program.



```
The sum is 11.
addThese has exited with status 0.
```

Figure 7.9 `addThese` *in action.*

Danger! Avoid Uninitialized Return Values!

Before we leave the topic of function return values, there's one pitfall worth mentioning. If you're going to use a function in an expression, make sure the function provides a return value. For example, this code will produce unpredictable results:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int sum;

 sum = AddTheseNumbers( 5, 6 );

 printf( "The sum is %d.", sum );

 return 0;
}


int  AddTheseNumbers( int num1, int num2 )
{
 return;   /* Yikes! We forgot to
                 set the return value */
}
```

When **`AddTheseNumbers()`** returns, what will its value be? No one knows! When I ran the above example on my computer, Xcode reported a warning (as it should), then ran the program, generating a sum of 7124. Unpredictable results! Don't forget to set a return value if you intend to use a function in an expression.

To Return or Not to Return

Should you use a return value or a passed-by-address parameter? Which is correct? This is basically a question of style. Either solution will get the job done, so feel free to use whichever works best for you. Just remember that a function can have only one return value but an unlimited number of parameters. If you need to get more than one piece of data back to the calling function, your best bet is to use parameters.

The function **`AddTheseNumbers()`** was a natural fit for the **`return`** statement. It took in a pair of numbers (the input parameters), and needed to return the sum of those numbers. Since it only needed to return a single value, the **`return`** statement worked perfectly.

Another nice thing about using the **`return`** statement, is that it frequently allows us to avoid declaring an extra variable. In **`addThese`**, we declared **`sum`** to receive the value returned by **`AddTheseNumbers()`**. Since all we did with **`sum`** was print its value, we could have accomplished the same thing with this version of **`main()`**:

```
int main (int argc, const char * argv[])
{
 printf( "The sum is %d.", AddTheseNumbers( 5,
 6 ) );

 return 0;
}
```

See the difference? We included the call to **`AddTheseNumbers()`** in the **`printf()`**, bypassing **`sum`** entirely. When **`AddTheseNumbers()`** returns its **`int`**, that value is passed on to **`printf()`**.

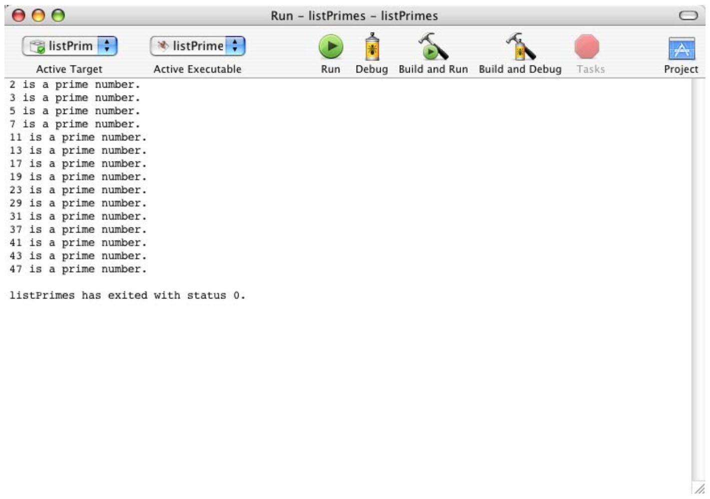
More Sample Programs

Are you ready for some more code? The next few sample programs make use of pointers, function parameters, global variables, and function returns. Fire up your Mac, crank up your iPod, and break out the pizza. Let's code!

listPrimes.xcode

Our next sample program is an updated version of Chapter 6's prime number program, **`nextPrime`**, which found the next prime number following a specified number. The example we presented reported that the next prime number after 19 was 23.

This program, called **`listPrimes`**, uses a function named **`IsItPrime()`** and lists all the prime numbers between 1 and 50. Open up the project *listPrimes.xcode*. You'll find it in the *Learn C Projects* folder, inside the subfolder named *07.04 - listPrimes*. Run **`listPrimes`**, then compare your results with the console window shown in Figure 7.10.



```
2 is a prime number.
3 is a prime number.
5 is a prime number.
7 is a prime number.
11 is a prime number.
13 is a prime number.
17 is a prime number.
19 is a prime number.
23 is a prime number.
29 is a prime number.
31 is a prime number.
37 is a prime number.
41 is a prime number.
43 is a prime number.
47 is a prime number.

listPrimes has exited with status 0.
```

Figure 7.10 **`listPrimes`** *in action.*

Let's take a look at the source code...

Stepping Through the Source Code

listPrimes.c consists of two functions: **`main()`** and **`IsItPrime()`**. **`IsItPrime()`** takes a single parameter, an **`int`** named **`candidate`**, which is passed by value. **`IsItPrime()`** returns a value of **`true`** if **`candidate`** is a prime number and a value of **`false`** otherwise.

listPrimes.c starts off with three **`#include`**s. **`stdio.h`** gives us access to the function prototype of **`printf()`**, **`c.h`** gives us the definitions of **`true`** and **`false`**, and **`math.h`** gives us access to the function prototype for **`sqrt()`**.

```
#include <stdio.h>
#include <c.h>
#include <math.h>
```

Next comes the function prototype for **`IsItPrime()`**. The compiler will use this function prototype to make sure that all calls to **`IsItPrime()`** pass the right number of parameters (in this case, 1) and that the parameters are of the correct type (in this case, a single **`int`**).

```
int  IsItPrime( int candidate );
```

`main()` defines a single variable, an **`int`** named **`i`**. We'll use **`i`** as a counter to step through the integers from 1 to 50. We'll pass each number to **`IsItPrime()`** and, if the result is **`true`**, we'll report the number as prime.

```
int main (int argc, const char * argv[])
{
 int i;

 for ( i = 1; i <= 50; i++ )
 {
    if ( IsItPrime( i ) )
           printf( "%d is a prime number.\n",
 i );
 }

 return 0;
}
```

As usual, **`main()`** ends with a **`return`** statement. By convention, returning a value of 0 tells the outside world that everything ran just honky-dory. If something goes wrong (if we ran out of memory perhaps) the same convention calls for us to return a negative number from **`main()`**. Some operating systems will make use of this return value, others won't. It doesn't cost you anything to follow the convention, so go ahead and follow it.

`IsItPrime()` first checks to see if the number passed in is less than 2. If so, **`IsItPrime()`** returns **`false`**, since 2 is the first prime number.

```
int  IsItPrime( int candidate )
{
  int i, last;

  if ( candidate < 2 )
     return false;
```

If **`candidate`** has a value of 2 or greater, we'll step through all the numbers between 2 and the square root of **`candidate`** looking for a factor. If this algorithm is new to you, go back to the previous chapter and check out the program **`nextPrime`**. If we find a factor, we know the number isn't prime, and we'll return **`false`**.

```
  else
  {
     last = sqrt( candidate );

     for ( i = 2; i <= last; i++ )
     {
        if ( (candidate % i) == 0 )
           return false;
     }
  }
```

If we get through the loop without finding a factor, we know **`candidate`** is prime, and we return **`true`**.

```
  return true;
}
```

If **`candidate`** is equal to 2, **`last`** will be equal to 1.414, which will get truncated to 1, since **`last`** is an **`int`**. If **`last`** is 1, the **`for`** loop won't even get through 1 iteration and will fall through to the statement:

```
return true;
```

The same thing happens if **`candidate`** is 3. Since 2 and 3 are both prime, this works just fine. On the other hand, this little example shows you how careful you have to be to check your code, to make sure it works in all cases.

Consider the function name **`IsItPrime()`**. In C, when you name a function in the form of a **`true`** or

false question, it is good form to return a value of **true** or **false**. The question this function answers is, “Is the candidate prime?” It is critical that **IsItPrime()** return **true** if the candidate was prime and **false** otherwise. When **main()** calls **IsItPrime()**, **main()** is asking the question, “Is the candidate prime?” In the case of the **if** statement:

```
if ( IsItPrime( i ) )
 printf( ... );
```

main() is saying, “If **i** is prime, do the **printf()**.” Make sure your function return values make sense!

power.xcode

Our next program combines a global variable, a pointer parameter, and some value parameters. At the heart of the program is a function, called **DoPower()**, that takes three parameters. **DoPower()** takes a base and an exponent, raises the base to the exponent power, and returns the result in a parameter. Raising a base to an exponent power is the same as multiplying the base by itself, an exponent number of times.

For example, raising 2 to the fifth power (written as 2^5) is the same as saying 2*2*2*2*2, which is equal to 32. In the expression 2^5, 2 is the base and 5 is the exponent. The function **DoPower()** takes a base and an exponent as parameters and raises the base to the exponent power. **DoPower()** uses a third parameter to return the result to the calling function.

The program also makes use of a global variable, an **int** named **gPrintTraceInfo**, which demonstrates one of the most important uses of a global variable. Every function in the program checks the value of the global **gPrintTraceInfo**. If **gPrintTraceInfo** is **true**, each function prints a message when the function is entered, and another message when the function exits. In this way, you can **trace** the execution of the program. By reading the **printf()**s, you can see when a function is entered and when it leaves.

Most modern development environments feature a piece of software, called a **debugger**, that lets you trace the execution of your program, one line at a time. Xcode has an excellent debugger!

To give it a try, first click in the left-hand-column of your source code window, just to the left of a line of code, to create a new **breakpoint** (a place for the debugger to stop). Once the little breakpoint arrow appears, select Show Debugger from the Debug menu, then click the *Build and Debug* icon at the top of the debugger window. Click the *Step Over* and *Step Into* buttons to step through your program. To exit, either step all the way through the program, or click the stop sign icon.

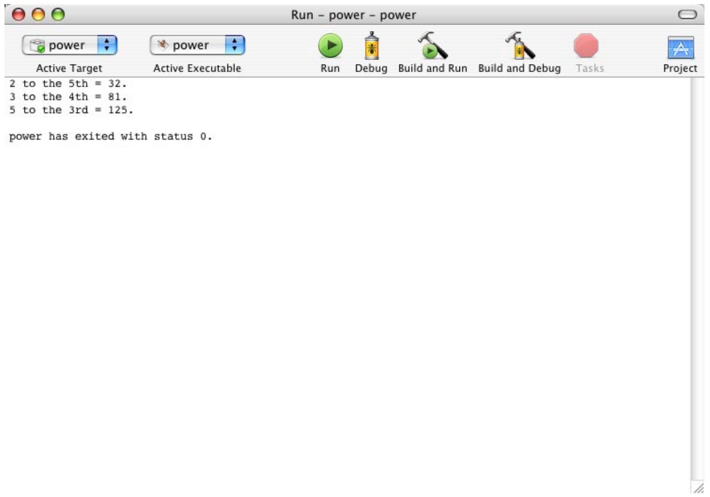
Even if you have a debugger, there will be times when it is handy to stick a debugging `printf()` in your code. Whatever gets the job done.

If `gPrintTraceInfo` is set to `true`, the extra function-tracing information will be printed in the console window. If `gPrintTraceInfo` is set to `false`, the extra information will not be printed.

As you'll see in a moment, by simply changing the value of a global, you can dramatically change the way your program runs.

Running power

You'll find *power.xcode* in the *Learn C Projects* folder, in the *07.05 - power* subfolder. Run `power` and compare your results with the console window shown in Figure 7.11. This output was produced by three consecutive calls to the function `DoPower()`. The three calls calculated the result of the expressions 2^5, 3^4, and 5^3. Here's how the program works.



```
2 to the 5th = 32.
3 to the 4th = 81.
5 to the 3rd = 125.

power has exited with status 0.
```

Figure 7.11 `power` *output, with* `gPrintTraceInfo` *set to* `false`.

Stepping Through the Source Code

main.c starts with a pair of standard `#include`s and the function prototype for `DoPower()`. Notice that `DoPower()` is declared to be of type `void`, telling you that `DoPower()` doesn't return a value. As you read through the code, think about how you might rewrite `DoPower()` to return its result using the `return` statement instead of via a parameter.

```
#include <stdio.h>
#include <c.h>

void DoPower( int *resultPtr, int base, int
 exponent );
```

Next comes the declaration of our global, **`gPrintTraceInfo`**. Once again, notice that the global starts with a **`g`**.

```
int        gPrintTraceInfo;
```

`main()` starts off by setting **`gPrintTraceInfo`** to **`false`**. Next, we check to see if tracing is turned on. If so, we'll print a message telling us we've entered **`main()`**.

```
int main (int argc, const char * argv[])
{
 int power;

 gPrintTraceInfo = false;

 if ( gPrintTraceInfo )
    printf( "---> Starting main()...\n" );
```

C guarantees that it will initialize all global variables to zero. Since **`false`** is equivalent to zero, we could have avoided setting **`gPrintTraceInfo`** to **`false`**, but that would have been a mistake. Explicitly setting the global to a value makes the code easier to read and is the right thing to do!

Here are our three calls to **`DoPower()`**, each of which is followed by a **`printf()`** reporting our results. If **`DoPower()`** returned its results via a **`return`** statement, we could have eliminated the variable **`power`**, and embedded the call to **`DoPower()`** inside the **`printf()`** in **`power`**'s place.

```
 DoPower( &power, 2, 5 );
 printf( "2 to the 5th = %d.\n", power );

 DoPower( &power, 3, 4 );
 printf( "3 to the 4th = %d.\n", power );

 DoPower( &power, 5, 3 );
 printf( "5 to the 3rd = %d.\n", power );
```

If tracing is turned on, we'll print a message saying that we are leaving **`main()`**.

```
 if ( gPrintTraceInfo )
    printf( "---> Leaving main()...\n" );

 return 0;
}
```

The function **`DoPower()`** takes three parameters. **`resultPtr`** is a pointer to an **`int`**. We'll use that pointer to pass back the function results. **`base`** and **`exponent`** are value parameters that represent the—guess what?—base and exponent.

```
void DoPower( int *resultPtr, int base, int
 exponent )
{
 int i;
```

Once again, check the value of **`gPrintTraceInfo`**. If it's **`true`**, print a message telling us we're at the beginning of **`DoPower()`**. Notice the tab character (represented by the characters **`\t`**) at the beginning of the **`printf()`** quoted string. You'll see what this was for when we set **`gPrintTraceInfo`** to **`true`**.

```
 if ( gPrintTraceInfo )
    printf( "\t---> Starting DoPower()...\n"
 );
```

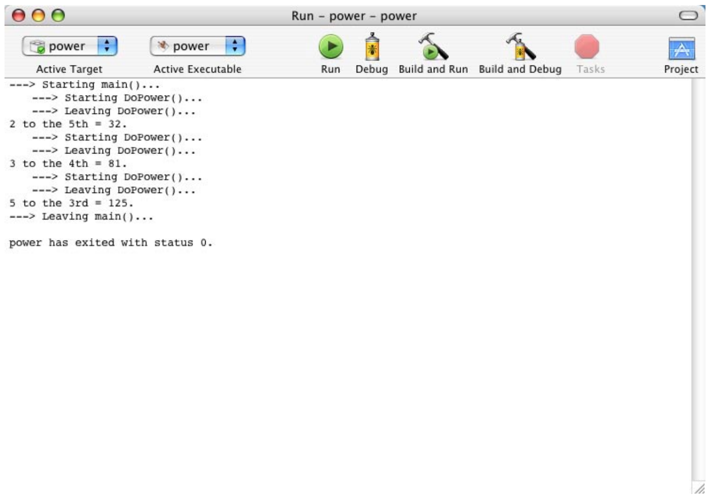
The following three lines calculate **`base`** raised to the **`exponent`** power, accumulating the results in the memory pointed to by **`resultPtr`**. When **`main()`** called **`DoPower()`**, it passed **`&power`** as its first parameter. This means that **`resultPtr`** contains the address of (points to) the variable **`power`**. Changing **`*resultPtr`** is exactly the same as changing **`power`**. When **`DoPower()`** returns to **`main()`**, the value of **`power`** will have been changed. **`power`** was passed by-address (also called by-reference), instead of by-value.

```
 *resultPtr = 1;
 for ( i = 1; i <= exponent; i++ )
    *resultPtr *= base;
```

Finally, if **`gPrintTraceInfo`** is **`true`**, print a message telling us we're leaving **`DoPower()`**.

```
 if ( gPrintTraceInfo )
    printf( "\t---> Leaving DoPower()...\n"
 );
}
```

Figure 7.12 shows the console window when **`power`** is run with **`gPrintTraceInfo`** set to **`true`**. See the trace information? Find the lines printed when you enter and exit **`DoPower()`**. The leading tab characters help distinguish these lines.



```
---> Starting main()...
    ---> Starting DoPower()...
    ---> Leaving DoPower()...
2 to the 5th = 32.
    ---> Starting DoPower()...
    ---> Leaving DoPower()...
3 to the 4th = 81.
    ---> Starting DoPower()...
    ---> Leaving DoPower()...
5 to the 3rd = 125.
---> Leaving main()...

power has exited with status 0.
```

Figure 7.12 `power` *output, with* `gPrintTraceInfo` *set to* `true`*.*

This tracing information was turned on and off by a single global variable. As you start writing your own programs, you'll want to develop your own set of global variable tricks.

What's Next?

Wow! You really are becoming a C programmer. In this chapter alone, you covered pointers, function parameters (both by-value and by-address), global variables, and function return values.

You're starting to develop a sense of just how powerful and sophisticated the C language really is. You've built an excellent foundation. Now you're ready to take off.

The second half of our book (Volume 2) starts with the introduction of the concept of data types. Throughout the book, you've been working with a single data type, the `int`. Our next chapter will introduce the concept of arrays, strings, pointer arithmetic and typed function return values. Let's go.

Exercises

1) Predict the result of each of the following code fragments:

a)

```
void AddOne( int   *myVar );

int main (int argc, const char * argv[])
{
 int num, i;

 num = 5;

 for ( i = 0; i < 20; i++ )
    AddOne( &num );

 printf( "Final value is %d.", num );

 return 0;
}

void AddOne( int   *myVar )
{
 (*myVar) ++;
}
```

b)

```
int  gNumber;
int  MultiplyIt( int myVar );

int main (int argc, const char * argv[])
{
 int i;
 gNumber = 2;

 for ( i = 1; i <= 2; i++ )
    gNumber *= MultiplyIt( gNumber );

 printf( "Final value is %d.", gNumber );

 return 0;
}

int  MultiplyIt( int myVar )
{
 return( myVar * gNumber );
}
```

c)

```
int  gNumber;
int  DoubleIt( int myVar );

int main (int argc, const char * argv[])
{
 int i;
 gNumber = 1;

 for ( i = 1; i <= 10; i++ )
    gNumber = DoubleIt( gNumber );

 printf( "Final value is %d.", gNumber );

 return 0;
}

int  DoubleIt( int myVar )
{
 return 2 * myVar;
}
```

2) Modify *main.c*. Delete the first parameter of the function **`DoPower()`**, modifying the routine to return its result as a function return value instead.

3) Modify *main.c*. Instead of printing prime numbers, print only non-prime numbers. In addition, print one message for non-primes that are multiples of 3 and a different message for non-primes that are not multiples of 3.

Chapter 8 Variable Data Types

Now we’re cooking! You may now consider yourself a C Programmer, First Class. At this point, you’ve mastered all the basic elements of C programming. You know that C programs are made up of functions, one—and only one!—of which is named **`main()`**. Each of these functions uses keywords (such as **`if`**, **`for`**, and **`while`**), operators (such as **`=`**, **`++`**, and **`*=`**), and variables to manipulate the program’s data.

Sometimes you’ll use a global variable to share data between several functions. At other times, you’ll use a parameter to share a variable between a calling and a called function. Sometimes these parameters are passed by value, and sometimes pointers are used to pass a parameter by address. Some functions return values. Others, declared with the **`void`** keyword, don’t return a value.

In this chapter, we’ll focus on **variable types**. Each of the variables in the previous example programs has been declared as an **`int`**. As you’ll soon see, there are many other data types out there.

Other Data Types

So far, the focus has been on **`int`**s, which are extremely useful when it comes to working with numbers. You can add two **`int`**s together. You can check if an **`int`** is even, odd, or prime. There are a lot of things you can do with **`int`**s, as long as you limit yourself to whole numbers.

Just as a reminder, 527, 33, and -2 are all whole numbers, while 35.7, 92.1, and -1.2345 are not whole numbers.

What do you do if you want to work with non-whole numbers, such as 3.14159 and -98.6? Check out this slice of code:

```
int myNum;

myNum = 3.5;
printf( “myNum = %d”, myNum );
```

Since **`myNum`** is an **`int`**, the number 3.5 will be truncated before it is assigned to **`myNum`**. When this code ends, **`myNum`** will be left with a value of 3 and not 3.5 as intended. Do not despair. There are several special C data types created especially for working with non-whole, or **floating point** numbers.

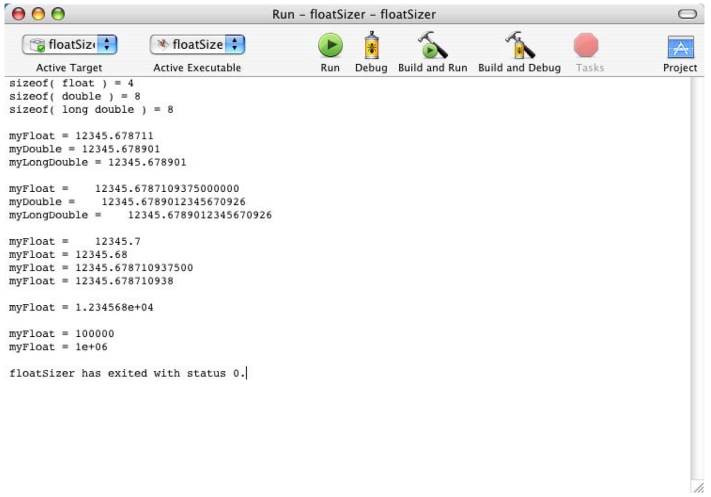
The name floating-point refers to the decimal point found in all floating-point numbers.

The three floating point data types are **`float`**, **`double`**, and **`long double`**. The difference between these types is the number of bytes allocated to each and, therefore, the range of values each can hold. The relative sizes of these three types is completely implementation dependent. Here's a program you can run to tell you the size of these three types in your development environment, and to show you various ways to use **`printf()`** to print floating point numbers.

floatSizer

Look inside the *Learn C Projects* folder, inside the subfolder named *08.01 - floatSizer*, and open the project named *floatSizer.xcode*. Figure 8.1 shows the results when I ran **`floatSizer`** on my Mac using Xcode. The first three lines of output tell you the size, in bytes, of the types **`float`**, **`double`**, and **`long double`**, respectively.

Never assume the size of a type. As you'll see when we go through the source code, C gives you everything you need to check the size of a specific type in your development environment. If you need to be sure of a type's size, write a program and check the size for yourself.



```
sizeof( float ) = 4
sizeof( double ) = 8
sizeof( long double ) = 8

myFloat = 12345.678711
myDouble = 12345.678901
myLongDouble = 12345.678901

myFloat =    12345.6787109375000000
myDouble =    12345.6789012345670926
myLongDouble =    12345.6789012345670926

myFloat =    12345.7
myFloat = 12345.68
myFloat = 12345.678710937500
myFloat = 12345.678710938

myFloat = 1.234568e+04

myFloat = 100000
myFloat = 1e+06

floatSizer has exited with status 0.
```

Figure 8.1 *The output from `floatSizer`.*

Walking Through the Source Code

main.c starts with the standard **`#include`**.

```
#include <stdio.h>
```

`main()` defines three variables, a **`float`**, a **`double`**, and a **`long double`**.

```
int main (int argc, const char * argv[])
{
 float                   myFloat;
 double                  myDouble;
 long double             myLongDouble;
```

Next, we'll assign a value to each of the three variables. Notice that we've assigned the same number to each.

```
 myFloat = 12345.67890123456789;
 myDouble = 12345.67890123456789;
 myLongDouble = 12345.67890123456789;
```

Now comes the fun part. We'll start by using C's **`sizeof`** operator to print the size of each of our three floating point types. Even though **`sizeof`** doesn't look like the other operators we've seen (**`+`**, **`*`**, **`<<`**, and so on) it is indeed an operator. Stranger still, **`sizeof`** requires a pair of parentheses surrounding a single parameter, much like a function. The parameter is either a type or a variable. **`sizeof()`** returns the size, in bytes, of its parameter.

Like **`return`**, **`sizeof`** doesn't always *require* a pair of parentheses. If the **`sizeof`** operand is a type, the parentheses are required. If the **`sizeof`** operand is a variable, the parentheses are optional. Rather than trying to remember this rule, avoid confusion and always use parentheses with **`sizeof`**.

Did you notice the **`(int)`** to the left of each **`sizeof`**? This is known as a **typecast**. A typecast tells the compiler to convert a value of one type to a specified type. In this case, we are taking the type returned by **`sizeof`** and converting it to an **`int`**. Why do this? **`sizeof`** returns a value of type **`size_t`** (wierd type name, eh?) and **`printf()`** doesn't have a format specifier that corresponds to a **`size_t`**. By converting the **`size_t`** to an **`int`**, we can use the "**`%d`**" format specifier to print the value returned by **`sizeof`**. Notice the extra "**`\n`**" at the end of the third **`printf()`** that gives us a blank line between the first three lines of output and the next line of output.

```
 printf( "sizeof( float ) = %d\n", (int)sizeof(
 float ) );
 printf( "sizeof( double ) = %d\n",
 (int)sizeof( double ) );
 printf( "sizeof( long double ) = %d\n\n",
 (int)sizeof( long double ) );
```

If the concept of typecasting is confusing to you, have no fear. We'll get into typecasting in Chapter 11. Till then, you can use this method whenever you want to print the value returned by **`sizeof`**. Alternatively, you might declare a variable of type **`int`**, assign the value returned by **`sizeof`** to the **`int`**, then print the **`int`**:

```
int   myInt;

myInt = sizeof( float );

printf( "sizeof( float ) = %d\n", myInt
);
```

Use whichever method works for you.

The rest of this program is dedicated to various and sundry ways you can print your floating point numbers. So far, all of our programs have printed **`int`**s using the "**`%d`**" format specifier. The Standard Library has a set of format specifiers for all of C's built-in data types, including several for printing floating point numbers.

First, we'll use the format specifer "**`%f`**" to print our three floating point numbers in their natural, decimal format.

```
printf( "myFloat = %f\n", myFloat );
printf( "myDouble = %f\n", myDouble );
printf( "myLongDouble = %f\n\n", myLongDouble
);
```

Here's the result of these three **`printf()`**s:

```
myFloat = 12345.678711
myDouble = 12345.678901
myLongDouble = 12345.678901
```

As a reminder, all three of these numbers was assigned the value:

```
12345.67890123456789
```

Hmmm...None of the numbers we printed matches this number. And the first number we printed is different than the second and third numbers. What gives? There are several problems here. As we've already seen, this development environment uses 4 bytes for a **`float`** and 8 bytes each for a **`double`** and **`long double`**. This means that the number:

```
12345.67890123456789
```

can be represented more accurately using a **`double`** or **`long double`** than it can be using a **`float`**. In addition, we are printing using the default precision of the "**`%f`**" format specifier. In this case, we are only printing 6 places past the decimal point. Though this might be plenty of precision for most applications, we'd like to see how accurate we can get.

Our next three **`printf()`**s use format specifier **modifiers** to more closely specify the output produced by **`printf()`**. By using "**`%25.16f`**" instead of "**`%f`**", we tell **`printf()`** to print the floating point number with an accuracy of 16 places past the decimal, and to add spaces if necessary so the number takes up at least 25 character positions.

```
printf( "myFloat = %25.16f\n", myFloat );
printf( "myDouble = %25.16f\n", myDouble );
printf( "myLongDouble = %25.16f\n\n",
myLongDouble );
```

Here's the result of these three **`printf()`**s:

```
myFloat =    12345.6787109375000000
myDouble =    12345.6789012345670926
myLongDouble =    12345.6789012345670926
```

`printf()` printed each of these numbers to 16 places past the decimal place (count the digits yourself), padding each result with zeros as needed. Since the 16 digits to the right of the decimal, plus 1 space for the decimal, plus 5 for the 5 digits to the left of the decimal is equal to 22 (16+1+5=22), and we asked **`printf()`** to use 25 character positions, **`printf()`** added 3 spaces to the left of the number.

We originally asked **`printf()`** to print a **`float`** with a value of:

```
12345.67890123456789
```

The best approximation of this number we were able to represent by a **`float`** is:

```
12345.6787109375000000
```

Where did this approximation come from? The answer has to do with the way your computer stores floating-point numbers.

The fractional part of a number (the number to the right of the decimal) is represented in binary just like an integer. Instead of the sum of powers of 2, the fractional part is represented as the sum of powers of 1/2. For example, the number .75 is equal to 1/2 + 1/4. In binary, that's 11.

The problem with this representation is that it's impossible to represent some numbers with complete accuracy. If you need a higher degree of accuracy, use **`double`** or **`long double`** instead of **`float`**. Unless you cannot afford the extra memory that the larger data types require, you are probably better off using a **`double`** or **`long double`** in your programs instead of a **`float`** for all your floating point calculations.

Note that even an 8-byte **`double`** is not big enough to perfectly represent our original number. Pretty darn close, though!

The next four **`printf()`**s show you the result of using different modifer values to print the same **`float`**.

```
printf( "myFloat = %10.1f\n", myFloat );
printf( "myFloat = %.2f\n", myFloat );
printf( "myFloat = %.12f\n", myFloat );
printf( "myFloat = %.9f\n\n", myFloat );
```

Here's the output produced by each of the **`printf()`**s.

```
myFloat =    12345.7
myFloat = 12345.68
myFloat = 12345.678710937500
myFloat = 12345.678710938
```

The specifier "**`%10.1f`**" told **`printf()`** to print 1 digit past the decimal and to use 10 character positions for the entire number. The specifier "**`%.2f`**" told **`printf()`** to print 2 digits past the decimal and to use as many character positions as necessary to print the entire number. Notice that **`printf()`** rounds off the result for you and doesn't simply cut off the number after the specified number of places.

The specifier "**`%.12f`**" told **`printf()`** to print 12 digits past the decimal and the specifier "**`%.9f`**" told **`printf()`** to print 9 digits past the decimal. Again, notice the rounding that takes place.

Unless you need to exactly control the total number of characters used to print a number, you'll probably leave off the first modifier and just specify the number of digits past the decimal you want printed, using specifiers like "**`%.2f`**" and "**`%.9f`**".

If you do use a two part modifier like "**`%3.2f`**", **`printf()`** will never cut off numbers to the left of the decimal. For example, this code:

```
myFloat = 255.543;

printf( "myFloat = %3.2f", myFLoat );
```

will produce this output:

```
myFloat = 255.54
```

Even though you told **`printf()`** to use 3 character positions to print the number, **`printf()`** was smart enough to not lose the numbers to the left of the decimal.

The next **`printf()`** uses the specifier "**`%e`**", asking **`printf()`** to print the **`float`** using **scientific** or **exponential** notation.

```
printf( "myFloat = %e\n\n", myFloat );
```

Here's the corresponding output:

```
myFloat = 1.234568e+04
```

1.234568e+04 is equal to 1.234568 times 10 to the 4th power, or 1.234568*10^4 or 1.234568 * 10000 **==** 12,345.68. The next two **printf()**s uses the specifier “**%g**”, letting **printf()** decide whether decimal or scientific notation will be the most efficient way to represent this number. The first “**%g**” deals with a **myFloat** value of 100,000.

```
   myFloat = 100000;
   printf( "myFloat = %g\n", myFloat );
```

Here’s the output:

```
 myFloat = 100000
```

Next, **myFLoat**’s value is changed to 1,000,000 and “**%g**” is used once again:

```
   myFloat = 1000000;
   printf( "myFloat = %g\n", myFloat );

   return 0;
 }
```

Here’s the result of this last **printf()**. As you can see, this time **printf()** decided to represent the number using exponential notation:

```
 myFloat = 1e+06
```

The lesson here is, use **float**s if you want to work with floating-point numbers. Use **double**s or **long double**s for extra accuracy, but beware the extra cost in memory usage. Use **int**s for maximum speed, if you want to work exclusively with whole numbers, or if you want to truncate a result.

The Integer Types

So far, you’ve learned about 4 different types - three floating point types (**float**, **double**, and **long double**) and one integer type (**int**). In this section, we’ll introduce the remaining integer types: **char**, **short**, and **long**. As was the case with the three floating point types, the size of each of the 4 integer types is implementation dependent. Our next program, **intSizer** proves that point. You’ll find *intSizer.xcode* in the *Learn C Projects* folder, in the *08.02 - intSizer* subfolder.

Though these forms are rarely used, a **`short`** is also known as a **`short int`** and a **`long`** is also known as a **`long int`**. As an example, these declarations are perfectly legal:

```
short int   myShort;

long int    myLong;
```

Though the preceding declarations are just fine, you are more likely to encounter declarations like these:

```
short     myShort;

long      myLong;
```

As always, choose your favorite style and be consistent.

`intSizer` consists of four **`printf()`**s, one for each of the integer types:

```
printf( "sizeof( char ) = %d\n", (int)sizeof(
char ) );
printf( "sizeof( short ) = %d\n",
(int)sizeof( short ) );
printf( "sizeof( int ) = %d\n", (int)sizeof(
int ) );
printf( "sizeof( long ) = %d\n", (int)sizeof(
long ) );
```

Like their **`floatSizer`** counterparts, these **`printf()`**s use **`sizeof`** to determine the size of a **`char`**, a **`short`**, an **`int`**, and a **`long`**. When I ran **`intSizer`** on my Mac, here's what I saw:

```
sizeof( char ) = 1
sizeof( short ) = 2
sizeof( int ) = 4
sizeof( long ) = 4
```

Again, the point to remember is, there are *no* guarantees. Don't assume the size of a type. Write a program and check for yourself.

Type Value Ranges

All the integer types can be either **`signed`** or **`unsigned`**. This obviously affects the range of values handled by that type. For example, a **`signed`** 1 byte **`char`** can store a value from -128 to 127, while an **`unsigned`** 1 byte **`char`** can store a value from 0 to 255. If this clouds your mind with pain, now might be a good time to go back and review Chapter 5.

A **`signed`** 2 byte **`short`** can store values ranging from –32,768 to 32,767, while an **`unsigned`** 2 byte **`short`** can store values ranging from 0 to 65,535.

A **`signed`** 4 byte **`long`** or **`int`** can store values ranging from -2,147,483,648 to 2,147,483,647, while an **`unsigned`** 4 byte **`long`** or **`int`** can store values ranging from 0 to 4,294,967,295.

A 4 byte **`float`** can range in value from -3.4e+38 to 3.4e+38. An 8 byte **`double`** or **`long double`** can range in value from -1.7e+308 to 1.7e+308.

Memory Efficiency Versus Safety

Each time you declare one of your program's variables, you'll have a decision to make. What's the best type for this variable? In general, it's a good policy not to waste memory. Why use a **`long`** when a **`short`** will do just fine? Why use a **`double`** when a **`float`** will do the trick?

There is a danger in being too concerned with memory efficiency. For example, suppose a customer asked you to write a program designed to print the numbers 1 through 100, one number per line. Sounds pretty straightforward. Just create a **`for`** loop and embed a **`printf()`** in the loop. In the interests of memory efficiency, you might use a **`char`** to act as the loop's counter. After all, if you declare your counter as an **`unsigned char`**, it can hold values ranging from 0 to 255. That should be plenty, right?

```
unsigned char    counter;

for ( counter=1; counter<=100; counter++ )
 printf( "%d\n", counter );
```

This program works just fine. But suppose your customer comes back with a request, asking you to extend the program to count from 1 to 1000 instead of just to 100. You happily change the 100 to 1000 like so:

```
unsigned char    counter;

for ( counter=1; counter<=1000; counter++ )
 printf( "%d\n", counter );
```

and take it for a spin. What do you think will happen when you run it? To find out, open the *Learn C Projects* folder, open the *08.03 - typeOverflow* subfolder, and open and run the project *typeOverflow.xcode*.

Keep an eye on the numbers as they scroll by on the screen. When the number 255 appears, a funny thing happens. The next number will be 0, then 1, 2, etc. If you leave the program running for a while it will climb back up to 255, then jump to 0 and climb back up again. This will continue forever. You'll also likely see a warning in the build window complaining that the *comparison is always true due to limited range of data type*.

Click on the red stop sign icon in the upper-right corner of the run window (or type option-command-R) to stop the program.

The problem with this program occurs when the **`for`** loop increments **`counter`** when it has a value of 255. Since an **`unsigned char`** can hold a maximum value of 255, incrementing it gives it a value of 0 again. Since **`counter`** can never get higher than 255, the **`for`** loop never exits.

Just for kicks, edit the code and change the **`unsigned char`** to a **`signed char`**. What do you think will happen? Try it!

The real solution here is to use a **`short`**, **`int`**, or **`long`** instead of a **`char`**. Don't be stingy. Unless there is a real reason to worry about memory usage, err on the side of extravagance. Err on the side of safety!

Working With Characters

With its minimal range, you might think that a **`char`** isn't good for much. Actually, the C deities created the **`char`** for a good reason. It is the perfect size to hold a single alphabetic character. In C, an alphabetic character is a single character placed between a pair of single quotes (**`'`**). Here's a test to see if a **`char`** variable contains the letter **`'a'`**:

```
char c;

c = 'a';

if ( c == 'a' )
 printf( "The variable c holds the character
 'a'." );
```

As you can see, the character **`'a'`** is used in both an assignment statement and an **`if`** statement, just as if it were a number or a variable.

The ASCII Character Set

In C, a **`signed char`** takes up a single byte and can hold a value from -128 to 127. Now, how can a **`char`** hold a numerical value, as well as a character value, such as **`'a'`** or **`'+'`**? The answer lies with the **ASCII character set**.

ASCII stands for the American Standard Code for Information Interchange.

The ASCII character set is a set of 128 standard characters, featuring the 26 lower-case letters, the 26 upper-case letters, the ten numerical digits, and an assortment of other exciting characters, such as **`'}'`** and **`'='`**. Each of these characters corresponds exactly to a value between 0 and 127. The ASCII character set ignores the values between -128 and -1.

For example, the character **`'a'`** has an ASCII value of 97. When a C compiler sees the character **`'a'`** in a piece of source code, it substitutes the value 97. Each of the values from 0 to 127 is interchangeable with a character from the ASCII character set.

Though we make use of the ASCII character set throughout this book, you should know that there are other character sets out there. Some foreign alphabets have more characters than can be represented by a single byte. To accommodate these multibyte characters, ISO C features **wide character** and **wide string** data types.

Though we won't get into multibyte character sets in this book, you should keep these things in mind as you write your own code. Read up on the multibyte extensions introduced as part of the ISO C standard. There's an excellent writeup in Harbison and Steele's *C, A Reference Manual*. The 5th edition was released in 2002. A terrific C reference, well worth the purchase price.

Here's an article whose title tells it all: *The Absolute Minumum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*, by Joel Spolsky:

http://joelonsoftware.com/articles/Unicode.html

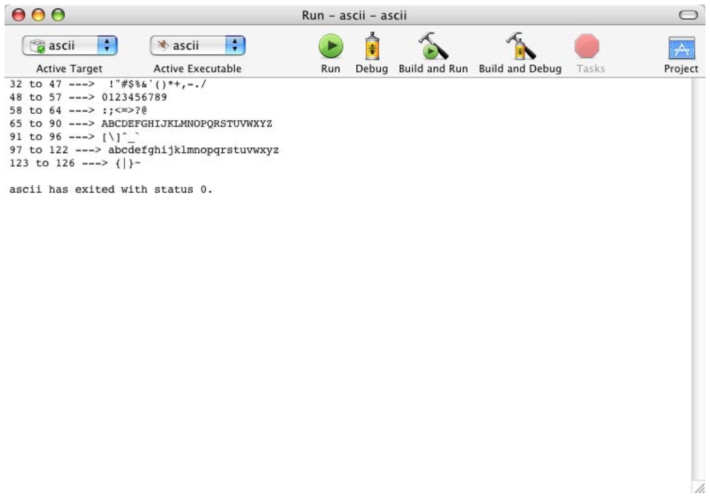
Rock on, Joel!

ascii.xcode

Here's a program that will make the ASCII character set easier to understand. Go into the *Learn C Projects* folder, then into the *08.04 - ascii* subfolder and open the project *ascii.xcode*.

Before we step through the project source code, let's take it for a spin. Select Build and Run from the

Build menu. A console window similar to the one in Figure 8.2 should appear. The first line of output shows the characters corresponding to the ASCII values from 32 to 47. Why start with 32? As it turns out, the ASCII characters between 0 and 31 are nonprintable characters like the backspace (ASCII 8) or the carriage return (ASCII 13). A table of the nonprintable ASCII characters is presented later on.



```
32 to 47 --->  !"#$%&'()*+,-./
48 to 57 ---> 0123456789
58 to 64 ---> :;<=>?@
65 to 90 ---> ABCDEFGHIJKLMNOPQRSTUVWXYZ
91 to 96 ---> [\]^_`
97 to 122 ---> abcdefghijklmnopqrstuvwxyz
123 to 126 ---> {|}~

ascii has exited with status 0.
```

Figure 8.2 *The printable ASCII characters.*

Notice that ASCII character 32 is a space, also known as `' '`. ASCII character 33 is `'!'`. ASCII character 47 is `'/'`. This presents some interesting coding possibilities. For example, this code is perfectly legitimate:

```
int  sumOfChars;

sumOfChars = '!' + '/';
```

What a strange piece of code! Though you will probably never do anything like this, try to predict the value of the variable **`sumOfChars`** after the assignment statement. And the answer is...

The character **`'!'`** has a value of 33 and the character **`'/'`** has a value of 47. Therefore, **`sumOfChars`** will be left with a value of 80 following the assignment statement. C allows you to represent any number between 0 and 127 in two different ways: as an ASCII character or as a number. Let's get back to the console window in Figure 8.2.

The second line of output shows the ASCII characters from 48 through 57. As you can see, these ten characters represent the digits 0 through 9. Here's a little piece of code that converts an ASCII digit to its numerical counterpart:

```
char digit;
int  convertedDigit;

digit = '3';

convertedDigit = digit - '0';
```

This code starts with a **`char`** named **`digit`**, initialized to hold the ASCII character `'3'`. The character `'3'` has a numerical value of 51. The next line of code subtracts the ASCII character `'0'` from `digit`. Since the character `'0'` has a numerical value of 48, and `digit` started with a numerical value of 51, `convertedDigit` ends up with a value of 51 - 48, also known as 3. Isn't that interesting?

Subtracting `'0'` from any ASCII digit yields that digit's numerical counterpart. Though this is a great trick if you know you're working with ASCII, your code will fail if the digits of the current character set are not represented in the same way as they are in ASCII. For example, if you were on a machine that used a character set where the digits were sequenced from 1 to 9, followed by 0, the above trick wouldn't work.

The next line of the console window shown in Figure 8.2 shows the ASCII characters with values ranging from 58 to 64. The following line is pretty interesting. It shows the range of ASCII characters from 65 to 90. Notice anything familiar about these characters? They represent the complete, upper-case alphabet.

The next line in Figure 8.2 lists ASCII characters with values from 91 through 96. The following line lists the ASCII characters with values ranging from 97 through 122. These 26 characters represent the complete lower-case alphabet.

Adding 32 to an upper-case ASCII character yields its lower-case equivalent. Likewise, subtracting 32 from a lower-case ASCII character yields its upper-case equivalent.

Guess what? You never want to take advantage of this information! Instead, use the Standard Library routines `tolower()` and `toupper()` to do the conversions for you.

As a general rule, try not to make assumptions about the order of characters in the current character set. Use Standard Library functions rather than working directly with character values. Though it is tempting to do these kinds of conversions yourself, by going through the Standard Library you know your program will work across single byte character sets.

The final line in Figure 8.2 lists the ASCII characters from 123 to 126. As it turns out, the ASCII character with a value of 127 is another nonprintable character. Figure 8.3 shows a table of these "unprintables." The left column shows the ASCII code. The right column shows the keyboard equivalent for that code along with any appropriate comments. The characters with comments by them are probably the only unprintables you'll ever make use of.

ASCII Unprintables	
0	Used to terminate text strings (Explained later in chapter)
1	Control-A
2	Control-B
3	Control-C
4	Control-D (End of file mark, see Chapter 10)
5	Control-E
6	Control-F
7	Control-G (Beep - works in Terminal, not in Xcode)
8	Control-H (Backspace)
9	Control-I (Tab)
10	Control-J (Line feed)
11	Control-K (Vertical feed)
12	Control-L (Form feed)
13	Control-M (Carriage return, no line feed)
14	Control-N
15	Control-O
16	Control-P
17	Control-Q
18	Control-R
19	Control-S
20	Control-T
21	Control-U
22	Control-V
23	Control-W
24	Control-X
25	Control-Y
26	Control-Z
27	Control-[(Escape character)
28	Control-|
29	Control-]
30	Control-^
31	Control-_
127	del

Figure 8.3 *The ASCII unprintables.*

Stepping Through the Source Code

Before we move on to our next topic, let's take a look at the source code that generated the ASCII character listing in Figure 8.2. *main.c* starts off with the usual **`#include`** and follows it by a function prototype of the function **`PrintChars()`**. **`PrintChars()`** takes two parameters which define a range of **`char`**s to print.

```
#include <stdio.h>

void PrintChars( char low, char high );
```

`main()` calls **`PrintChars()`** 7 times in an attempt to functionally organize the ASCII characters.

```
int main (int argc, const char * argv[])
{
 PrintChars( 32, 47 );
 PrintChars( 48, 57 );
 PrintChars( 58, 64 );
 PrintChars( 65, 90 );
 PrintChars( 91, 96 );
 PrintChars( 97, 122 );
 PrintChars( 123, 126 );

 return 0;
}
```

`PrintChars()` declares a local variable, **`c`**, to act as a counter as we step through a range of **`char`**s.

```
void PrintChars( char low, char high )
{
 char      c;
```

We'll use **`low`** and **`high`** to print a label for the current line, showing the range of ASCII characters to follow. Notice that we use **`%d`** to print the integer version of these **`char`**s. **`%d`** can handle any integer types no bigger than an **`int`**.

```
   printf( "%d to %d ---> ", low, high );
```

Next, a **`for`** loop is used to step through each of the ASCII characters, from **`low`** to **`high`**, using **`printf()`** to print each of the characters next to each other on the same line. The **`printf()`** bears closer inspection. Notice the use of **`%c`** (instead of our usual **`%d`**) to tell **`printf()`** to print a single ASCII character.

```
   for ( c = low; c <= high; c++ )
      printf( "%c", c );
```

Once the line is printed, a single new line is printed, moving the cursor to the beginning of the next line in the console window. Thus ends **`PrintChars()`**.

```
   printf( "\n" );
  }
```

The **`char`** data type is extremely useful to C programmers (such as yourself). The next two topics, arrays and text strings, will show you why. As you read through these two topics, keep the concept of ASCII characters in the back of your mind. As you reach the end of the section on text strings, you'll see an important relationship develop between all three topics.

Arrays

The next topic for discussion is **arrays**. An array turns a single variable into a list of variables. For example, this declaration:

```
int  myNumber[ 3 ];
```

creates three separate **`int`** variables, referred to in your program as **`myNumber[ 0 ]`**, **`myNumber[ 1 ]`**, and **`myNumber[ 2 ]`**. Each of these variables is known as an **array element**. The number between the brackets (**`[`** and **`]`** are known as brackets or square brackets) is called an **index**. In this declaration:

```
char myChar[ 20 ];
```

the name of the array is **`myChar`**. This declaration will create an array of type **`char`** with a **dimension** of 20. The dimension of an array is the array's number of elements. The array elements will have **indices** (indices, indexes, we're talking more than one index here) that run from 0 to 19.

In C, array indices always run from 0 to one less than the array's dimension.

This slice of code first declares an array of 100 **`int`**s, then assigns each **`int`** a value of 0:

```
int  myNumber[ 100 ], i;

for ( i=0; i<100; i++ )
 myNumber[ i ] = 0;
```

You could have accomplished the same thing by declaring 100 individual **`int`**s, then initializing each individual **`int`**. Here's what that code might look like:

```
int  myNumber0, myNumber1, ......., myNumber99;

myNumber0 = 0;
myNumber1 = 0;
      .
      .
      .
myNumber99 = 0;
```

It would take 100 lines of code just to initialize these variables! By using an array, we've accomplished the same thing in just a few lines of code. Look at this code fragment:

```
sum = 0;
for ( i=0; i<100; i++ )
 sum += myNumber[ i ];

printf( "The sum of the 100 numbers is %d.",
 sum );
```

This code adds together the value of all 100 elements of the array **`myNumber`**.

> In this example, the **`for`** loop is used to **step through** an array, performing some operation on each of the array's elements. You'll use this technique frequently in your own C programs.

Why Use Arrays?

Programmers would be lost without arrays. Arrays allow you to keep lists of things. For example, if you need to maintain a list of 50 employee numbers, declare an array of 50 **`int`**s. You can declare an array using any C type. For example, this code:

```
float salaries[ 50 ];
```

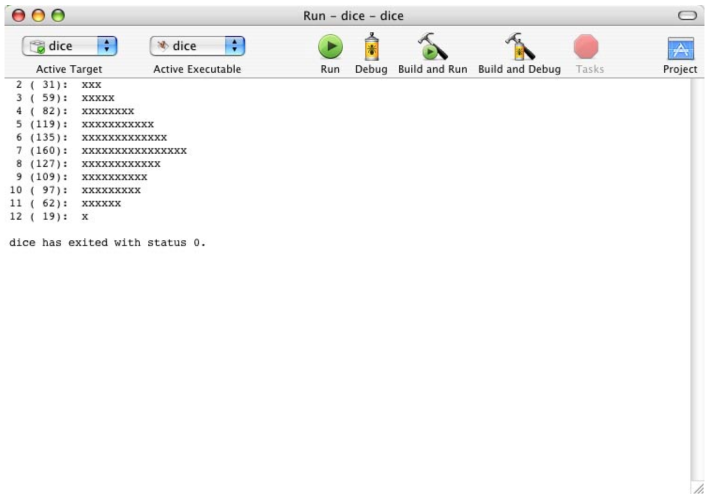
declares an array of 50 floating-point numbers. This might be useful for maintaining a list of employee salaries.

Use an array when you want to maintain a list of related data. Here's an example.

dice.xcode

Look in the *Learn C Projects* folder, inside the *08.05 - dice* subfolder, and open the project *dice.xcode*. **`dice`** simulates the rolling of a pair of dice. After each roll, the program adds the two dice together, keeping track of the total. It rolls the dice 1,000 times, then reports on the results. Give it a try!

Run **`dice`** by selecting Build and Run from the Build menu. A console window should appear, similar to the one in Figure 8.4. Take a look at the output—it's pretty interesting. The first column lists all the possible totals of two dice. Since the lowest possible roll of a pair of six-sided dice is 1 and 1, the first entry in the column is 2. The column counts all the way up to 12, the highest possible roll (achieved by a roll of 6 and 6).



```
 2 ( 31):  xxx
 3 ( 59):  xxxxx
 4 ( 82):  xxxxxxxx
 5 (119):  xxxxxxxxxxx
 6 (135):  xxxxxxxxxxxxx
 7 (160):  xxxxxxxxxxxxxxxx
 8 (127):  xxxxxxxxxxxx
 9 (109):  xxxxxxxxxx
10 ( 97):  xxxxxxxxx
11 ( 62):  xxxxxx
12 ( 19):  x

dice has exited with status 0.
```

Figure 8.4 **`dice`** *in action. Your mileage may vary!*

The number in parentheses is the total number of rolls (out of 1,000 rolls) that matched that row's number. For example, the first row describes the dice rolls that total 2. In this run, the program rolled 31

2's. Finally, the program prints an **x** for every ten of these rolls. Since 31 2's were rolled, three **x**'s were printed at the end of the 2's row. Since 160 7's were rolled, 16 **x**'s were printed at the end of the 7's row.

Recognize the curve depicted by the **x**'s in Figure 8.4? The curve represents a "normal" probability distribution, also known as a bell curve. According to the curve, you are about 8.4 times more likely to roll a 7 as you are to roll a 12. Want to know why? Check out a book on probability and statistics.

Let's take a look at the source code that makes this possible.

Stepping Through the Source Code

main.c starts off with three **#include**s. **<stdlib.h>** gives us access to the routines **rand()** and **srand()**, **<time.h>** gives us access to **clock()**, and **<stdio.h>** gives us access to **printf()**.

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
```

Here are the function prototypes for **RollOne()**, **PrintRolls()**, and **PrintX()**. You'll see how these routines work as we walk through the code.

```
int  RollOne( void );
void PrintRolls( int rolls[] );
void PrintX( int howMany );
```

main() declares an array of 13 **int**s named **rolls**. **rolls** will keep track of the 11 possible types of dice rolls. **rolls[2]** will keep track of the total number of 2's, **rolls[3]** will keep track of the total number of 3's, and so on, up until **rolls[12]** which will keep track of the total number of 12's rolled. Since there is no way to roll a 0 or a 1 with a pair of dice, **rolls[0]** and **rolls[1]** will go unused.

```
int main (int argc, const char * argv[])
{
 int       rolls[ 13 ], twoDice, i;
```

We could have rewritten the program using an array of 11 **int**s, thereby saving 2 **int**s worth of memory. If we did that, **rolls[0]** would track the number of 2's rolled, **rolls[1]** would track the number of 3's rolled, etc. This would have made the program a little harder to read, since **rolls[i]** would be referring to the number of **(i+2)**'s rolled.

In general, it is OK to sacrifice memory to make your program easier to read, as long as program performance isn't compromised.

The function **srand()** is part of the Standard Library. It initializes a random number generator, using a seed provided by another Standard Library function, **`clock()`**. Once the random number generator is initialized, another function, **`rand()`**, will return an **`int`** with a random value.

```
srand( clock() );
```

Why random numbers? Sometimes you want to add an element of unpredictability to your program. For example, in our program, we want to roll a pair of dice again and again. The program would be pretty boring if it rolled the same numbers over and over. By using a random number generator, we can generate a random number between 1 and 6, thus simulating the roll of a single die!

`main()`'s next step is to initialize each of the elements of the array **`rolls`** to 0. This is appropriate since no rolls of any kind have taken place yet.

```
for ( i=0; i<=12; i++ )
    rolls[ i ] = 0;
```

Now comes Miller time! This **`for`** loop rolls the dice 1,000 times. As you'll see, the function **`RollOne()`** returns a random number between 1 and 6, simulating the roll of a single die. By calling it twice, then storing the sum of the two rolls in the variable **`twoDice`**, we've simulated the roll of two dice.

```
  for ( i=1; i <= 1000; i++ )
  {
    twoDice = RollOne() + RollOne();
```

The next line is pretty tricky, so hang on. At this point, the variable **`twoDice`** holds a value between 2 and 12, the total of two individual dice rolls. We'll use that value to specify which of the **`rolls`**' **`int`**s to increment. If **`twoDice`** is 12 (if we rolled a pair of sixes) we'll increment **`rolls[12]`**. Get it? If not, go back and read through this again. If you still feel stymied (and it's OK if you do) find a C buddy to help you through this. It is important that you get this concept. Be patient.

```
    ++ rolls[ twoDice ];
  }
```

Once we're finished with our 1,000 rolls, we'll pass **`rolls`** as a parameter to **`PrintRolls()`**.

```
  PrintRolls( rolls );

  return 0;
}
```

Notice that we used the array name, without the brackets (**`rolls`** instead of **`rolls[]`**). The name of an array is a pointer to the first element of the array. If you have access to this pointer, you have access to the entire array. You'll see how this works when we look at **`PrintRolls()`**.

Just remember that passing the name of an array as a parameter is exactly the same as passing a pointer to the first element of the array. To prove this, edit *main.c* and change this line of code:

```
PrintRolls( rolls );
```

to

```
PrintRolls( &( rolls[0] ) );
```

These two lines are exactly equivalent! The second form passes the address of the first array element. If you think back to our last chapter, we use the **&** operator to pass a parameter by reference instead of by value. By passing the address of the first array element, you give **`PrintRolls()`** the ability to both access and modify all of the array elements. This is an important concept!

`RollOne()` first calls **`rand()`** to generate a random number, ranging from 0 to 32,767 (actually, the upper bound is defined by the constant **`RAND_MAX`**, which is guaranteed to be at least 32,767). Next, the **`%`** operator is used to return the remainder when the random number is divided by 6. This yields a random number ranging from 0 to 5. Finally, 1 is added to this number, converting it to a number between 1 and 6, and that number is returned.

```
int  RollOne( void )
{
 return (rand() % 6) + 1;
}
```

`PrintRolls()` starts off by declaring a single parameter, an array pointer named **`rolls`**. Notice that **`rolls`** was declared using square brackets, telling the compiler that **`rolls`** is a pointer to the first element of an array (in this case, to an array of **`int`**s).

```
void PrintRolls( int rolls[] )
{
 int       i;
```

`PrintRolls()` could also have declared its parameter using this notation:

```
void   PrintRolls( int *rolls )
```

instead of:

```
void   PrintRolls( int rolls[] )
```

Both of these notations describe a pointer to an **`int`**, and both can be used to access the elements of an array. You'll learn more about the close relationship between pointers and arrays as you make your way through the rest of the book.

For now, remember this convention. If you are declaring a parameter that will point to an array, use the square bracket form. Otherwise, use the normal pointer form.

Let's get back to our program. Before the previous tech block, we had just started looking at **`PrintRolls()`**. The **`for`** loop steps through the **`rolls`** array, one **`int`** at a time, starting with **`rolls[2]`** and making its way to **`rolls[12]`**. For each element, **`PrintRolls()`** first prints the roll number and then, in parentheses, the number of times (out of 1,000) that roll occurred. Next, **`PrintX()`** is called to print a single **`x`** for every ten rolls that occurred. Finally, a carriage return is printed, preparing the console window for the next roll.

```
for ( i=2; i<=12; i++ )
{
    printf( "%2d (%3d):  ", i, rolls[ i ] );
    PrintX( rolls[ i ] / 10 );
    printf( "\n" );
}
}
```

`PrintX()` is pretty straightforward. It uses a **`for`** loop to print the number of **`x`**'s specified by the parameter **`howMany`**.

```
void PrintX( int   howMany )
{
 int i;

 for ( i=1; i<=howMany; i++ )
    printf( "x" );
}
```

Danger, Will Robinson!!!

Before we move on to our next topic, there is one danger worth discussing at this point. See if you can spot the potential hazard in this piece of code:

```
int  myInts[ 3 ];

for ( i=0; i<20; i++ )
 myInts[ i ] = 0;
```

Yikes! The array **`myInts`** consists of exactly three array elements, yet the **`for`** loop tries to initialize 20 elements. This is called **exceeding the bounds** of your array. Because C is such an informal language, it will let you "get away" with this kind of source code. To you, that means Xcode will compile this code without complaint. Your problems will start as soon as the program tries to initialize the fourth array element, which was never allocated.

What will happen? The safest thing to say is that the results will be unpredictable. The problem is, the program is trying to assign a value of 0 to a block of memory that it doesn't necessarily own. Anything could happen. The program would most likely crash, which means it stops behaving in a rational manner. I've seen some cases where the computer actually leaps off the desk, hops across the floor, and jumps face first into the trash can.

Well, OK, not really. Modern operating systems protect the boundaries of individual applications to protect one application from crashing another. But odd things will happen if you don't keep your array references in bounds.

As you code, be aware of the limitations of your variables. For example, a **`char`** is limited to values from -128 to 127. Don't try to assign a value such as 536 to a **`char`**. Don't reference **`myArray[ 27 ]`** if you declared **`myArray`** with only ten elements. Be careful!

Text Strings

The first C program in this book made use of a text string:

```
  printf( "Hello, world!" );
```

This section will teach you how to use text strings like **"Hello, world!"** in your own programs. It will teach you how these strings are stored in memory and how to create your own strings from scratch.

A Text String in Memory

The text string **"Hello, world!"** exists in memory as a sequence of 14 bytes (Figure 8.5). The first 13 bytes consist of the 13 ASCII characters in **"Hello, world!"**. Note that the seventh byte contains a space (on an ASCII-centric computer, that translates to a value of 32).

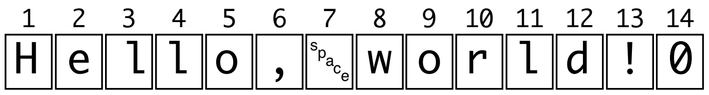


Figure 8.5 *The* `"Hello, World!"` *text string. Don't forget, byte 14 contains a zero.*

The final byte (byte 14) has a value of zero, not to be confused with the ASCII character **'0'**. The zero is what makes this string a C string. Every C string ends with a byte having a value of 0. The 0 identifies the end of the string.

When you use a quoted string like **"Hello, world!"** in your code, the compiler creates the string for you. This type of string is called a **string constant**. When you use a string constant in your code, the detail work is done for you automatically. In this example:

```
  printf( "Hello, world!" );
```

the 14 bytes needed to represent the string in memory are allocated automatically. The 0 is placed in the fourteenth byte, automatically. You don't have to worry about these details when you use a string constant.

String constants are great, but they are not always appropriate. For example, suppose you want to read in somebody's name, then pass the name on to **printf()** to display in the console window. Since you won't be able to predict the name that will be typed in, you can't predefine the name as a string constant. Here's an example.

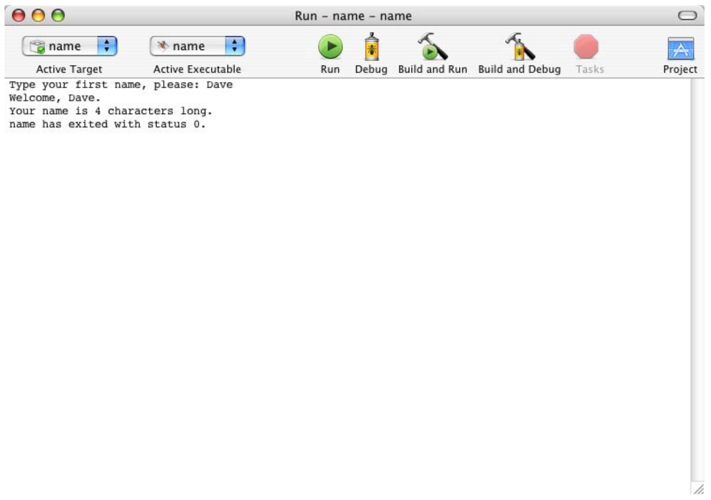
name.xcode

Look in the *Learn C Projects* folder, inside the *08.06 - name* subfolder, and open the project *name.xcode*. **name** will ask you to type your first name on the keyboard. Once you've typed your first name, the program will use your name to create a custom welcome message. Then, **`name`** will tell you how many characters long your name is. How useful!

To run **`name`**, select Build and Run from the Build menu. A console window will appear, prompting you for your first name, like this:

```
Type your first name, please:
```

Type your first name, then hit a carriage return. When I did, I saw the output shown in Figure 8.6. Let's take a look at the source code that generated this output.



```
Type your first name, please: Dave
Welcome, Dave.
Your name is 4 characters long.
name has exited with status 0.
```

Figure 8.6 **`name`** *prompts you to type in your name, then tells you how long your name is.*

Stepping Through the Source Code

At the heart of *main.c* is a new Standard Library function called **`scanf()`**. **`scanf()`** uses the same format specifiers as **`printf()`** to read text in from the keyboard. This code will read in an **`int`**:

```
int  myInt;

scanf( "%d", &myInt );
```

The **`%d`** tells **`scanf()`** to read in an **`int`**. Notice the use of the **`&`** before the variable **`myInt`**. This passes **`myInt`**'s address to **`scanf()`**, allowing **`scanf()`** to change **`myInt`**'s value. To read in a float, use code like:

```
float myFloat;

scanf( "%f", &myFloat );
```

main.c starts off with a pair of **`#include`**s. **`<string.h>`** gives us access to the Standard Library function **`strlen()`**, and **`<stdio.h>`**, well, you know what we get from **`<stdio.h>`**. **`printf()`**, right? Right.

```
#include <string.h>
#include <stdio.h>
```

To read in a text string, you have to first declare a variable to place the text characters in. *main.c* uses an array of characters for this purpose:

```
int main (int argc, const char * argv[])
{
 char      name[ 50 ];
```

The array **`name`** is big enough to hold a 49-byte text string. When you allocate space for a text string, remember to save 1 byte for the **`0`** that terminates the string.

The program starts by printing a **prompt**. A prompt is a text string that lets the user know the program is waiting for input.

```
printf( "Type your first name, please: " );
```

The Input Buffer

Before we get to the **`scanf()`** call, it helps to understand how the computer handles input from the keyboard. When the computer starts running your program, it automatically creates a big array of **`char`**s for the sole purpose of storing keyboard input to your program. This array is known as your program's **input buffer**. The input buffer is carriage-return based. Every time you hit a carriage return, all the characters typed since the last carriage return are appended to the current input buffer.

When your program starts, the input buffer is empty. If you type this line from your keyboard:

```
123 abcd
```

followed by a carriage return, the input buffer will look like Figure 8.7. The computer keeps track of the current end of the input buffer. The space character between the **`'123'`** and the **`'abcd'`** has an ASCII value of 32. Notice that the carriage return was actually placed in the input buffer.

The ASCII value of the character used to indicate a carriage return is implementation dependent. In most consoles, an ASCII 10 indicates a carriage return. On some, an ASCII 13 indicates a carriage return. Use the **'\n'** character and you'll always be safe.

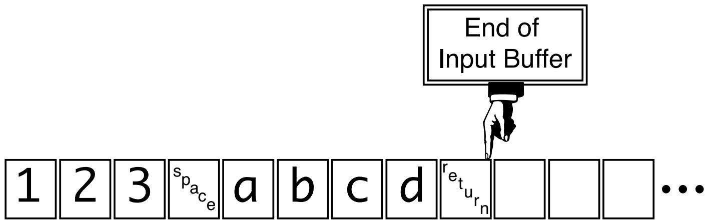


Figure 8.7 *A snapshot of the input buffer.*

Given the input buffer shown in Figure 8.7, suppose your program called **scanf()**, like this:

```
scanf( "%d", &myInt );
```

scanf() starts at the beginning of the input buffer and reads a character at a time until it hits one of the nonprintables; that is, a carriage return, tab, space, or a **0**, until it hits the end of the buffer, or until it hits a character that conflicts with the format specifier (if **%d** was used and the letter **'a'** was encountered, for example).

After the **scanf()**, the input buffer looks like Figure 8.8. Notice that the characters passed on to **scanf()** were removed from the input buffer and that the rest of the characters slid over to the beginning of the buffer. **scanf()** took the characters **'1'**, **'2'**, and **'3'** and converted them to the integer 123, placing 123 in the variable **myInt**.

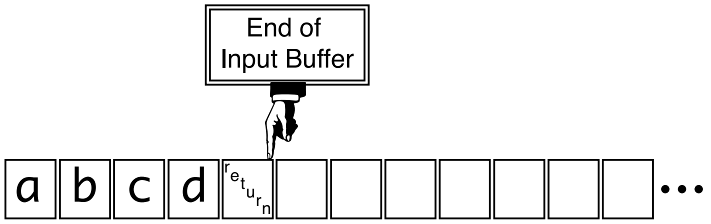


Figure 8.8 *A second snapshot of the input buffer.*

If you then typed the line:

```
3.5 Dave
```

followed by a carriage return, the input buffer would look like Figure 8.9. At this point the input buffer contains two carriage returns. To the input buffer, a carriage return is just like any other character. To a function like **scanf()**, the carriage return is white space.

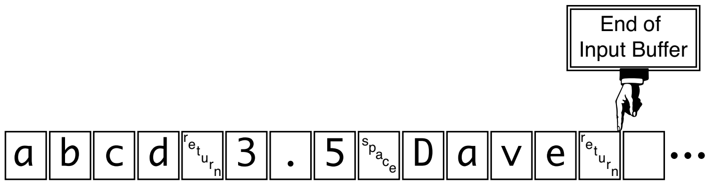


Figure 8.9 A third snapshot of the input buffer.

If you forgot what white space is, now would be a good time to turn back to Chapter 5, where white space was first described.

On With the Program

Before we started our discussion on the input buffer, **`main()`** had just called **`printf()`** to prompt the user for his or her first name:

```
printf( "Type your first name, please: " );
```

Next, we called **`scanf()`** to read the first name from the input buffer:

```
scanf( "%s", name );
```

Since the program just started, the input buffer is empty. **`scanf()`** will wait until characters appear in the input buffer, which will happen as soon as you type some characters and hit a carriage return. Type your first name and hit a carriage return.

`scanf()` will ignore white space characters in the input buffer. For example, if you type a few spaces and tabs, then hit a carriage return, **`scanf()`** will still sit there, waiting for some real input. Try it!

Once you type in your name, **`scanf()`** will copy the characters, a byte at a time, into the array of **`char`**s pointed to by **`name`**. Remember, because **`name`** was declared as an array, **`name`** points to the first of the 50 bytes allocated for the array.

If you type in the name **`Dave`**, **`scanf()`** will place the four characters **`'D'`**, **`'a'`**, **`'v'`**, and **`'e'`** in the first four of the 50 bytes allocated for the array. Next, **`scanf()`** will set the fifth byte to a value of **`0`** to terminate the string properly (Figure 8.10). Since the string is properly terminated by the **`0`** in **`name[4]`**, we don't really care about the value of the bytes **`name[5]`** through **`name[49]`**.

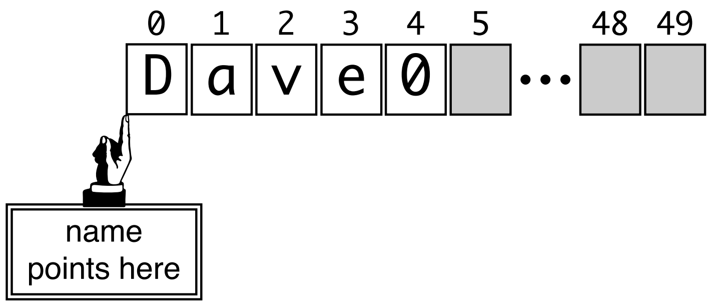


Figure 8.10 The array **name** *after the string* **"Dave"** *is copied to it. Notice that* **name[4]** *has a value of zero.*

Next, we pass **name** on to **printf()**, asking it to print the name as part of a welcoming message. The **%s** tells **printf()** that **name** points to the first byte of a zero-terminated string. **printf()** will step through memory, one byte at a time, starting with the byte that **name** points to. **printf()** will print each byte in turn until it hits a byte with a value of zero. The zero byte marks the end of the string.

```
    printf( "Welcome, %s.\n", name );
```

If **name[4]** didn't contain a **0**, the string wouldn't be properly terminated. Passing a non-terminated string to **printf()** is a sure way to confuse **printf()**. **printf()** will step through memory one byte at a time, printing a byte and looking for a **0**. It will keep printing bytes until it happens to encounter a byte set to **0**. Remember, C strings must be terminated!

The next line of the program calls another Standard Library function, called **strlen()**. **strlen()** takes a pointer as a parameter and returns the length, in bytes, of the string pointed to by the parameter. **strlen()** depends on the string being **0** terminated. Just like **sizeof()**, **strlen()** returns a value of type **size_t**. We'll use a typecast to convert the value to an **int**, then print it using **%d**. Again, we'll cover typecasting later in the book.

```
    printf( "Your name is %d characters long.",
    (int)strlen( name ) );

    return 0;
  }
```

Our last program for this chapter demonstrates a few more character-handling techniques, a new Standard Library function, and an invaluable programmer's tool, the **#define**.

The #define

The **`#define`** (pronounced *pound-define*) tells the compiler to substitute one piece of text for another throughout your source code. This statement:

```
#define kMaxPlayers      6
```

tells the compiler to substitute the character “**`6`**” every time it finds the text “**`kMaxPlayers`**” in the source code. **`kMaxPlayers`** is known as a **macro**. As the C compiler goes through your code, it enters all the **`#define`**s into a list, known as a **dictionary**, performing all the **`#define`** substitutions as it goes.

It’s important to note that the compiler never actually modifies your source code. The dictionary it creates as it goes through your code is separate from your source code and the substitutions it performs are made as the source code is translated into machine code.

Here’s an example of a **`#define`** in action:

```
#define kMaxArraySize    100

int main (int argc, const char * argv[])
{
  char      myArray[ kMaxArraySize ];
  int       i;

  for ( i=0; i<kMaxArraySize; i++ )
      myArray[ i ] = 0;

  return 0;
}
```

The **`#define`** at the beginning of this example substitutes “**`100`**” for “**`kMaxArraySize`**” everywhere it finds it in the source code file. In this example, the substitution will be done twice. Though your source code is not actually modified, here’s the effect of this **`#define`**:

```
int main (int argc, const char * argv[])
{
  char      myArray[ 100 ];
  int       i;

  for ( i=0; i<100; i++ )
      myArray[ i ] = 0;

  return 0;
}
```

Note that a **#define** must appear in the source code file before it is used. In other words, this code won't compile:

```
int main (int argc, const char *
argv[])
{
  char   myArray[ kMaxArraySize ];
  int    i;
#define kMaxArraySize   100
  for ( i=0; i<kMaxArraySize; i++ )
      myArray[ i ] = 0;
  return 0;
}
```

Having a **#define** in the middle of your code is just fine. The problem here is that the declaration of **myArray** uses a **#define** that hasn't occurred yet!

If you use **#define**s effectively, you'll build more flexible code. In the previous example, you can change the size of the array by modifying a single line of code, the **#define**. If your program is designed correctly, you should be able to change the line to:

```
#define kMaxArraySize    200
```

then recompile your code, and your program should still work properly. A good sign that you are using **#define**s properly is an absence of constants in your code. In the above example, the constant 100 was replaced by **kMaxArraySize**.

Many programmers use the same naming convention for **#define**s as they use for global variables. Instead of starting the name with a **g** (as in **gMyGlobal**), a **#define** constant starts with a **k** (as in **kMyConstant**).

Unix programmers tend to name their **#define** constants using all upper case letters, sprinkled with underscores "_" to act as word dividers (as in **MAX_ARRAY_SIZE**).

As you'll see in our next program, you can put practically anything, even source code, into a **#define**. Take a look:

```
#define kPrintReturn     printf( "\n" );
```

While not particularly recommended, this **#define** will work just fine, substituting the statement:

```
printf( "\n" );
```

for every occurrence of the text **kPrintReturn** in your source code. You can base one **#define** on a previous **#define**:

```
#define kSideLength      5
#define kArea      kSideLength * kSideLength
```

Interestingly, you could have reversed the order of these two **#define**s, and your code would still have compiled. As long as both entries are in the dictionary, their order of occurrence in the dictionary is not important.

What is important is that **#define** appear in the source code before any source code that refers to it.

If this seems confusing, don't sweat it. It won't be on the test.

Function-Like #define Macros

You can create a **#define** macro that takes one or more arguments. Here's an example:

```
#define kSquare( a )      ((a) * (a))
```

This macro takes a single argument. The argument can be any C expression. If you called the macro like this:

```
myInt = kSquare( myInt + 1 );
```

the compiler would use its first pass to turn the line into this:

```
myInt = (( myInt + 1 ) * ( myInt + 1 ));
```

Notice the usefulness of the parentheses in the macro. If the macro were defined like this:

```
#define kSquare( a )      a * a
```

the compiler would have produced:

```
myInt = myInt + 1 * myInt + 1;
```

which is not what we wanted. The only multiplication that gets performed by this statement is **1 * myInt**, because the ***** operator has a higher precedence than the **+** operator.

Be sure you pay strict attention to your use of white space in your **#define** macros. For example, there's a world of difference between this macro:

```
#define kSquare( a )      ((a) * (a))
```

and this macro (note the space between **kSquare** and **(a)**:

```
#define kSquare ( a )     ((a) * (a))
```

This second form creates a **`#define`** constant named **`kSquare`** which is defined as "**`( a ) ((a) * (a))`**". A call to this macro won't even compile because the compiler doesn't know what "**`a`**" is.

Here's another interesting macro side-effect. Imagine calling this macro:

```
#define kSquare( a )      ((a) * (a))
```

like this:

```
mySquare = kSquare( myInt++ );
```

The preprocessor pass expands this macro call to:

```
mySquare = ((myInt++) * (myInt++));
```

Do you see the problems here? First off, **`myInt`** will get incremented twice by this macro call (probably not what was intended). Secondly, the first **`myInt++`** will get executed before the multiply happens, yielding a final result of **`myInt*(myInt+1)`**, definitely not what you wanted! The point here: Be careful when you pass an expression as a parameter to a macro.

Let's move on to our final example.

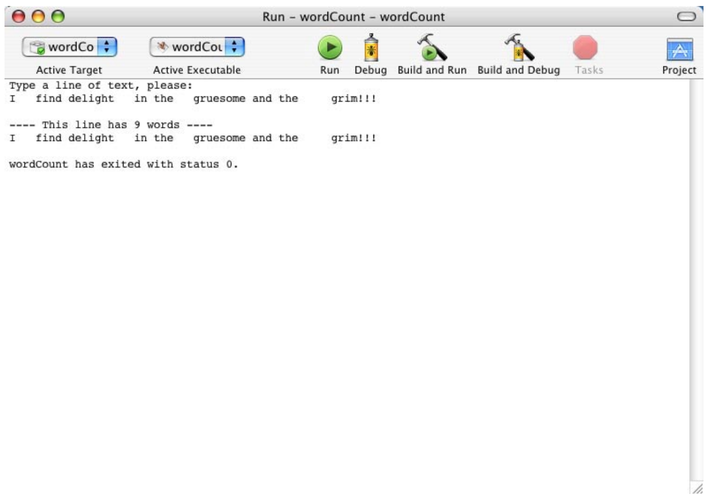
wordCount.xcode

Look in the *Learn C Projects* folder, inside the *08.07 - wordCount* subfolder, and open the project *wordCount.xcode*. **`wordCount`** will ask you to type in a line of text and will count the number of words in the text you type.

To run **`wordCount`**, select Build and Run from the Build menu. **`wordCount`** will prompt you to type in a line of text:

```
Type a line of text, please:
```

Type in a line of text, at least a few words long. End your line by typing a carriage return. When you hit the return, **`wordCount`** will report its results. **`wordCount`** will ignore any white space, so feel free to sprinkle your input with tabs, spaces, and the like. My output is shown in Figure 8.11. Let's take a look at the source code that generated this output.



```
Type a line of text, please:
I   find delight   in the   gruesome and the     grim!!!

---- This line has 9 words ----
I   find delight   in the   gruesome and the     grim!!!

wordCount has exited with status 0.
```

Figure 8.11 *`wordCount`, doing its job.*

Stepping Through the Source Code

main.c starts off with the usual **`#include`**s, and then adds a new one. **`<ctype.h>`** includes the prototype of the function **`isspace()`**, which takes a **`char`** as input and returns **`true`** if the **`char`** is either a tab (**`'\t'`**), hard carriage return (return without a line feed - **`'\r'`**), newline (return with a line feed - **`'\n'`**), vertical tab (**`'\v'`**), form feed (**`'\f'`**), or space (**`' '`**), and returns **`false`** otherwise.

```
#include <stdio.h>
#include <c.h>
#include <ctype.h>
```

Older C environments may include a variant of **`isspace()`** called **`iswhite()`**.

Next, we define a pair of constants. **`kMaxLineLength`** specifies the largest line this program can handle. 200 bytes should be plenty. **`kZeroByte`** has a value of zero and is used to mark the end of the line of input. More of this in a bit.

```
#define kMaxLineLength          200
#define kZeroByte               0
```

Here are the function prototypes for the two functions **`ReadLine()`** and **`CountWords()`**. **`ReadLine()`** reads in a line of text and **`CountWords()`** takes a line of text and returns the number of words in the line.

```
void ReadLine( char *line );
int         CountWords( char *line );
```

`main()` starts by defining an array of **`char`**s that will hold the line of input we type and an **`int`** that will hold the result of our call to **`CountWords()`**.

```
int main (int argc, const char * argv[])
{
 char       line[ kMaxLineLength ];
 int        numWords;
```

Once we type the prompt, we'll pass **`line`** to **`ReadLine()`**. Remember that **`line`** is a pointer to the first byte of the array of **`char`**s. When **`ReadLine()`** returns, **`line`** contains a line of text, terminated by a zero byte, making **`line`** a legitimate, zero-terminated C string. We'll pass that string on to **`CountWords()`**.

```
   printf( “Type a line of text, please:\n” );

   ReadLine( line );
   numWords = CountWords( line );
```

We then print a message telling us how many words we just counted.

```
   printf( “\n---- This line has %d word”,
   numWords );

   if ( numWords != 1 )
      printf( “s” );

   printf( “ ----\n%s\n”, line );

   return 0;
 }
```

This last bit of code shows attention to detail, something very important in a good program. Notice that the first **`printf()`** ended with the characters **“`word`”**. If the program found either no words or more than one word, we want to say:

```
This line has 0 words.
```

or

```
This line has 2 words.
```

If the program found exactly one word, the sentence should read:

```
This line has 1 word.
```

The last **`if`** statement makes sure the “**`s`**” gets added if needed.

In **`main()`**, we defined an array of **`char`**s to hold the line of characters we type in. When **`main()`** called **`ReadLine()`**, it passed the name of the array as a parameter to **`ReadLine()`**:

```
char             line[ kMaxLineLength ];

ReadLine( line );
```

As we said earlier, the name of an array also acts as a pointer to the first element of the array. In this case, **`line`** is equivalent to **`&(line[0])`**. **`ReadLine()`** now has a pointer to the first byte of **`main()`**'s **`line`** array.

```
void ReadLine( char *line )
{
```

This **`while`** loop calls **`getchar()`** to read a character at a time from the input buffer. **`getchar()`** returns the next character in the input buffer or, if there's an error, it returns the constant **`EOF`**. You'll learn more about **`EOF`** in Chapter 10.

The first time through the loop, **`line`** points to the first byte of **`main()`**'s **`line`** array. At this point, the expression **`*line`** is equivalent to the expression **`line[0]`**. The first time through the loop, we're getting the first character from the input buffer and copying it into **`line[0]`**.

The **`while`** loop continues as long as the character we just read in is not **`'\n'`** (as long as we have not yet retrieved the return character from the input buffer).

```
   while ( (*line = getchar()) != '\n' )
      line++;
```

Each time through the loop, we'll increment **`ReadLine()`**'s local copy of the pointer **`line`**, so it points to the next byte in **`main()`**'s **`line`** array. The next time through the loop, we'll read a character into the second byte of the array, then the third byte, etc., until we read in a **`'\n'`**, and we drop out of the loop.

This technique is known as **pointer arithmetic**. When you increment a pointer that points into an array, the value of the pointer is actually incremented just enough to point to the next element of the array. For example, if **`line`** were an array of 4 byte **`float`**s instead of **`char`**s, this line of code:

```
line++;
```

would increment **`line`** by 4 instead of by 1. In both cases, **`line`** would start off pointing to **`line[0]`** then, after the statement **`line++`**, **`line`** would point to **`line[1]`**.

Take a look at this code:

```
char    charPtr;

float   floatPtr;

double  doublePtr;

charPtr++;

floatPtr++;

doublePtr++;
```

In the last three statements, **`charPtr`** gets incremented by 1 byte, **`floatPtr`** gets incremented by 4 bytes, and **`doublePtr`** gets incremented by 8 bytes (assuming 1 byte **`char`**s, 4 byte **`float`**s, and 8 byte **`double`**s).

This is an extremely important concept to understand. If this seems fuzzy to you, go back and reread this section, then write some code to make sure you truly understand how pointers work, especially as they relate to arrays.

Once we drop out of the loop, we'll place a zero in the next position of the array. This turns the line into a zero-terminated string we can print using **`printf()`**.

```
    *line = kZeroByte;
  }
```

`CountWords()` also takes a pointer to the first byte of **`main()`**'s **`line`** array as a parameter. **`CountWords()`** will step through the array, looking for non-white space characters. When one is encountered, **`CountWords()`** sets **`inWord`** to **`true`** and increments **`numWords`**, then keeps stepping through the array looking for a white-space character which marks the end of the current word. Once the white-space is found, **`inWord`** is set to **`false`**.

```
  int  CountWords( char *line )
  {
    int       numWords, inWord;

    numWords = 0;
    inWord = false;
```

This process continues until the zero byte marking the end of the line is encountered.

```
    while ( *line != kZeroByte )
    {
        if ( ! isspace( *line ) )
        {
            if ( ! inWord )
            {
                numWords++;
                inWord = true;
            }
        }
        else
            inWord = false;

        line++;
    }
```

Once we drop out of the loop, we'll return the number of words in the line.

```
    return numWords;
  }
```

Now that you've seen arrays and pointers, there's something you should know. Every program in our book features a **`main()`** function that takes a pair of parameters:

```
int main (int argc, const char *
argv[])
```

Though we won't make use of these parameters in the book, here's the basics. The first parameter, **`argc`**, is an **`int`** that tells you how many parameters are folded into the second parameter, **`argv`**. **`argv`** is an array of pointers, each of which points to a parameter.

And where do these parameters come from, you may ask? Great question! They come from the Unix command line. When you launch a program using Terminal, you can add in a list of parameters. For example, suppose we wrote a program that counted the number of words in a text file. In the Unix universe, we'd typically start the program like this:

```
$ countWords
```

The dollar sign (**$**) is the Unix command line prompt and countWords is the name of the program we are running. countWords might prompt us for the name of a text file, then go count its words.

Another approach would be to launch the program like so:

```
$ countWords myFile.txt
```

Now, when countWords gets launched, argc will have a value of 1 and argv will contain a single array element, a pointer to a char array containing the string **`"myFile.txt"`**. Just thought you might be wondering!

What's Next?

Congratulations! You've made it through one of the longest chapters in the book. You've mastered several new data types, including **`float`**s and **`char`**s. You've learned how to use arrays, especially in conjunction with **`char`**s. You've also learned about C's text-substitution mechanism, the **`#define`**.

Chapter 9 will teach you how to combine C's data types to create your own customized data types called **`struct`**s. So go grab some lunch, lean back, prop up your legs, and turn the page.

Exercises

1) What’s wrong with each of the following code fragments:

a)

```
char      c;
int       i;

i=0;
for ( c=0; c<=255; c++ )
   i += c;
```

b)

```
float     myFloat;

myFloat = 5.125;
printf( “The value of myFloat is %d.\n”, f );
```

c)

```
char      c;

c = “a”;

printf( “c holds the character %c.”, c );
```

d)

```
char      c[ 5 ];

c = “Hello, world!”;
```

e)

```
char      c[ kMaxArraySize ]

#define kMaxArraySize   20

int i;

for ( i=0; i<kMaxArraySize; i++ )
   c[ i ] = 0;
```

f)

```
#define kMaxArraySize   200

char      c[ kMaxArraySize ];

c[ kMaxArraySize ] = 0;
```

g)

```
#define kMaxArraySize   200

char      c[ kMaxArraySize ], *cPtr;
int       i;

cPtr = c;
for ( i=0; i<kMaxArraySize; i++ )
   cPtr++ = 0;
```

h)

```
#define kMaxArraySize   200

char      c[ kMaxArraySize ];
int       i;

for ( i=0; i<kMaxArraySize; i++ )
{
   *c = 0;
   c++;
}
```

i)

```
#define kMaxArraySize   200;
```

2) Rewrite *dice.xcode*'s *main.c*, showing the possible rolls using three dice instead of two.

3) Rewrite *wordCount.xcode*'s *main.c*, printing each of the words, one per line.

Chapter 9 Design Your Own Data Structures

In Chapter 8, we introduced several new data types, such as **`float`**, **`char`**, and **`short`**. We discussed the range of each type and introduced the format specification characters necessary to print each type using **`printf()`**. Next, we introduced the concept of arrays, focusing on the relationship between **`char`** arrays and text strings. Along the way, we discovered the **`#define`**, C's text substitution mechanism.

This chapter will show you how to use existing C types as building blocks to design your own customized data structures.

Structures

There will be times when your programs will want to bundle certain data together. For example, suppose you were writing a program to organize your compact disc collection. Imagine the type of information you'd like to access for each CD. At the least, you'd want to keep track of the artist's name and the name of the CD. You might also want to rate each CD's listenability on a scale of 1 to 10.

In the next few sections, we'll look at two separate approaches to a basic CD-tracking program. Each approach will revolve around a different set of data structures. One will make use of arrays and the other a set of custom designed data structures.

Model A: Three Arrays

One way to model your CD collection is with a separate array for each CD's attributes:

```
#define kMaxCDs                   5000
#define kMaxArtistLength          256
#define kMaxTitleLength           256

char rating[ kMaxCDs ];
char artist[ kMaxCDs ][ kMaxArtistLength ];
char title[ kMaxCDs ][ kMaxTitleLength ];
```

This code fragment uses three **`#define`**s. **`kMaxCDs`** defines the maximum number of CDs this program will track. **`kMaxArtistLength`** defines the maximum length of a CD artist's name. **`kMaxTitleLength`** defines the maximum length of a CD's title.

`rating` is an array of 5,000 **`char`**s, one **`char`** per CD. Each of the **`char`**s in this array will hold a number from 1 to 10, the rating we've assigned to a particular CD. This line of code assigns a value of 8 to CD 37:

```
rating[ 37 ] = 8; /* A pretty good CD */
```

The arrays **`artist`** and **`title`** are each known as **multi-dimensional arrays**. A normal array, like **`rating`**, is declared using a single dimension. The statement:

```
float       myArray[ 5 ];
```

declares a normal or one-dimensional array containing 5 **`float`**s, namely:

```
myArray[ 0 ]
myArray[ 1 ]
myArray[ 2 ]
myArray[ 3 ]
myArray[ 4 ]
```

The statement:

```
float       myArray[ 3 ][ 5 ];
```

declares a two-dimensional array, containing 3*5 = 15 **`float`**s, namely:

```
myArray[0][0]
myArray[0][1]
myArray[0][2]
myArray[0][3]
myArray[0][4]
myArray[1][0]
myArray[1][1]
myArray[1][2]
myArray[1][3]
myArray[1][4]
myArray[2][0]
myArray[2][1]
```

```
myArray[2][2]
myArray[2][3]
myArray[2][4]
```

Think of a two dimensional array as an array of arrays. **`myArray[0]`** is an array of 5 **`float`**s. **`myArray[1]`** and **`myArray[2]`** are also arrays of 5 **`float`**s each.

Here's a three dimensional array:

```
float        myArray[ 3 ][ 5 ][ 10 ];
```

How many **`float`**s does this array contain? Tick, tick, tick... Got it? 3*5*10 = 150. This version of **`myArray`** contains 150 **`float`**s.

C allows you to create arrays of any dimension, though you'll rarely have a need for more than a single dimension.

So why would you ever want a multi-dimensional array? If you haven't already guessed, the answer to this question is going to lead us back to our CD tracking example.

Here are the declarations for our three CD-tracking arrays:

```
#define kMaxCDs                 5000
#define kMaxArtistLength        256
#define kMaxTitleLength         256

char rating[ kMaxCDs ];
char artist[ kMaxCDs ][ kMaxArtistLength ];
char title[ kMaxCDs ][ kMaxTitleLength ];
```

Once again, **`rating`** contains one **`char`** per CD. **`artist`**, on the other hand, contains an array of **`char`**s for each CD. Each CD gets an array of **`char`**s whose length is **`kMaxArtistLength`**. Each array is large enough to hold an artist's name up to 255 bytes long with a single byte left over to hold the terminating zero byte. To restate this, the two-dimensional array **`artist`** is large enough to hold up to 5,000 artist names, each of which can be up to 255 characters long, not including the terminating byte.

multiArray.xcode

Here's a sample program that brings this concept to life. **`multiArray`** defines the two dimensional array **`title`** (as described above), prompts you to type in a series of CD titles, stores the titles in the two-dimensional **`title`** array, then prints out the contents of **`title`**.

Open the *Learn C Projects* folder, go inside the folder *09.01 - multiArray*, and open the project *multiArray.xcode*. Run **`multiArray`** by selecting Build and Run from the Build menu. **`multiArray`** will first tell you how many bytes of memory are allocated for the entire **`title`** array:

```
The artist array takes up 1024 bytes of
 memory.
```

To see where this number came from, here's the declaration of **`title`** from **`MultiArray`**:

```
#define kMaxCDs              4
#define kMaxTitleLength      256

char title[ kMaxCDs ][ kMaxTitleLength ];
```

By performing the **`#define`** substitution yourself, you can see that **`title`** is defined as a 4 by 256 array. 4 times 256 is 1,024, matching the result reported by **`multiArray`**.

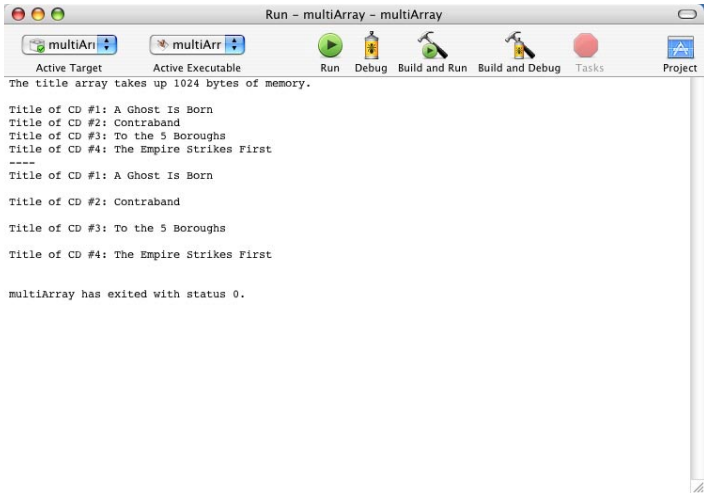
After **`multiArray`** reports the **`title`** array size, it enters a loop, prompting you for your list of favorite musical artists:

```
Title of CD #1:
```

Enter a CD title, then hit a return. You'll be prompted to enter a second CD title. Type in a total of 4 CD titles, hitting a return at the end of each one.

`multiArray` will then step through the array, using **`printf()`** to list the CDs you've entered. In case your entire music collection consists entirely of a slightly warped vinyl copy of Leonard Nimoy singing some old Dylan classics, feel free to use my list, shown in Figure 9.1.

Let's take a look at the source code.



```
The title array takes up 1024 bytes of memory.

Title of CD #1: A Ghost Is Born
Title of CD #2: Contraband
Title of CD #3: To the 5 Boroughs
Title of CD #4: The Empire Strikes First
----
Title of CD #1: A Ghost Is Born

Title of CD #2: Contraband

Title of CD #3: To the 5 Boroughs

Title of CD #4: The Empire Strikes First


multiArray has exited with status 0.
```

Figure 9.1 **`multiArray`** *in action.*

Stepping Through the Source Code

main.c starts off with a standard **`#include`**. **`<stdio.h>`** gives us access to both **`printf()`** and **`fgets()`**. **`fgets()`** reads a line of text from console's input buffer (also known as **`stdin`**).

```
#include <stdio.h>
```

Back in the olden days, before black hat hackers, when everyone was nice and good, programmers used a function called **`gets()`** to read data from the console. The problem is, **`gets()`** uses a finite buffer to store a potentially infinite amount of input. For example, suppose you declared a 256-byte array for **`gets()`** to use to capture the user's input. If all you were asking for was an artist name, 256 bytes should be plenty, right?

Suppose the user types in 300 bytes? Since you send **`gets()`** a pointer to a buffer, the extra characters will be written right off the end of the array, perhaps writing over some other variables, perhaps trashing the program itself.

Our replacement function, **`fgets()`**, allows you to specify a limit on how many characters it can read in. By limiting **`fgets()`** to the actual length of the buffer you pass it, you avoid the problem of **buffer overflow**. The concept of buffer overflow is extremely important. Keep it in mind as you design your own programs.

If you look up **`gets()`** in the Standard Library documentation, you'll see the notation that you should not use **`gets()`** because of the potential for buffer overflow. Check this out for yourself:

http://www.infosys.utas.edu.au/info/documentation/C/CStdLib.html

Here's an interesting article on buffer overflows, as used by black hat hackers to attack individual computers and the internet itself:

http://www.networkmagazine.com/article/NMG20000511S0015

These two **`#define`**s will be used throughout the code:

```
#define kMaxCDs              4
#define kMaxTitleLength      256
```

Here's the function prototypes for **`PrintCDTitle()`**. **`PrintCDTitle()`** will prints out the specified CD title.

```
void PrintCDTitle( int cdNum,
        char title[][ kMaxTitleLength ] );
```

`main()` starts off by defining **`title`**, our two-dimensional array. **`title`** is large enough to hold 4 artists. The name of each artist can be up to 255 bytes long, plus the zero terminating byte.

```
int main (int argc, const char * argv[])
{
 char     title[ kMaxCDs ][
 kMaxTitleLength ];
```

Notice anything different about the declaration of **`title` in the `PrintCDTitle()` prototype and the declaration of `title` in `main()`**? More on that in a bit.

`cdNum` is a counter used to step through each of the CD titles in a **`for`** loop.

```
    short      cdNum;
```

`result` is a pointer to a **`char`**, also known as a **`char`**-star. Though we don't make use of it, **`result`** captures the value returned by **`fgets()`**. If you call a function that returns a value, be sure you are prepared to capture the result in a variable of the appropriate type, even if you never intend to use that returned value.

```
    char       *result;
```

This **`printf()`** prints out the size of the **`title`** array. Notice that we've used the **`%ld`** format specifier to print the result returned by **`sizeof`**. **`%ld`** indicates that the type you are printing is the size of a **`long`**, which is true for **`size_t`**, the type returned by **`sizeof`**. If you use **`%ld`**, you won't need the **`(int)`** typecast we used in earlier programs.

```
    printf( "The artist array takes up %ld bytes
    of memory.\n\n",
                      sizeof( title ) );
```

`size_t` is not guaranteed to be an **`unsigned long`**, though it usually is. The only guarantee is that **`size_t`** is the same size as that returned by the **`sizeof`** operator. In our case, **`size_t`** is defined as an **`unsigned long`**, so the "**`%ld`**" format specifier will work just fine.

Here's the loop that reads in the title names. **`cdNum`** starts with a value of 0, is incremented by 1 each time through the loop, and stops as soon as **`cdNum`** is equal to **`kMaxCDs`**. Why "equal to **`kMaxCDs`**"? Since **`cdNum`** acts as an array index, it has to start with a value of 0. Since there are 4 elements in the array, they range in number from 0 to 3. If **`cdNum`** is *equal* to **`kMaxCDs`**, we need to drop out of the loop or we'll be trying to access **`title[4]`**, which does not exist. Make sense?

```
    for ( cdNum = 0; cdNum < kMaxCDs; cdNum++ )
    {
```

Each time through the loop, we first print out the prompt "**`Title of CD #`**", followed by the value **`cdNum + 1`**. Though C starts its arrays with 0, in real life we start numbering things with 1.

```
        printf( "Title of CD #%d: ", cdNum + 1 );
```

Once the prompt is printed, we'll call **`fgets()`** to read in a line of text from the console. We'll store the line in the **`char`** array stored in **`title[ cdNum ]`**. We'll tell **`fgets()`** to limit input to the length of that **`char`** array, which is **`kMaxTitleLength`**. The last parameter, **`stdin`**, tells **`fgets()`** to read its input from the console, as opposed to reading from a file.

```
        result = fgets( title[ cdNum ],
                        kMaxTitleLength, stdin );
    }
```

Take a look at the first parameter we passed to **`fgets()`**:

```
    title[ cdNum ]
```

What type is this parameter? Remember, **`title`** is a two-dimensional array, and a two-dimensional array is an array of arrays. **`title`** is an array of an array of **`char`**s. **`title[ cdNum ]`** is an array of **`char`**s, and thus exactly suited as a parameter to **`fgets()`**.

Imagine an array of **`char`**s named **`blap`**:

```
    char blap[ 100 ];
```

You'd have no problem passing **`blap`** as a parameter to **`fgets()`**, right? **`fgets()`** would read the characters from the input buffer and place them in **`blap`**. **`title[0]`** is just like **`blap`**. Both are pointers to an array of **`char`**s. **`blap[0]`** is the first **`char`** of the array **`blap`**. Likewise, **`title[0][0]`** is the first **`char`** of the array **`title[0]`**.

OK, back to the code.

Once we drop out of the loop, we print a dividing line, then loop on a call to **`PrintCDTitle()`** to print the contents of our array of CD titles. The first parameter to **`PrintCDTitle()`** specifies the number of the CD you want printed. The second parameter is the **`title`** array pointer.

```
    printf( "----\n" );

    for ( cdNum = 0; cdNum < kMaxCDs; cdNum++ )
        PrintCDTitle( cdNum, title );
```

Finally, we **`return 0`** and thus ends **`main()`**.

```
    return 0;
  }
```

Take a look at the definition of **`PrintCDTitle()`**'s second parameter. Notice that the first of the two dimensions is missing (the first pair of brackets is empty). While we could have included the first dimension (**kMaxCDs**), the fact that we were able to leave it out makes a really interesting point. When memory is allocated for an array, it is allocated as one big block. To access a specific element of the array, the compiler uses the dimensions of the array, as well as the specific element requested to calculate an offset into this block.

```
void PrintCDTitle( int cdNum, char title[][
 kMaxTitleLength ] )
{
 printf( "Title of CD #%d: %s\n",
          cdNum + 1, title[ cdNum ] );
}
```

In the case of **title**, the compiler allocated a block of memory 4 * 256 = 1,024 bytes long. Think of this block as 4 **char** arrays, each of which is 256 bytes long. To get to the first byte of the first array, we just use the pointer that was passed in (**title** points to the first byte of the first of the 4 arrays). To access the first byte of the second array (in C notation, **title[1][0]**) the compiler adds 256 to the pointer **title**. In other words, the start of the second array is 256 bytes further in memory than the start of the first array. The start of the 4th array is 3*256 = 768 bytes further in memory than the start of the first array.

While it is nice to know how to compute array offsets in memory, the point I'm going for here is that the compiler calculates the **title** array offsets using the second dimension and not the first dimension of **title** (256 is used, 4 is not used).

The compiler could use the first array bound (4) to verify that you don't reference an array element that is **out of bounds**. For example, the compiler could complain if it sees this line of code:

```
title[5][0] = '\0';
```

In this case, the compiler *could* tell you that you are trying to reference a memory location outside the block of memory allocated for **title**.

Guess what. C compilers don't do bounds checking of any kind. If you want to access memory beyond the bounds of your array, no one will stop you. This is part of the "charm" of C. C gives you the freedom to write programs that crash in spectacular ways. Your job is to learn how to avoid such pitfalls.

Take another look at the **printf()** inside **PrintCDTitle()**:

```
printf( "Title of CD #%d: %s\n",
          cdNum + 1, title[ cdNum ] );
```

Note the two format specifiers. The first, **%d**, is used to print the CD number. The second, **%s**, is used to print the CD title itself. The "**\n**" at the end of the string is used to force a carriage return between each of the CD titles.

The more sharp-eyed among you may have noticed that there is an extra carriage return between each of the titles. To see this, flip back to Figure 9.1. Go ahead, I'll wait.

As it turns out, **`fgets()`** captures the carriage return at the end of your input as part of the input string. This means each CD title has a "**`\n`**" embedded in it, just before the terminating `0`. Not a big deal, but simple enough to fix, if you care to.

First, include this line of code with your other **`#include`** at the top of *main.c*:

```
#include <string.h>
```

After your call to **`fgets()`**, still inside the **`for`** loop, insert this line of code:

```
title[ cdNum ][ strlen( title[ cdNum ]
) - 1 ] = '\0';
```

Yikes!!! Take a few moments to digest this line of code. What we are doing is using the Standard Library function **`strlen()`** to determine the length of the current CD title. For example, if the title was *Jamboree*, **`strlen()`** would return 9, because of the extra "**`\n`**" character in the title.

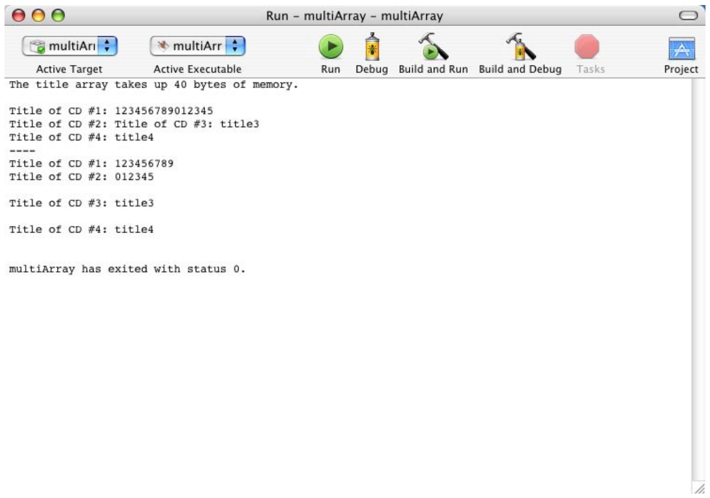
We want to replace that "**`\n`**" with a `0`. Note that we used the character **`'\0'`**, which has a value of `0`. We could have used a `0` instead. So we subtract 1 from 9 to get 8. Since strings start counting with 0, **`title[cdNum][8]`** is actually the 9th character in the string. By setting that to **`'\0'`**, we've replace the "**`\n`**" with a `0`.

Now when you print, your extra carriage returns will be gone.

Wanna see something interesting? Take a look at the output shown in Figure 9.2. I shortened **`kMaxTitleLength`** to 10, recompiled, ran **`multiArray`**, then typed the digits **`123456789012345`** as the title of the first CD.

When I hit a return, **`fgets()`** read its limit of 9 characters from the input buffer, saving one byte for the terminating `0`. The remaining 6 characters and the trailing carriage return were read by **`fgets()`** the next time through the **`for`** loop and the program finished normally.

No big deal. When I return **`kMaxTitleLength`** to 256, all is well again.



```
The title array takes up 40 bytes of memory.

Title of CD #1: 123456789012345
Title of CD #2: Title of CD #3: title3
Title of CD #4: title4
----
Title of CD #1: 123456789
Title of CD #2: 012345

Title of CD #3: title3

Title of CD #4: title4


multiArray has exited with status 0.
```

Figure 9.2 *This output is the result of a bug in the program. Take a look at the end of both lines labeled* **`Title of CD #1`**.

Back to Model A

Back in the beginning of the chapter, we described a program that would track your CD collection. The goal was to look at two different approaches to solving the same problem. The first approach, Model A, uses three arrays to hold a rating, artist name and title for each CD in the collection:

```
#define kMaxCDs                5000
#define kMaxArtistLength       256
#define kMaxTitleLength        256

char rating[ kMaxCDs ];
char artist[ kMaxCDs ][ kMaxArtistLength ];
char title[ kMaxCDs ][ kMaxTitleLength ];
```

Before we move on to Model B, let's take a closer look at the memory used by the Model A arrays.

- The array **`rating`** uses 1 byte per CD (enough for a 1-byte rating from 1 to 10).
- The array **`artist`** uses 256 bytes per CD (enough for a text string holding the artist's name, up to 255 bytes in length, plus the terminating byte).
- The array **`title`** also uses 256 bytes per CD (enough for a text string holding the CD's title, up to 255 bytes in length, plus the terminating byte).

Add those three together and you find that Model A allocates 513 bytes per CD. Since Model A allocates space for 5,000 CDs when it declares its three key arrays, it uses 5,000 * 513 = 2,565,000 bytes.

Since the program really only needs 513 bytes per CD, wouldn't it be nice if you could allocate the memory for a CD when you need it? With this type of approach, if your collection only consisted of 50 CDs, you'd only have to use 50 * 513 = 25,650 bytes of memory instead of 2,565,000.

As you'll see by the end of the chapter, C provides a mechanism for allocating memory as you need it. Model B takes a first step toward memory efficiency by creating a single data structure that contains all the information relevant to a single CD. Later in the chapter you'll learn how to allocate just enough memory for a single structure.

Model B: The Data Structure Approach

As stated earlier, our CD program must keep track of a rating (from 1 to 10), the CD artist's name, and the CD's title:

```
#define kMaxCDs                 5000
#define kMaxArtistLength        256
#define kMaxTitleLength         256

char rating[ kMaxCDs ];
char artist[ kMaxCDs ][ kMaxArtistLength ];
char title[ kMaxCDs ][ kMaxTitleLength ];
```

C provides the perfect mechanism for wrapping all three of these variables in one tidy bundle. A **`struct`** allows you to associate any number of variables together under a single name. Here's an example of a **`struct`** declaration:

```
#define kMaxArtistLength        256
#define kMaxTitleLength         256

struct CDInfo
{
 char      rating;
 char      artist[ kMaxArtistLength ];
 char      title[ kMaxTitleLength ];
}
```

This **struct type declaration** creates a new type called **`CDInfo`**. Just as you'd use a type like **`short`** or

float to declare a variable, you can use this new type to declare an individual **struct**. Here's an example:

```
struct CDInfo     myInfo;
```

This line of code uses the previous type declaration as a template to create an individual **struct**. The compiler uses the type declaration to tell it how much memory to allocate for the **struct**, then allocates a block of memory large enough to hold all of the individual variables that make up the **struct**.

The variables that form the **struct** are known as **fields**. A **struct** of type **CDInfo** has 3 fields: a **char** named **rating**, an array of **char**s named **artist**, and an array of **char**s named **title**. To access the fields of a **struct**, use the "**.**" operator:

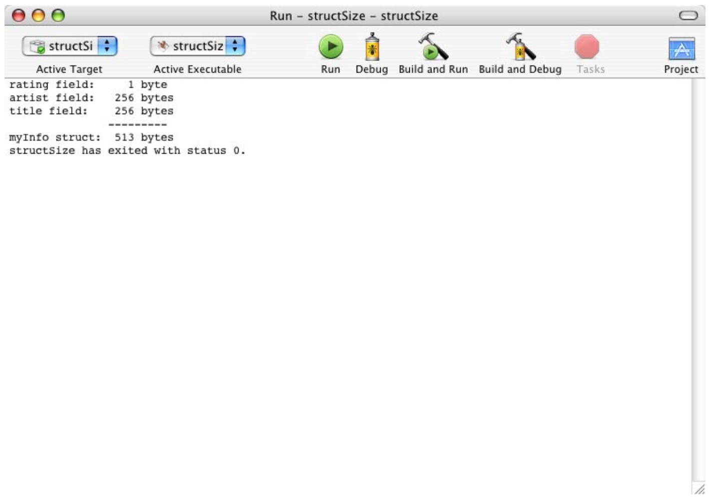
```
struct CDInfo     myInfo;

myInfo.rating = 7;
```

Notice the **.** between the **struct** name (**myInfo**) and the field name (**rating**). The **.** following a **struct** name tells the compiler that a field name is to follow.

structSize.xcode

Here's a program that demonstrates the declaration of a **struct** type, as well as the definition of an individual **struct**. Open the *Learn C Projects* folder, go inside the folder *09.02 - structSize*, and open the project *structSize.xcode*. Run **structSize**.

Compare your output with the console window shown in Figure 9.3. They should be the same. The first three lines of output show the **rating**, **artist**, and **title** fields. To the right of each field name, you'll find printed the number of bytes of memory allocated to that field. The last line of output shows the memory allocated to the entire **struct**.



```
rating field:     1 byte
artist field:   256 bytes
title field:    256 bytes
              ---------
myInfo struct:  513 bytes
structSize has exited with status 0.
```

Figure 9.3 **`structSize`** *shows the size of a* **`CDInfo struct`**.

Stepping Through the Source Code

If you haven't done so already, quit **`structSize`** and take a minute to look over the source code in *main.c*. Once you feel comfortable with it, read on.

main.c starts off with our standard **`#include`** along with a brand new one:

```
#include <stdio.h>
#include "structSize.h"
```

The angle-brackets (**`<>`**) that surrounds all the include files we've seen so far tell the compiler to look in the include file directories that it knows about. When you surround the include file name by double-quotes (**`""`**) instead of angle-brackets, like those around **`"structSize.h"`** in this example, you are telling the compiler to look for this include file in the same folder as the including source code file.

Regardless of where it locates the include file, the compiler treats the contents of the include file as if it were actually inside the including file. In this case, the compiler treats **`<stdio.h>`** and **`"structSize.h"`** as if they were directly inside *main.c*.

As you've already seen, C include files typically end in the two characters "**`.h`**". Though you *can* give your include files any name you like, the "**`.h`**" convention is one you should definitely stick with. Include files are also known as **header files**, which is where the "**`h`**" comes from.

Let's take a look at *structSize.h*. One way to do this is to select Open… from the File menu, navigate into the same directory as the *structSize.xcode* project, and select the file.

A simpler way to do this is to use Xcode's include file popup menu, as shown in Figure 9.4. In the project window, look towards the right side of the window, just above the vertical scrollbar, for a popup menu whose label is in the shape of a **#**. Click on the popup and select structSize.h from the menu. A new window will open containing *structSize.h*.

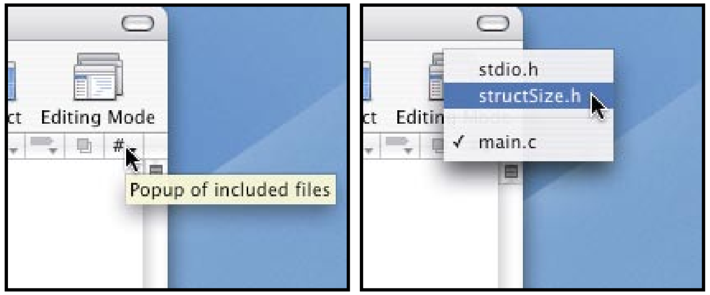


Figure 9.4 Selecting an include file from Xcode's include file popup.

Include files typically contain things like **`#define`**s, global variables, and function prototypes. By embedding these things in an include file, you declutter your source code file and, more importantly, you make this common source code available to other source code files via a single **`#include`**.

structSize.h starts off with two **`#define`**s you've seen before.

```
#define kMaxArtistLength  256
#define kMaxTitleLength   256
```

Next comes the declaration of the **`struct`** type, **`CDInfo`**:

```
/***********************/
/* Struct Declarations */
/***********************/
struct CDInfo
{
 char       rating;
 char       artist[ kMaxArtistLength ];
 char       title[ kMaxTitleLength ];
};
```

By including the header file at the top of the file (where we might place our globals), we've made the **`CDInfo struct`** type available to all of the functions inside *main.c*. If we placed the **`CDInfo`** type declaration inside of **`main()`** instead, our program would still have worked (as long as we placed it before the definition of **`myInfo`**), but we would then not have access to the **`CDInfo`** type outside of **`main()`**.

That's all that was in the header file *structSize.h*. Back in *main.c*, **`main()`** starts by defining a **`CDInfo struct`** named **`myInfo`**. **`myInfo`** has 3 fields, **`myInfo.rating`**, **`myInfo.artist`**, and **`myInfo.title`**.

```
int main (int argc, const char * argv[])
{
 struct CDInfo    myInfo;
```

The next three statements print the size of the three **`myInfo`** fields. Notice that we are again using the **`%ld`** format specifier to print the value returned by `sizeof`.

```
   printf( "rating field:     %ld byte\n",
             sizeof( myInfo.rating ) );

   printf( "artist field:   %ld bytes\n",
             sizeof( myInfo.artist ) );

   printf( "title field:    %ld bytes\n",
             sizeof( myInfo.title ) );
```

This next **`printf()`** prints a separator line, purely for aesthetics. Notice the way everything lines up in Figure 9.3?

```
   printf( "                ---------\n" );
```

Finally, we print the total number of bytes allocated to the struct. Do the numbers add up? They should!

```
   printf( "myInfo struct: %ld bytes",
             sizeof( myInfo ) );

   return 0;
  }
```

As it turns out, there are some computers where the numbers will *not* add up. Here's why. Some computers have rules they follow to keep various data types lined up a certain way. For example, on old 680x0 machines, the compiler forces all data larger than a **`char`** to start on an even-byte boundary (at an even memory address). A **`long`** will always start at an even address. A **`short`** will always start at an even address. A **`struct`**, no matter its size, will always start at an even address. Conversely, a **`char`** or array of **`char`**s can start at either an odd or even address. In addition, on a 680x0 machine, a **`struct`** must always have an even number of bytes.

In our example, the three **`struct`** fields are all either **`char`**s or arrays of **`char`**s, so they are all allowed to start at either an odd or even address. The three fields total to 103 bytes. Since a **`struct`** on a 680x0 must always have an even number of bytes, the compiler adds an extra byte (known as **padding** or a **pad byte**) at the end of the **`struct`**.

You might never see an example of this, but it is worth remembering that data alignment rules are not specific to the C language and can vary from CPU type to CPU type. When in doubt, write some code and try it out.

Passing a Struct as a Parameter

Think back to the CD tracking program we've been discussing throughout the chapter. We started off with three separate arrays, each of which tracked a separate element. One array stored the rating field, another stored the CD artist, and the third stored the title of each CD.

We then introduced the concept of a structure that would group all the elements of one CD together, in a single **`struct`**. One advantage of a **`struct`** is that you can pass all the information about a CD using a single pointer. Imagine a routine called **`PrintCD()`**, designed to print the three elements that describe a single CD. Using the original array-based model, we'd have to pass three parameters to **`PrintCD()`**:

```
void PrintCD( char rating, char *artist, char
 *title )
{
 printf("rating: %d\n", rating );
 printf("artist: %s\n", artist );
 printf("title: %s\n", title );
}
```

Using the **`struct`**-based model, however, we could pass the info using a single pointer. As a reminder, here's the **`CDInfo struct`** declaration again:

```
#define kMaxArtistLength        256
#define kMaxTitleLength         256

/***********************/
/* Struct Declarations */
/***********************/
struct CDInfo
{
 char      rating;
 char      artist[ kMaxArtistLength ];
 char      title[ kMaxTitleLength ];
};
```

This version of **`main()`** defines a **`CDInfo struct`** and passes its address to a new version of **`PrintCD()`** (we'll get to it next).

```
int main (int argc, const char * argv[])
{
 struct CDInfo    myInfo;

 PrintCD( &myInfo );

 return 0;
}
```

Just as has been the case in earlier programs, passing the address of a variable to a function gives that function the ability to modify the original variable. Passing the address of **`myInfo`** to **`PrintCD()`** gives **`PrintCD()`** the ability to modify the three **`myInfo`** fields. Though our new version of **`PrintCD()`** doesn't modify **`myInfo`**, it's important to know that that opportunity exists. Here's the new, **`struct`**-based version of **`PrintCD()`**:

```
void PrintCD( struct CDInfo *myCDPtr )
{
 printf( "rating: %d\n", (*myCDPtr).rating );
 printf( "artist: %s\n", myCDPtr->artist );
 printf( "title: %s\n", myCDPtr->title );
}
```

Notice that **`PrintCD()`** receives its parameter as a pointer to (i.e., the address of) a **`CDInfo struct`**. The first **`printf()`** uses the **`*`** operator to turn the **`struct`** pointer back to the **`struct`** it points to, then uses the **`.`** operator to access the **`rating`** field:

```
(*myCDPtr).rating
```

C features a special operator, **`->`**, that lets you accomplish the exact same thing. The **`->`** operator is binary. That is, it requires both a left and right operand. The left operand is a pointer to a **`struct`**, and the right operand is the **`struct`** field. The notation:

```
myCDPtr->artist
```

is exactly the same as:

```
(*myCDPtr).rating
```

Use whichever form you prefer. In general, most C programmers use the **`->`** operator to get from a **`struct`**'s pointer to one of the **`struct`**'s fields.

Passing a Copy of the Struct

Here's a version of **`main()`** that passes the **`struct`** itself, instead of its address:

```
int main (int argc, const char * argv[])
{
 struct CDInfo    myInfo;

 PrintCD( myInfo );
}
```

As always, when the compiler encounters a function parameter, it passes a copy of the parameter to the receiving routine. The previous version of **`PrintCD()`** received a copy of the address of a **`CDInfo struct`**.

In this new version of **`PrintCD()`**, the compiler passes a copy of the entire **`CDInfo struct`**, not just a copy of its address. This copy of the **`CDInfo struct`** includes copies of the **`rating`** field, and the **`artist`** and **`title`** arrays.

```
void PrintCD( struct CDInfo myCD )
{
 printf( "rating: %d\n", myCD.rating );
 printf( "artist: %s\n", myCD.artist );
 printf( "title: %s\n", myCD.title );
}
```

When a function exits, all of its local variables (except for **`static`** variables, which we'll cover in chapter 11) are no longer available. This means that any changes you make to a local parameter are lost when the function returns. If this version of **`PrintCD()`** made changes to its local copy of the **`CDInfo struct`**, those changes would be lost when **`PrintCD()`** returned.

Sometimes you'll want to pass a copy of a **`struct`**. One advantage this technique offers is that there's no way that the receiving function can modify the original **`struct`**. Another advantage is that it offers a simple mechanism for making a copy of a **`struct`**. A disadvantage of this technique is that copying a **`struct`** takes time and uses memory. Though time won't usually be a problem, memory usage might be, especially if your **`struct`** gets pretty large. Just be aware that whatever you pass as a parameter is going to get copied by the compiler. Pass a **`struct`** as a parameter, the compiler will copy the **`struct`**. Pass a pointer to a **`struct`**, the compiler will copy the pointer.

paramAddress.xcode

There's a sample program in the *Learn C Projects* folder, inside a subfolder named *09.03 - paramAddress*, that should help show the difference between passing the address of a **`struct`** and passing a copy of the **`struct`**. Open and run *paramAddress.xcode*.

`main()` defines a **`CDInfo struct`** named **`myCD`**, then prints the address of **`myCD`**'s **`rating`** field:

```
printf( "Address of myCD.rating in main():
%p\n",
         &(myCD.rating) );
```

Notice that we print an address using the **`%p`** format specifier. The p stands for pointer. This is the proper way to print an address in C. Here's the output of this **`printf()`**:

```
Address of myCD.rating in main():
 0xbffffba0
```

Next, **`main()`** passes the address of **`myCD`** as well as **`myCD`** itself as parameters to a routine named **`PrintParamInfo()`**:

```
PrintParamInfo( &myCD, myCD );
```

Here's the prototype for **`PrintParamInfo()`**:

```
void PrintParamInfo( struct CDInfo *myCDPtr,
                     struct CDInfo myCDCopy );
```

The first parameter is a pointer to **`main()`**'s **`myCD`** **`struct`**. The second parameter is a copy of the same

`struct`. **`PrintParamInfo()`** prints the address of the **`rating`** field of each version of **`myCD`**:

```
printf( "Address of myCDPtr->rating in
PrintParamInfo(): %p\n",
         &(myCDPtr->rating) );
printf( "Address of myCDCopy.rating in
PrintParamInfo(): %p\n",
         &(myCDCopy.rating) );
```

Here are the results, including the line of output generated by **`main()`**:

```
Address of myCD.rating in main():
 0xbffffba0
Address of myCDPtr->rating in
 PrintParamInfo(): 0xbffffba0
Address of myCDCopy.rating in
 PrintParamInfo(): 0xbffffb2c
```

Notice that the **`rating`** field accessed via a pointer has the same address as the original **`rating`** field in **`main()`**'s **`myCD struct`**. If **`PrintParamInfo()`** uses the first parameter to modify the **`rating`** field, it will, in effect, be changing **`main()`**'s **`rating`** field.

If **`PrintParamInfo()`** uses the second parameter to modify the **`rating`** field, **`main()`**'s **`rating`** field will remain untouched.

By the way, most programmers use **hexadecimal** notation (**hex** for short) when they print addresses. Hex notation represents numbers as base 16 instead of the normal base 10 you are used to. Instead of the ten digits 0 through 9, hex features the 16 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f. Each digit of a number represents a successive power of 16 instead of successive powers of 10.

For example, the number 532 in base 10 is equal to $5*10^2 + 3*10^1 + 2*10^0 = 5*100+3*10+2*1$. The number 532 in hex is equal to $5*16^2 + 3*16^1 + 2*16^0 = 5*256+3*16+2*1 = 1330$ in base 10. The number ff in hex is equal to $15*16 + 15*1 = 255$ in base 10. Remember, the hex digit f has a decimal (base 10) value of 15.

To represent a hex constant in C, preceded it by the characters "**`0x`**". The constant **`0xff`** has a decimal value of 255. The constant **`0xFF`** also has a decimal value of 255. C doesn't distinguish between upper and lower case when representing hex digits.

Struct Arrays

Just as you can declare an array of **`char`**s or **`int`**s, you can also declare an array of **`struct`**s:

```
#define kMaxCDs    5000

struct CDInfo      myCDs[ kMaxCDs ];
```

This declaration creates an array of 5,000 **`struct`**s of type **`CDInfo`**. The array is named **`myCDs`**. Each of the 5,000 **`struct`**s will have the three fields **`rating`**, **`artist`**, and **`title`**. You access the fields of the **`struct`**s as you might expect. Here's an example (Note the use of the all-important **`.`** operator):

```
myCDs[ 10 ].rating = 9;
```

We now have an equivalent to our first CD tracking data structure. Where the first model used three arrays, we now have a solution that uses a single array. As you'll see when you start writing your own programs, packaging your data in a **`struct`** makes life a bit simpler. Instead of passing three parameters each time you need to pass a CD to a function, you can simply pass a **`struct`**.

From a memory standpoint, both CD tracking solutions cost the same. With three separate arrays, the cost is:

```
                    5,000 bytes /*rating array*/
5,000 * 256 = 1,280,000 bytes /*artist array*/
5,000 * 256 = 1,280,000 bytes /*title array*/
              ---------------
Total         2,565,000 bytes
```

With an array of **`struct`**s, the cost is:

```
5,000 * 513 = 2,565,000 bytes  /* Cost of
 array of 5,000 CDInfo structs */
```

So what can we do to cut this memory cost down? Thought you'd never ask!

Allocating Your Own Memory

One of the limitations of an array-based CD-tracking model is that arrays are not resizable. When you define an array, you have to specify exactly how many elements make up your array.

For example, this code defines an array of 5,000 **`CDInfo struct`**s.

```
#define kMaxCDs    5000

struct CDInfo     myCDs[ kMaxCDs ];
```

As we calculated earlier, this array will take up 2,565,000 bytes of memory, whether we use the array to track 1 CD or 5,000. If you know in advance exactly how many elements your array requires, arrays are just fine. In the case of our CD-tracking program, this just isn't practical. For example, if my CD collection consists entirely of a test CD that came with my CD burner and a rare soundtrack recording of Gilligan's Island outtakes, a 5,000 **`struct`** array is overkill. Even worse, what happens if I've got more than 5,000 CDs? No matter what number I pick for **`kMaxCDs`**, there's always the chance that it won't prove large enough.

The problem here is that arrays are just not flexible enough to do what we want. Instead of trying to predict the amount of memory we'll need in advance, what we need is a method that will give us a chunk of memory the exact size of a **`CDInfo struct`**, as we need it. In more technical terms, we need to allocate and manage our own memory.

When your program starts running, your operating system (Mac OS X, Unix, and Windows XP are all examples of operating systems) carves out a chunk of memory for the exclusive use of your application.

Some of this memory is used to hold the object code that makes up your application. Still more of it is used to hold things like your application's global variables. As your application runs, some of this memory will be allocated to **`main()`**'s local variables. When **`main()`** calls a function, memory is allocated for that function's local variables. When that function returns, the memory allocated for its local variables is freed up, made available to be allocated all over again.

In the next few sections, you'll learn about some functions you can call to allocate a block of memory and to free the memory (to return it to the pool of available memory). Ultimately, we'll combine these functions with a data structure called a **linked list** to provide a more memory efficient, more flexible alternative to the array.

malloc()

The Standard Library function **`malloc()`** allows you to to allocate a block of memory of a specified size. To access **`malloc()`**, you'll need to include the file **`<stdlib.h>`**:

```
#include <stdlib.h>
```

`malloc()` takes a single parameter, the size of the requested block, in bytes. **`malloc()`** returns a pointer to the newly allocated block of memory. Here's the function prototype:

```
void *malloc( size_t size );
```

Note that the parameter is declared to be of type **`size_t`**, the same type returned by **`sizeof`**. Think of **`size_t`** as equivalent to an **`unsigned long`** (**`unsigned`** in that it only takes on positive values, and the size of a **`long`**). Note also that **`malloc()`** returns the type (**`void *`**), a pointer to a **`void`**. A **`void`** pointer is essentially a generic pointer. Since there's no such thing as a variable of type **`void`**, the type (**`void *`**) is used to declare a pointer to a block of memory whose type has not been determined.

In general, you'll convert the (**`void *`**) returned by **`malloc()`** to the pointer type you really want. Read on to see an example of this.

If **`malloc()`** can't allocate a block of memory the size you requested, it returns a pointer with the value **`NULL`**. **`NULL`** is a constant, usually defined to have a value of 0, used to specify an invalid pointer. In other words, a pointer with a value of **`NULL`** does not point to a legal memory address. You'll learn more about **`NULL`** and (**`void *`**) as we use them in our examples.

Here's a code fragment that allocates a single **`CDInfo struct`**:

```
struct CDInfo     *myCDPtr;

myCDPtr = malloc( sizeof( struct CDInfo ) );
```

The first line of code declares a new variable, **`myCDPtr`**, which is a pointer to a **`CDInfo struct`**. At this point, **`myCDPtr`** doesn't point to a **`CDInfo struct`**. You've just told the compiler that **`myCDPtr`** is designed to point to a **`CDInfo struct`**.

The second line of code calls **`malloc()`** to create a block of memory the size of a **`CDInfo struct`**. **`sizeof`** returns its result as a **`size_t`**, the type we need to pass as a parameter to **`malloc()`**. How convenient!

On the right side of the = operator we've got a (**`void *`**) and on the left side we've got a (**`struct CDInfo *`**). The compiler automatically resolves this type difference for us. We could have used a typecast here to make this more explicit:

```
myCDPtr = (struct CDInfo *)malloc(
sizeof(struct CDInfo) );
```

This explicit typecast really isn't necessary and, besides, we won't get into typecasting until Chapter 11!

If **`malloc()`** was able to allocate a block of memory the size of a **`CDInfo struct`**, **`myCDPtr`** contains the address of the first byte of this new block. If **`malloc()`** was unable to allocate our new block (perhaps there wasn't enough unallocated memory left) **`myCDPtr`** will be set to **`NULL`**.

```
if ( myCDPtr == NULL )
 printf( "Couldn't allocate the new block!\n"
 );
else
 printf( "Allocated the new block!\n" );
```

If **`malloc()`** succeeded, **`myCDPtr`** points to a **`struct`** of type **`CDInfo`**. For the duration of the program, we can use **`myCDPtr`** to access the fields of this newly allocated **`struct`**:

```
myCDPtr->rating = 7;
```

It is important to understand the difference between a block of memory allocate using **`malloc()`** and a block of memory that corresponds to a local variable. When a function declares a local variable, the memory associated with that variable is temporary. As soon as the function exits, the block of memory associated with that memory is returned to the pool of available memory.

A block of memory that you allocate using **`malloc()`** sticks around until you specifically return it to the pool of available memory or until your program exits.

free()

The Standard Library provides a function, called **`free()`**, which returns a previously allocated block of memory back to the pool of available memory. Here's the function prototype:

```
void free( void *ptr );
```

`free()` takes a single argument, a pointer to the first byte of a previously allocated block of memory. This line:

```
free( myCDPtr );
```

returns the block allocated earlier to the free memory pool. Use **`malloc()`** to allocate a block of memory.

Use `free()` to free up a block of memory allocated via `malloc()`. When a program exits, the operating system automatically frees up all memory allocated by that program.

Caution: Never put a fork in an electrical outlet. Never pass an address to `free()` that didn't come from `malloc()`. Both will make you extremely unhappy!

Keep Track of That Address!

The address returned by `malloc()` is critical. If you lose it, you've lost access to the block of memory you just allocated. Even worse, you can never `free()` the block, and it will just sit there, wasting valuable memory, for the duration of your program.

One great way to lose a block's address is to call `malloc()` inside a function, saving the address returned by `malloc()` in a local variable. When the function exits, your local variable goes away, taking the address of your new block with it!

One way to keep track of a newly allocated block of memory is to place the address in a global variable. Another way is to place the pointer inside a special data structure known as a **linked list**.

Working With Linked Lists

The linked list is one of the most widely used data structures in C. A linked list is a series of `struct`s, each of which contains, as a field, a pointer. Each `struct` in the series uses its pointer to point to the next `struct` in the series. Figure 9.5 shows a linked list containing three elements.

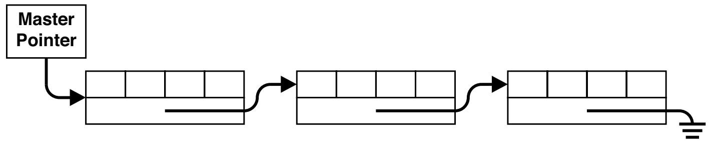


Figure 9.5 *A linked list containing 3 elements.*

A linked list starts with a **master pointer**. The master pointer is a pointer variable, typically a global, that points to the first `struct` in the list. This first `struct` contains a field, also a pointer, which points to the second `struct` in the linked list. The second `struct` contains a pointer field that points to the third element. The linked list in Figure 9.5 ends with the third element. The pointer field in the last element of a linked list is typically set to `NULL`.

The notation used at the end of the linked list in Figure 9.5 is borrowed from our friends in electrical engineering. The funky three line symbol at the end of the last pointer represents a `NULL` pointer.

Why Linked Lists?

Linked lists allow you to be extremely memory efficient. Using a linked list, you can implement our CD-tracking data structure, allocating exactly the number of **structs** that you need. Each time a CD is added to your collection, you'll allocate a new **struct** and add it to the linked list.

A linked list starts out as a single master pointer. When you want to add an element to the list, call **malloc()** to allocate a block of memory for the new element. Next, make the master pointer point to the new block. Finally, set the new block's next element pointer to **NULL**.

Creating a Linked List

The first step in creating a linked list is the design of the main link, the linked list **struct**. Here's a sample:

```
#define kMaxArtistLength        256
#define kMaxTitleLength         256

struct CDInfo
{
 char               rating;
 char               artist[ kMaxArtistLength ];
 char               title[ kMaxTitleLength ];
 struct CDInfo      *next;
}
```

The change here is the addition of a fourth field, a pointer to a **CDInfo struct**. The **next** field is the key to connecting two different **CDInfo structs** together. If **myFirstPtr** is a pointer to one **CDInfo struct** and **mySecondPtr** is a pointer to a second **struct**, this line:

```
myFirstPtr->next = mySecondPtr;
```

connects the two **structs** together. Once they are connected, you can use a pointer to the first **struct** to access the second **struct**'s fields! For example:

```
myFirstPtr->next->rating = 7;
```

This line sets the **rating** field of the second **struct** to 7. Using the **next** field to get from one **struct** to the next is also known as **traversing** a linked list.

Our next (and final) program for this chapter will incorporate the new version of the **CDInfo struct** to demonstrate a more memory-efficient CD-tracking program. This program is pretty long, so you may want to take a few moments to let the dog out and answer your mail.

There are many variants of the linked list. If you connect the last element of a linked list to the first element, you create a never-ending circular list. You can add a `prev` field to the `struct` and use it to point to the previous element in the list (as opposed to the next one). This technique allows you to traverse the linked list in two directions and creates a doubly-linked list.

As you gain more programming experience, you'll want to check out some books on data structures. Three books well worth exploring are *Algorithms in C, Parts 1-5* by Robert Sedgewick, *Data Structures and C Programs* by Christopher J. Van Wyk and, my personal favorite, Volume 1 of Donald Knuth's Computer Science Series (subtitled *Fundamental Algorithms*).

cdTracker.xcode

`cdTracker` implements Model B of our CD-tracking system. It uses a text-based menu, allowing you to quit, add a new CD to the collection, or list all of the currently tracked CDs.

Open the *Learn C Projects* folder, go inside the folder *09.04 - cdTracker*, and open the project *cdTracker.xcode*. Run **`cdTracker`**. The console window will appear, showing the prompt:

```
Enter command (q=quit, n=new, l=list):
```

At this point you have three choices. You can type a **`q`**, followed by a carriage return, to quit the program. You can type an **`n`**, followed by a carriage return, to add a new CD to your collection. Finally, you can type an **`l`**, followed by a carriage return, to list all the CDs in your collection.

Start by typing an **`l`**, followed by a carriage return. You should see the message:

```
No CDs have been entered yet...
```

Next, the original command prompt should reappear:

```
Enter command (q=quit, n=new, l=list):
```

This time type an **`n`**, followed by a carriage return. You will be prompted for the artist's name and the title of a CD you'd like added to your collection:

```
Enter Artist's Name:  Frank Zappa
Enter CD Title:  Anyway the Wind Blows
```

Next, you'll be prompted for a rating for the new CD. The program expects a number between 1 and 10. Try typing something unexpected, such as the letter **`x`**, followed by a carriage return:

```
Enter CD Rating (1-10):  x
Enter CD Rating (1-10):  10
```

The program checks your input, discovers it isn't in the proper range, and repeats the prompt. This time, type a number between 1 and 10, followed by a carriage return. The program returns you to the main command prompt:

```
Enter command (q=quit, n=new, l=list):
```

Type the letter **`l`**, followed by a carriage return. The single CD you just entered will be listed and the command prompt will again be displayed:

```
Artist:  Frank Zappa
Title:   Anyway the Wind Blows
Rating:  10

----------
Enter command (q=quit, n=new, l=list):
```

Type an **`n`**, followed by a carriage return and enter another CD. Repeat the process one more time, adding a third CD to the collection. Now enter the letter **`l`**, followed by a carriage return to list all three CDs. Here's my list:

```
Enter command (q=quit, n=new, l=list):  l
----------
Artist:  Frank Zappa
Title:   Anyway the Wind Blows
Rating:  10

----------
Artist:  Duke Ellington
Title:   Never No Lament
Rating:  8

----------
Artist:  Jane Siberry
Title:   Bound by the Beauty
Rating:  9

----------
Enter command (q=quit, n=new, l=list):
```

Finally, enter a **`q`**, followed by a carriage return to quit the program. Let's hit the source code.

Stepping Through the Source Code

main.c starts by including four different files. **`<stdio.h>`** gives us access to routines like **`printf()`**, **`getchar()`**, and **`fgets()`**. **`<stdlib.h>`** gives us access to **`malloc()`** and **`free()`**. **`<string.h>`** gives us access to **`strlen()`**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

The third include file is our own **"cdTracker.h"**. **"cdTracker.h"** starts off with three **#define**s that you should know pretty well by now.

```
/***********/
/* Defines */
/***********/
#define kMaxArtistLength        256
#define kMaxTitleLength         256
```

As you make your way through the **cdTracker** source code, you'll notice we've added some decorative comments used to mark the beginning of a section of code. For example, in *cdTracker.h*, we've added comments to mark off areas for defines, **struct** declarations, and function prototypes.

In *main.c*, we've done something similar to set off the beginning of each function. You should do this in your own code. It'll make your code easier to read.

Next comes the new and improved **CDInfo struct** declaration.

```
/**********************/
/* Struct Declarations */
/**********************/
struct CDInfo
{
 char             rating;
 char             artist[ kMaxArtistLength ];
 char             title[ kMaxTitleLength ];
 struct CDInfo    *next;
} *gFirstPtr, *gLastPtr;
```

Notice the two variables hanging off the end of this **struct** declaration. This is a shorthand declaration of two globals, each of which is a pointer to a **CDInfo struct**. We'll use these two globals to keep track of our linked list.

gFirstPtr will always point to the first **struct** in the linked list. **gLastPtr** will always point to the last **struct** in the linked list. We'll use **gFirstPtr** when we want to step through the linked list, starting at the beginning. We'll use **gLastPtr** when we want to add an element to the end of the list. As long as we keep these pointers around, we'll have access to the linked list of memory blocks we'll be allocating.

We could have split this declaration into two parts, like this:

```
struct CDInfo
{
char   rating;
char   artist[ kMaxArtistLength + 1 ];
char   title[ kMaxTitleLength + 1 ];
struct CDInfo   *next;
};
struct CDInfo
*gFirstPtr, *gLastPtr;
```

Either form is fine, though the shorthand version in *cdTracker.h* does a better job of showing that **`gFirstPtr`** and **`gLastPtr`** belong with the **`CDInfo struct`** declaration.

cdTracker.h ends with a series of function prototypes:

```
/***********************/
/* Function Prototypes */
/***********************/
char              GetCommand( void );
struct CDInfo     *ReadStruct( void );
void      AddToList( struct CDInfo *curPtr );
void              ListCDs( void );
void              Flush( void );
```

Let's get back to *main.c*. **`main()`** defines a **`char`** named **`command`** which will be used to hold the single-letter command typed by the user.

```
/********************************************
 ***> main <*/
int main (int argc, const char * argv[])
{
 char                        command;
```

Next, the variables **`gFirstPtr`** and **`gLastPtr`** are set to a value of **`NULL`**. As defined earlier, **`NULL`** indicates that these pointers do not point to valid memory addresses. Once we add an item to the list, these pointers will no longer be **`NULL`**.

```
 gFirstPtr = NULL;
 gLastPtr = NULL;
```

Next, **`main()`** enters a **`while`** loop, calling the function **`GetCommand()`**. **`GetCommand()`** prompts you for a one-character command, either a **`'q'`**, **`'n'`**, or **`'l'`**. Once **`GetCommand()`** returns a **`'q'`**, we drop out of the **`while`** loop and exit the program.

```
 while ( (command = GetCommand() ) != 'q' )
 {
```

If **`GetCommand()`** returns an **`'n'`**, the user wants to enter information on a new CD. First we call **`ReadStruct()`**, which allocates space for a **`CDInfo struct`**, then prompts the user for the information to place in the new **`struct`**'s fields. Once the **`struct`** is filled out, **`ReadStruct()`** returns a pointer to the newly allocated **`struct`**.

The pointer returned by **`ReadStruct()`** is passed on to **`AddToList()`**, which adds the new **`struct`** to the linked list.

```
		switch( command )
		{
			case 'n':
				AddToList( ReadStruct() );
				break;
```

If **`GetCommand()`** returns an **`'l'`**, the user wants to list all the CDs in his or her collection. That's what the function **`ListCDs()`** does.

```
			case 'l':
				ListCDs();
				break;
		}
	}
```

Before the program exits, it says **"Goodbye..."**.

```
	printf( "Goodbye..." );

	return 0;
}
```

Next up on the panel is **`GetCommand()`**. **`GetCommand()`** declares a **`char`** named **`command`**, used to hold the user's command.

```
/****************************************>
 GetCommand <*/
char GetCommand( void )
{
 char      command;
```

Because we want to execute the body of this next loop at least once, we used a **`do`** loop instead of a **`while`** loop. We'll first prompt the user to enter a command, then use **`scanf()`** to read a character from the input buffer. The function **`Flush()`** will read characters, one at a time, from the input buffer until it reads in a carriage return. If we didn't call **`Flush()`**, any extra characters we typed after the command (including the **`'\n'`**) would be picked up the next time through this loop and extra prompt lines would appear, one per extra character. To see this effect, comment out the call to **`Flush()`** and type more than one character when prompted for a command.

```
do
{
   printf( "Enter command (q=quit, n=new,
l=list):  " );
   scanf( "%c", &command );
   Flush();
}
while ( (command != 'q') && (command != 'n')
                         && (command != 'l')
);
```

We'll drop out of the loop once we get either a **`'q'`**, an **`'n'`**, or an **`'l'`**.

Here's a cool trick Keith Rollin (C guru extraordinaire) showed me. Instead of ending the **do** loop with this statement:

```
while ( (command != 'q') && (command !=
'n') && (command != 'l') );
```

try this code instead:

```
while ( ! strchr( "qnl", command ) );
```

`strchr()` takes two parameters: a **0** terminated string and an **`int`** containing a character. It searches the string for the character and returns a pointer to the character inside the string, if it was found. If the character wasn't in the string, **`strchr()`** returns **NULL**. Pretty cool, eh?

Once we drop out of the loop, we'll print a separator line and return the single-letter command.

```
 printf( "\n----------\n" );
 return( command );
}
```

Next up is **`ReadStruct()`**. Notice the unusual declaration of the function name.

```
/****************************************>
 ReadStruct <*/
struct CDInfo    *ReadStruct( void )
{
```

This line says that **`ReadStruct()`** returns a pointer to a **`CDInfo struct`**:

```
struct CDInfo    *ReadStruct( void )
```

`ReadStruct()` uses **`malloc()`** to allocate a block of memory the size of a **`CDInfo struct`**. The variable **`infoPtr`** will act as a pointer to the new block. We'll use the variable **`num`** to read in the rating which we'll eventually store in **`infoPtr->rating`**. **`result`** is a dummy variable we'll never really make use of. It exists because we needed a variable to catch the value returned by **`fgets()`**. Since **`fgets()`** also puts the same value in one of its parameters, we won't need the value returned to **`result`**.

```
struct CDInfo    *infoPtr;
int              num;
char             *result;
```

ReadStruct() calls **malloc()** to allocate a **CDInfo struct**, assigning the address of the block returned to **infoPtr**.

```
infoPtr = malloc( sizeof( struct CDInfo ) );
```

If **malloc()** cannot allocate a block of the requested size, it will return a value of **NULL**. If this happens, we'll print an appropriate message and call the Standard Library function **exit()**. As its name implies, **exit()** causes the program to immediately exit.

```
if ( infoPtr == NULL )
{
   printf( "Out of memory!!!  Goodbye!\n" );
   exit( 0 );
}
```

The parameter you pass to **exit()** will be passed back to the operating system (or to whatever program launched your program).

If we're still here, **malloc()** must have succeeded. Next, we'll print a prompt for the CD artist's name, then call **fgets()** to read a line from the input buffer. **fgets()** will place the line in the **artist** field of the newly allocated **struct**.

```
printf( "Enter Artist's Name:  " );
result = fgets( infoPtr->artist,
kMaxArtistLength, stdin );
```

Earlier in the chapter (in **multiArray**), we discovered that **fgets()** leaves the **'\n'** in place when it reads in a line of input. In this next line, we use **strlen()** and the **=** operator to replace the **'\n'** with a terminating **'\0'**.

As a reminder of how this works, imagine that the line typed in was **"hello"**, with a carriage return acting as the sixth character in the string. This means that **infoPtr->artist[5]** is the character that needs to be replaced.

In this case, strlen(infoPtr->artist) returns 6 (the characters **"hello"**plus the **'\n'**). We subtract 1 to get 5. Now we'll use the **=** operator to replace the **'\n'** at **infoPtr->artist[5]** with a terminating **'\0'**.

```
infoPtr->artist[ strlen( infoPtr->artist ) -
1 ] = '\0';
```

We then repeat the process to prompt for and read in the CD title.

```
printf( "Enter CD Title:  " );
result = fgets( infoPtr->title,
kMaxTitleLength, stdin );
infoPtr->title[ strlen( infoPtr->title ) - 1
] = '\0';
```

This loop prompts the user to enter a number between 1 and 10. We then use **`scanf()`** to read an **`int`** from the input buffer. Note that we used a temporary **`int`** to read in the number instead of reading it directly into **`infoPtr->rating`**. We did this because the **`%d`** format specifier expects an **`int`** and **`rating`** is declared as a **`char`**. Once we read the number, we call **`Flush()`** to get rid of any other characters (including the **`'\n'`**).

```
do
{
    printf( "Enter CD Rating (1-10):  " );
    scanf( "%d", &num );
    Flush();
}
while ( ( num < 1 ) || ( num > 10 ) );
```

This **`do`** loop is not as careful as it could be. If **`scanf()`** encounters an error of some kind, **`num`** will end up with an undefined value. If that undefined value happens to be between 1 and 10, the loop will exit and an unwanted value will be entered in the **`rating`** field. Though that might not be that big a deal in our case, we probably would want to drop out of the loop or, at the very least, print some kind of error message if this happens.

Here's another version of the same code:

```
do
{
  printf( "Enter CD Rating (1-10): " );
  if ( scanf( "%d", &num ) != 1 )
  {
    printf( "Error returned by
scanf()!\n" );
    exit( -1 );
  };
  Flush();
}
while ( ( num < 1 ) || ( num > 10 ) );
```

`scanf()` returns the number of items it read. Since we've asked it to read a single **`int`**, this version prints an error message and exits if we don't read exactly one item. This is a pretty simplistic error strategy, but it does make a point. Pay attention to error conditions and to function return values.

Once a number is read in that's between 1 and 10, the number is assigned to the **`rating`** field of the newly allocated **`struct`**.

```
  infoPtr->rating = num;
```

Finally, a separating line is printed and the pointer to the new **`struct`** is returned.

```
  printf( "\n----------\n" );

  return( infoPtr );
}
```

`AddToList()` takes a pointer to a **`CDInfo`** **`struct`** as a parameter. It uses the pointer to add the **`struct`** to the linked list.

```
/****************************************>
 AddToList <*/
void AddToList( struct CDInfo *curPtr )
{
```

If **`gFirstPtr`** is **`NULL`**, the list must be empty. If so, make **`gFirstPtr`** point to the new **`struct`**.

```
  if ( gFirstPtr == NULL )
    gFirstPtr = curPtr;
```

If **`gFirstPtr`** is not **`NULL`**, there's at least one element in the linked list. In that case, make the **`next`** field of the very last element on the list point to the new **`struct`**.

```
  else
    gLastPtr->next = curPtr;
```

In either case, set **`gLastPtr`** to point to the new "last element in the list." Finally, make sure the **`next`** field of the last element in the list is **`NULL`**. You'll see why we did this in the next function, **`ListCDs()`**.

```
  gLastPtr = curPtr;
  curPtr->next = NULL;
}
```

`ListCDs()` lists all the CDs in the linked list. The variable **`curPtr`** is used to point to the link element currently being looked at.

```
/****************************************>
 ListCDs <*/
void ListCDs( void )
{
 struct CDInfo   *curPtr;
```

If no CDs have been entered yet, we'll print an appropriate message.

```
 if ( gFirstPtr == NULL )
 {
    printf( "No CDs have been entered yet...\
n" );
    printf( "\n----------\n" );
 }
```

Otherwise we'll use a **for** loop to step through the linked list. The **for** loop starts by setting **curPtr** to point to the first element in the linked list and continues as long as **curPtr** is not **NULL**. Each time through the loop, **curPtr** is set to point to the next element in the list. Since we make sure that the last element's **next** pointer is always set to **NULL**, When **curPtr** is equal to **NULL**, we know we have been through every element in the list and we are done.

```
 else
 {
    for ( curPtr=gFirstPtr; curPtr!=NULL;
 curPtr = curPtr->next )
    {
```

The first two **printf()**s use the **"%s"** format specifier to print the strings in the fields **artist** and **title**.

```
       printf( "Artist: %s\n", curPtr->artist );
       printf( "Title:  %s\n", curPtr->title );
```

Next, the **rating** field and a separating line are printed and it's back to the top of the loop.

```
       printf( "Rating: %d\n", curPtr->rating );

       printf( "\n----------\n" );
    }
  }
}
```

Flush() uses **getchar()** to read characters from the input buffer until it reads in a carriage return. **Flush()** is a good utility routine to have around.

```
/****************************************>
 Flush <*/
void Flush( void )
{
 while ( getchar() != '\n' )
    ;
}
```

`Flush()` was based on the Standard Library function `fflush()`. `fflush()` flushes the input buffer associated with a specific file. Since we haven't gotten into files yet, we wrote our own version, though as you can see, it wasn't that hard.

What's Next?

This chapter covered a wide range of topics, from `#include`s to linked lists. The intent of the chapter, however, was to attack a real-world programming problem; in this case, a program to catalog CDs. The chapter showed several design approaches, discussing the pros and cons of each. Finally, the chapter presented a prototype for a CD-tracking program. The program allows you to enter information about a series of CDs and, on request, will present a list of all the CDs tracked.

One problem with this program is that once you exit, all of the data you entered is lost. The next time you run the program, you have to start all over again.

Chapter 10 offers a solution to this problem. The chapter introduces the concept of files and file management, showing you how to save your data from memory out to your hard-disk drive and how to read your data back in again. The chapter updates `cdTracker`, storing the CD information collected in a file on your disk drive.

Exercises

1) What's wrong with each of the following code fragments:

a)

```
struct Employee
{
   char   name[ 20 ];
   int    employeeNumber
};
```

b)

```
while ( getchar() == '\n' ) ;
```

c)

```
#include "stdio.h"
```

d)

```
struct Link
{
   name[ 50 ];
   Link   *next;
};
```

e)

```
struct Link
{
   struct Link  next;
   struct Link prev;
}
```

f)

```
StepAndPrint( char *line )
{
while ( *line != 0 )
   line++;

printf( "%s", line );
}
```

2) Update **`cdTracker`** so it maintains its linked list in order from the lowest rating to the highest rating. If two CDs have the same rating, the order is unimportant.

3) Update **`cdTracker`**, adding a **`prev`** field to the **`CDInfo struct`** so it maintains a doubly-linked list. As before, the next field will point to the next link in the list. Now, however, the **`prev`** field should point to the previous link in the list. Add an option to the menu that prints the list backward, from the last struct in the list to the first.

Chapter 10 Working with Files

Chapter 9 introduced **`cdTracker`**, a program designed to keep track of your compact disc collection. **`cdTracker`** allowed you to enter a new CD, as well as list all existing CDs. **`cdTracker`**'s biggest shortcoming was that it didn't save the CD information when it exited. If you ran **`cdTracker`**, entered information on ten CDs, and then quit, your information would be gone. The next time you ran **`cdTracker`**, you'd have to start from scratch.

The solution to this problem is to somehow save all of the CD information before you quit the program. This chapter will show you how. Chapter 10 introduces the concept of **files**, the long-term storage for your program's data.

As you move on to other programming languages (such as Objective-C, Java, or C++), sophisticated development toolkits (such as Cocoa), and even other Operating Systems, you'll find there are many ways to work with files. Most of them are based on the concepts you'll learn in this chapter.

Stay with the program! Learn the basics and you'll find moving on to other development platforms much, much easier in the long run.

What is a File?

A file is a series of bytes residing in some storage media. Files can be stored on your hard drive, on a recordable CD or DVD, or even on your iPod. The iTunes application is made up of a collection of files, including the actual executable, the preference files, and all the song files. Your favorite word processor lives in a file, and so does each and every document you create with your word processor.

The project archive that came with this book contains many different files. Apple's developer tools are made up of hundreds of files. Each of the *Learn C* projects consists of at least two files: a project file and at least one source code file. When you compile and link a project, you produce a new kind of file, an application file.

All of these are examples of the same thing: a collection of bytes known as a file.

All of the files on your computer share a common set of traits. For example, each file has a size. The file *main.c* from the **`cdTracker`** project has a size of 2,425 bytes. The *main.c* from **`multiArray`** was only 913 bytes. Each of these files resides on my Mac's internal hard drive.

Working With Files, Part One

In the C world, each file consists of a **stream** of consecutive bytes. When you want to access the data in a file, you first **open** the file using a Standard Library function named **`fopen()`**, pronounced *eff-open*. Once your file is open, you can **read** data from the file or **write** new data back into the file using Standard Library functions like **`fscanf()`** and **`fprintf()`**. Once you are done working with your file, you'll close it using the Standard Library function **`fclose()`**.

Opening and Closing a File

Here's the function prototype for **`fopen()`**, found in the file **`<stdio.h>`**:

```
FILE *fopen( const char *name, const char
  *mode );
```

The **`const`** keyword marks a variable or parameter as read-only. In other words, **`fopen()`** is not allowed to modify the array of characters pointed at by **`name`** or **`mode`**. Here's another example:

```
const int

myInt = 27;
```

This declaration creates an **`int`** named **`myInt`** and assigns it a value of 27 (we'll talk about definitions that also initialize in Chapter 11). More importantly, the value of **`myInt`** is now permanently set. **`myInt`** is now read-only. As long as **`myInt`** remains in scope, you can't change its value.

The first parameter, **`name`**, tells **`fopen()`** which file you want to open. For example, the file name **`"My Data File"`** tells **`fopen()`** to look in the current folder (the folder containing the currently running application) for a file named *My Data File*.

The "**`/`**" (slash), "**`.`**" (dot), and "**`~`**" (tilde) characters have a special meaning when naming Unix and Mac OS X files. The "**`.`**" refers to the current folder, the "**`/`**" is a directory separator, and the "**`~`**" specifies your home directory.

For example, if I wanted to refer to the file "**`My Data File`**" in the current directory, I'd use the string **`"./My Data File"`**. The string **`"/My Data File"`** refers to the file named "**`My Data File`**" at the very top level of your hard drive. This top level is also known as the **root level** of your hard drive.

Two dots in a row refer to the parent directory of the current directory. So the string **`"../My Data File"`** refers to the file named "**`My Data File`**" one level up from the current directory.

The string **`"~/My Data File"`** refers to the file named "**`My Data File`**" in your home directory. On my Mac, my home directory is the directory **`/Users/davemark`**.

As you make your way through the programs in this chapter, play with the file names till you understand these concepts.

The second parameter, **`mode`**, tells **`fopen()`** how you'll be accessing the file. The three basic file modes are **`"r"`**, **`"w"`**, and **`"a"`**, which stand for **read**, **write**, and **append**, respectively.

`"r"` tells **`fopen()`** that you want to read data from the file and that you won't be writing to the file at all. The file must already exist in order to use this mode.

In other words, you can't use the mode **"r"** to create a file.

The mode **"w"** tells **fopen()** that you want to write to the specified file. If the file doesn't exist yet, a new file with the specified name is created. If the file does exist, **fopen()** deletes it and creates a new empty file for you to write into.

This last point bears repeating. Calling **fopen()** with a mode of **"w"** will delete a file's contents if the file already exists, essentially starting you over from the beginning of the file. Be careful!

The mode **"a"** is similar to **"w"**. It tells **fopen()** that you want to write to the specified file and to create the file if it doesn't exist. If the file does exist, however, the data you write to the file is appended to the end of the file.

If **fopen()** successfully opens the specified file, it allocates a **struct** of type **FILE** and returns a pointer to the **FILE struct**. The **FILE struct** contains information about the open file, including the current mode (**"r"**, **"w"**, **"a"** or whatever) as well as the current **file position**. The file position is a pointer into the file that acts like a bookmark in a book. When you open a file for reading, for example, the file position points to the first byte in the file. When you read the first byte, the file position moves to the next byte.

It's not really important to know the details of the **FILE struct**. All you need to do is keep track of the **FILE** pointer returned by **fopen()**. By passing the pointer to a Standard Library function that reads or writes, you'll be sure the read or write takes place in the right file and at the right file position. You'll see how all this works as we go through the chapter sample code.

Here's a sample **fopen()** call:

```
FILE *fp;

if ( (fp = fopen( "My Data File", "r")) ==
 NULL )
{
 printf( "File doesn't exist!!!\n" );
 exit(1);
}
```

This code first calls **fopen()**, attempting to open the file named "**My Data File**" for reading. If **fopen()** cannot open the file for some reason (perhaps you've asked it to open a file that doesn't exist or you've already opened the maximum number of files - see the next tech block), it returns **NULL**. In that case, we'll print an error message and exit.

There is a limit to the number of simultaneous open files. This limit is implemented as a constant, **FOPEN_MAX**, defined in the file **<stdio.h>**.

If **fopen()** does manage to open the file, it will allocate the memory for a **FILE struct**, and **fp** will point to that **struct**. We can then pass **fp** to routines that read from the file. Once we're done with the file, we'll pass **fp** to the function **fclose()**:

```
int fclose( FILE *stream );
```

fclose() takes a pointer to a **FILE** as a parameter and attempts to close the specified file. If the file is closed successfully, **fclose()** frees up the memory allocated to the **FILE struct** and returns a value of **0**. It is very important that you match every **fopen()** with a corresponding **fclose()**, otherwise you'll end up with unneeded **FILE structs** floating around in memory.

In addition, once you've passed a **FILE** pointer to **fclose()**, that **FILE** pointer no longer points to a **FILE struct**. If you want to access the file again, you'll have to make another **fopen()** call.

If **fclose()** fails, it returns a value of -1. Many programmers ignore the value returned by **fclose()**, since there's not a whole lot you can do about it. On the other hand, you can never have too much error checking in your code, so you might consider checking the value returned by **fclose()** and, at the very least, printing an appropriate error message if **fclose()** fails.

Reading a File

Once you open a file for reading, the next step is to read data from the file. There are several Standard Library functions to help you do just that. For starters, the function **fgetc()** reads a single character from a file's input buffer. Here's the function prototype:

```
int fgetc( FILE *fp );
```

The single parameter is the **FILE** pointer returned by **fopen()**. **fgetc()** reads a single character from the file and advances the file position pointer. If the file position pointer is already at the end of the file, **fgetc()** returns the constant **EOF**.

Though **fgetc()** returns an **int**, a line like this:

```
char   c;

c = fgetc( fp );
```

works just fine. When the C compiler encounters two different types on each side of an assignment operator, it does its best to convert the value on the right side to the type of the left side before doing the assignment. As long as the type of the right side is no larger than the type of the left side (as is the case here: an **int** is at least as large as a **char**) this won't be a problem.

We'll get into the specifics of typecasting in Chapter 11.

The function **fgets()**, which we made use of in Chapter 9, reads a series of characters into an array of **char**s. Here's the function prototype:

```
char *fgets( char *s, int n, FILE *fp );
```

The first parameter is a pointer to an array of **char**s that you've already allocated. Don't just declare a (**char ***) and pass it in to **fgets()**. Instead, allocate an array of **char**s large enough to hold the largest block of **char**s you might end up reading in, then pass a pointer to that array as the first parameter (you'll see an example in a second).

The second parameter is the maximum number of characters you'd like to read. **fgets()** stops reading once it reads in **n-1 char**s, or if it encounters an end-of-file or a **'\n'** before it reads **n-1 char**s. If **fgets()** successfully reads **n-1 char**s, it appends a **0** terminator to the **char** array (that's why the array has to be at least **n char**s in size).

If **fgets()** encounters a **'\n'** before it reads **n-1 char**s, it stops reading after the **'\n'** is read, then adds the **0** terminator to the array, right after the **'\n'**.

If **fgets()** encounters an end-of-file before it reads **n-1 char**s, it adds the **0** terminator to the array, right after the last character read.

If **fgets()** encounters an end-of-file before it reads in any **char**s, it returns **NULL**. Otherwise, **fgets()** returns a pointer to the **char** array.

Finally, the third parameter is the **FILE** pointer returned by **fopen()**.

Here's an example:

```
#define kMaxBufferSize          200

FILE        *fp;
char        buffer[ kMaxBufferSize ];

if ( (fp = fopen( "My Data File", "r")) ==
 NULL )
{
 printf( "File doesn't exist!!!\n" );
 exit(1);
}
```

```
if ( fgets( buffer, kMaxBufferSize, fp ) ==
 NULL )
{
 if ( feof( fp ) )
    printf( "End-of-file!!!\n" );
 else
    printf( "Unknown error!!!\n" );
}
else
 printf( "File contents: %s\n", buffer );
```

Notice that the example calls a function named **`feof()`** if **`fgets()`** returns **`NULL`**. **`fgets()`** returns **`NULL`** no matter what error it encounters. **`feof()`** returns **`true`** if the last read on the specified file resulted in an end-of-file, **`false`** otherwise.

The function **`fscanf()`** is similar to **`scanf()`**, reading from a file instead of the keyboard. Here's the prototype:

```
int  fscanf( FILE *fp, const char* format, ...
 );
```

The first parameter is the **`FILE`** pointer returned by **`fopen()`**. The second parameter is a format specification embedded inside a character string. The format specification tells **`fscanf()`** what kind of data you want read from the file. The **`...`** operator in a parameter list tells the compiler that 0 or more parameters may follow the second parameter. Like **`scanf()`** and **`printf()`**, **`fscanf()`** uses the format specification to determine the number of parameters it expects to see. Be sure to pass the correct number of parameters or your program will get confused.

These are a few of the file access functions provided by the Standard Library. Wanna look up something? Here's that link to that online Standard Library reference I keep mentioning:

http://www.infosys.utas.edu.au/info/documentation/C/CStdLib.html

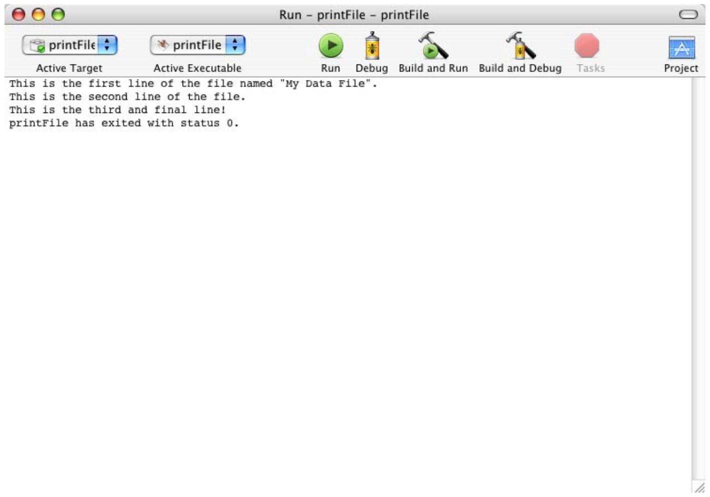
Click on the link to **`<stdio.h>`** at the top of the page. You might also want to take a look at *C, A Reference Manual* by Harbison and Steele and check out Chapter 15, entitled "Input/Output Facilities".

In the meantime, here's an example that uses the functions **`fopen()`** and **`fgetc()`** to open a file and display its contents.

printFile.xcode

`printFile` opens a file named *My Data File*, reads in all the data from the file, one character at a time, and prints each character in the console window.

Open the *Learn C Projects* folder, go inside the folder *10.01 - printFile*, and open and run the project *printFile.xcode*. Compare your output with the console window shown in Figure 10.1. They should be the same.



```
This is the first line of the file named "My Data File".
This is the second line of the file.
This is the third and final line!
printFile has exited with status 0.
```

***Figure 10.1** The* **`printFile`** *output, showing the contents of the file* My Data File.

Let's take a look at the data file read in by **`printFile`**. Select Open... from Xcode's File menu. Xcode will prompt you for a text file to open. Be sure you are in the *10.01 - printFile* directory and select the file named *My Data File*. An editing window will open allowing you to edit the contents of *My Data File*. Feel free to make some changes to the file and run the program again. Make sure you don't change the name *or* the location of the file.

Let's take a look at the source code.

Stepping Through the Source Code

Open the source code file *main.c* by double-clicking on its name in the project window. Take a minute to look over the source code. Once you feel comfortable with it, read on.

main.c starts off with the usual **`#include`**.

```
#include <stdio.h>
```

`main()` defines two variables. **`fp`** is our **`FILE`** pointer, and **`c`** is an **`int`** that will hold the **`char`**s we read from the file.

```
int main (int argc, const char * argv[])
{
 FILE              *fp;
 int       c;
```

This call of the function **`fopen()`** opens the file named *My Data File* for reading, returning the file pointer to the variable **`fp`**.

```
 fp = fopen( "../My Data File", "r" );
```

Notice the "`../`" in the beginning of the file name we passed to **`fopen()`**. As described earlier, the "`../`" means that the file is in the parent directory, one level up from the application. But wait. The file *My Data File* is in the same directory as the project file. What gives?

This behavior is specific to the way applications are built under Mac OS X. Though an application might look like a single file, it really is a directory with a series of files embedded in it, including the executable, all packaged together to look like a single file. Since the executable is actually buried one level deep within the package, we need to pop up one level to find *My Data File*.

If we built this program as a Unix binary using the Terminal app, we'd need to remove the "`../`" from the beginning of the file name, since Unix apps do not do the fakeout package trick.

Just thought you'd like to know.

If **`fp`** is not **`NULL`**, the file was opened successfully.

```
if ( fp != NULL )
{
```

The **`while`** loop continuously calls **`fgetc()`**, passing it the file pointer **`fp`**. **`fgetc()`** returns the next character in **`fp`**'s input buffer. The returned character is assigned to **`c`**. If **`c`** is not equal to **`EOF`**, **`putchar()`** is called, taking **`c`** as a parameter.

```
while ( (c = fgetc( fp )) != EOF )
    putchar( c );
```

`putchar()` prints the specified character to the console window. We could have accomplished the same thing by using **`printf()`**:

```
printf( "%c", c );
```

As you program, you'll often find two different solutions to the same problem. Should you use **`putchar()`** or **`printf()`**? If performance is critical, pick the option that is more specific to your particular need. In this case, **`printf()`** is designed to handle many different data types. **`putchar()`** is designed to handle one data type, an **`int`**. Chances are the source code for **`putchar()`** is simpler and more efficient than the source code for **`printf()`** *when it comes to printing an* **`int`**. If performance is critical, you might want to use **`putchar()`** instead of **`printf()`**. If performance isn't critical, go with your own preference.

Once we are done, we'll close the file by calling **`fclose()`**. Remember to always balance each call of **`fopen()`** with a corresponding call to **`fclose()`**.

```
        fclose( fp );
    }

    return 0;
}
```

stdin, stdout, and stderr

C provides you with three **FILE** pointers that are always available and always open. **stdin** represents the keyboard, **stdout** represents the console window, and **stderr** represents the file where the user wants all error messages sent. **stdin**, **stdout**, and **stderr** are normally associated with command line oriented operating systems like Unix and DOS and are rarely used on the Macintosh, but it's definitely worth knowing about them.

In **printFile**, we used the function **fgetc()** to read a character from a previously opened file. This line:

```
c = fgetc( stdin );
```

will read the next character from the keyboard's input buffer.

```
fgetc( stdin )
```

is equivalent to calling

```
getchar()
```

As you'll see in the next few sections, whenever C provides a mechanism for reading or writing to a file, C will also provide a similar mechanism for reading from **stdin** or writing to **stdout**.

Working With Files, Part Two

So far, you've learned how to open a file using **`fopen()`** and how to read from a file using **`fgetc()`**. You've seen, once again, that you can often use two different functions to solve the same problem. Now let's look at some functions that allow you to write data out to a file.

Writing to a File

The Standard Library offers several functions that write data out to a previously opened file. This section will introduce three of them: **`fputc()`**, **`fputs()`**, and **`fprintf()`**.

`fputc()` takes an **`int`** holding a character value, and writes the character out to the specified file. **`fputc()`** is declared as follows:

```
int  fputc( int c, FILE *fp );
```

If **`fputc()`** successfully writes the character out to the file, it returns the value passed to it in the parameter **`c`**. If the write fails for some reason, **`fputc()`** returns the value **`EOF`**.

Note that:

```
fputc( c, stdout );
```

is the same as calling:

```
putchar( c );
```

`fputs()` is similar to **`fputc()`**, but writes out a **`0`**-terminated string instead of a single character. **`fputs()`** is declared as follows:

```
int  fputs( const char *s, FILE *fp );
```

`fputs()` writes out all the characters in the string, but does not write out the terminating **`0`**. If the write succeeds, **`fputs()`** returns a **`0`**. If the write fails, **`fputs()`** returns **`EOF`**.

`fprintf()` works just like **`printf()`**. Instead of sending its output to the console window, **`fprintf()`** writes its output to the specified file. **`fprintf()`** is declared as follows:

```
int  fprintf( FILE *fp, const char *format,
  ... );
```

The first parameter specifies the file to be written to. The second is the format specification text string. Any further parameters depend on the contents of the format specification string.

cdFiler.xcode

In Chapter 9, we ran **`cdTracker`**, a program designed to help you track your CD collection. The big shortcoming of **`cdTracker`** is its inability to save your carefully entered CD data. As you quit the program, the CD information you entered gets discarded, forcing you to start over the next time you run **`cdTracker`**.

Our next program, **`cdFiler`**, solves this problem by adding two special functions to **`cdTracker`**. **`ReadFile()`** opens a file named *cdData*, reads in the CD data from the file, and uses the data to build a linked list of **`cdInfo structs`**. **`WriteFile()`** writes the linked list back out to the file.

Open the *Learn C Projects* folder, go inside the folder *10.02 - cdFiler*, and open the project *cdFiler.xcode*. Check out the *cdFiler.xcode* project window shown in Figure 10.2. Notice that there are three separate source code files (two *.c* files and one *.h* file). Your project can contain as many source code files as you like. Just make sure that only one of the files has a function named **`main()`**, since that's where your program will start.

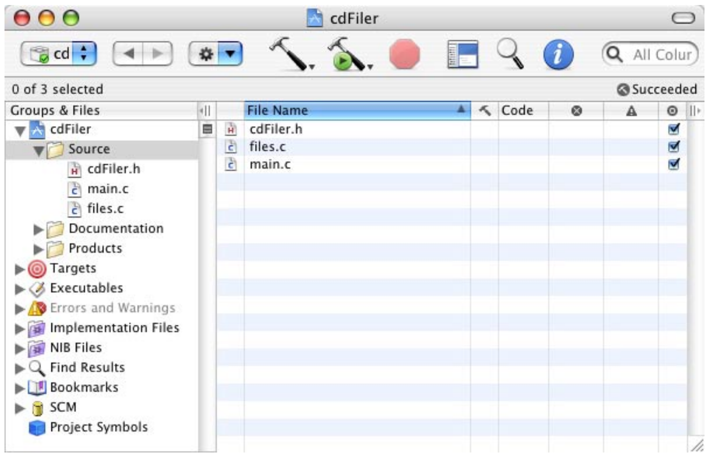


Figure 10.2 *The* **`cdFiler`** *project window.*

The file *main.c* is almost identical to the file *main.c* from Chapter 9's **`cdTracker`** program. The file *files.c* contains the functions that allow **`cdFiler`** to read and write the file *cdData*.

Exploring cdData

Before you run the program, use Xcode to take a quick look at the file *cdData*. At first glance, the contents of the file may not make much sense, but the text does follow a well-defined pattern:

```
Frank Zappa
Anyway the Wind Blows
8
Edith Piaf
The Voice of the Sparrow
```

```
10
Joni Mitchell
For the Roses
9
```

The file is organized in clusters of three lines each. Each cluster contains a one-line CD artist, a one-line CD title, and a one-line numerical CD rating.

The layout of your data files is as important a part of the software design process as the layout of your program's functions. The file described above follows a well-defined pattern. As you lay out a file for your next program, think about the future. Can you live with one-line CD titles? Do you want the ability to add a new CD field, perhaps the date of the CD's release?

The time to think about these types of questions is at the beginning of your program's life, during the design phase.

Running cdFiler

Before you run **`cdFiler`**, close the *cdData* text-editing window.

To create this window, Xcode had to open the file *cdData*. If you don't close the window before you run the program, the file will remain open. When you run **`cdFiler`**, it will also open the file. You'll have the same file open in two places. This is not a good idea.

Why? Suppose you make some changes to the file in Xcode but don't save your changes. Now you run **`cdFiler`** and make some changes, with **`cdFiler`** saving the changes. What happens if you go back to Xcode and save your changes? Most likely, Xcode will overwrite the changes you made using **`cdFiler`** and the **`cdFiler`** changes will be lost. Not good!

Once the window is closed, run **`cdFiler`**. The console window will appear, prompting you for a **`'q'`**, **`'n'`**, or **`'l'`**:

```
Enter command (q=quit, n=new, l=list): l
```

Type an **`l`**, followed by a carriage return. This will list the CDs currently in the program's linked list. If you need a refresher on linked lists, now would be a perfect time to turn back to Chapter 9.

```
Enter command (q=quit, n=new, l=list):  l

----------
Artist:  Frank Zappa
Title:   Anyway the Wind Blows
Rating:  8

----------
Artist:  Edith Piaf
Title:   The Voice of the Sparrow
Rating:  10

----------
```

```
Artist:  Joni Mitchell
Title:   For the Roses
Rating:  9

----------
Enter command (q=quit, n=new, l=list):
```

While Chapter 9's **`cdTracker`** started with an empty linked list, **`cdFiler`** starts with a linked list built from the contents of the *cdData* file. The CDs you just listed should match the CDs you saw when you edited the *cdData* file.

Let's add a fourth CD to the list. Type an **`'n'`** followed by a carriage return:

```
Enter command (q=quit, n=new, l=list): n

----------
Enter Artist's Name: Adrian Belew
Enter CD Title: Mr. Music Head
Enter CD Rating (1-10): 8

----------
Enter command (q=quit, n=new, l=list):
```

Next, type an **`'l'`** to make sure your new CD made it into the list:

```
Enter command (q=quit, n=new, l=list): l

----------
Artist:  Frank Zappa
Title:   Anyway the Wind Blows
Rating:  8

----------
Artist:  Edith Piaf
Title:   The Voice of the Sparrow
Rating:  10

----------
Artist:  Joni Mitchell
Title:   For the Roses
Rating:  9

----------
Artist: Adrian Belew
Title:  Mr. Music Head
Rating: 8

----------
Enter command (q=quit, n=new, l=list):
```

Finally, type a **`'q'`** followed by a carriage return. This causes the program to write the current linked list back out to the file *cdData*. To prove this worked, run **`cdFiler`** one more time. When prompted for a command, type an **`'l'`** to list your current CDs. You should find your new CD nestled at the bottom of the list. Let's see how this works.

Stepping Through the Source Code

The file *cdFiler.h* contains source code that will be included by both *main.c* and *files.c*. The first two **`#define`**s should be familiar to you. The third creates a constant containing the name of the file containing our CD data.

```
/***********/
/* Defines */
/***********/
#define kMaxArtistLength         256
#define kMaxTitleLength          256

#define kCDFileName              "../cdData"
```

This **CDInfo struct** is identical to the one found in **cdTracker**.

```
/**********************/
/* Struct Declarations */
/**********************/
struct CDInfo
{
 char            rating;
 char            artist[ kMaxArtistLength ];
 char            title[ kMaxTitleLength ];
 struct CDInfo   *next;
};
```

Just as we did in **cdTracker**, we've declared two globals to keep track of the beginning and end of our linked list. The **extern** keyword at the beginning of the declaration tells the C compiler to link this declaration to the definition of these two globals, which can be found in *main.c*. If you removed the **extern** keyword from this line, the compiler would first compile *files.c*, defining space for both pointers. When the compiler went to compile *main.c*, it would complain that these globals were already declared.

The **extern** mechanism allows you to declare a global without actually allocating memory for it. Since the **extern** declaration doesn't allocate memory for your globals, you'll need another declaration (usually found in the same file as **main()**) that does allocate memory for the globals. You'll see that declaration in *main.c*.

```
/***********************/
/* Global Declarations */
/***********************/
 extern struct CDInfo   *gFirstPtr, *gLastPtr;
```

Next comes the list of function prototypes. By listing all the functions in this **#include** file, we make all functions available to be called from all other functions. As your programs get larger and more sophisticated, you might want to create a separate include file for each of your source code files. Some programmers create one include file for globals, another for defines, and another for function prototypes.

```
/*******************************/
/* Function Prototypes - main.c */
/*******************************/
char        GetCommand( void );
struct CDInfo     *ReadStruct( void );
void        AddToList( struct CDInfo *curPtr );
void        ListCDs( void );
void        ListCDsInReverse( void );
void        Flush( void );
```

```
/**********************************/
/* Function Prototypes - files.c */
/**********************************/
void WriteFile( void );
void ReadFile( void );
char ReadStructFromFile( FILE *fp, struct
 CDInfo *infoPtr );
```

The file *main.c* is almost exactly the same as the file *main.c* from Chapter 9's **`cdTracker`** program. There are four differences. First, we include the file *cdFiler.h* instead of *cdTracker.h*.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "cdFiler.h"
```

Next, we include the definitions of our two globals directly in this source code file, to go along with the **`extern`** declarations in *cdFiler.h*. This definition is where the memory actually gets allocated for these two global pointers.

```
/**********************/
/* Global Definitions */
/**********************/
struct CDInfo     *gFirstPtr, *gLastPtr;
```

`main()` contains the last two differences. Before we enter the command processing loop, we call **`ReadFile()`** to read in the *cdData* file and turn the contents into a linked list.

```
/************************> main <*/
int main (int argc, const char * argv[])
{
 char            command;

 gFirstPtr = NULL;
 gLastPtr = NULL;

 ReadFile();

 while ( (command = GetCommand() ) != 'q' )
 {
    switch( command )
    {
          case 'n':
                AddToList( ReadStruct() );
                break;
          case 'l':
                ListCDs();
                break;
    }
 }
```

Once we drop out of the loop, we call **`WriteFile()`** to write the linked list out to the file *cdData*.

```
 WriteFile();

 printf( "Goodbye..." );
 return 0;
}
```

For completeness, here's the remainder of *main.c*. Each of these functions are identical to their **`cdTracker`** counterpart.

```
/************************> GetCommand <*/
char GetCommand( void )
{
 char      command;

 do
 {
    printf( "Enter command (q=quit, n=new,
          l=list):  " );
    scanf( "%c", &command );
    Flush();
 }
 while ( (command != 'q') && (command != 'n')
                 && (command != 'l') );

 printf( "\n----------\n" );
 return( command );
}


/************************> ReadStruct <*/
struct CDInfo     *ReadStruct( void )
{
 struct CDInfo    *infoPtr;
 int              num;
 char             *result;

 infoPtr = malloc( sizeof( struct CDInfo ) );

 if ( infoPtr == NULL )
 {
    printf( "Out of memory!!!  Goodbye!\n" );
    exit( 0 );
 }

 printf( "Enter Artist's Name:  " );
 result = fgets( infoPtr->artist,
          kMaxArtistLength, stdin );
 infoPtr->artist[ strlen( infoPtr->artist ) -
          1 ] = '\0';

 printf( "Enter CD Title:  " );
 result = fgets( infoPtr->title,
          kMaxTitleLength, stdin );
 infoPtr->title[ strlen( infoPtr->title ) -
          1 ] = '\0';

 do
 {
    printf( "Enter CD Rating (1-10):  " );
    scanf( "%d", &num );
    Flush();
 }
 while ( ( num < 1 ) || ( num > 10 ) );

 infoPtr->rating = num;

 printf( "\n----------\n" );

 return( infoPtr );
}

/*************************> AddToList <*/
void AddToList( struct CDInfo *curPtr )
{
 if ( gFirstPtr == NULL )
    gFirstPtr = curPtr;
 else
    gLastPtr->next = curPtr;

 gLastPtr = curPtr;
 curPtr->next = NULL;
}


/*****************************> ListCDs <*/
void ListCDs( void )
{
 struct CDInfo    *curPtr;

 if ( gFirstPtr == NULL )
 {
    printf( "No CDs have been entered yet...
          \n" );
    printf( "\n----------\n" );
 }
 else
 {
    for ( curPtr=gFirstPtr; curPtr!=NULL;
 curPtr = curPtr->next )
    {
          printf( "Artist:  %s\n",
                curPtr->artist );
          printf( "Title:   %s\n",
                curPtr->title );
          printf( "Rating:  %d\n",
                curPtr->rating );

          printf( "\n----------\n" );
    }
 }
}


/*****************************> Flush <*/
void Flush( void )
{
 while ( getchar() != '\n' )
    ;
}
```

`files.c` starts out with these **`#include`**s:

```
#include <stdlib.h>
#include <stdio.h>
#include <c.h>
#include "cdFiler.h"
```

`WriteFile()` first checks to see if there are any CDs to write out. If **`gFirstPtr`** is **`NULL`** (the value it was set to in **`main()`**), no CDs have been entered yet and we can just return.

```
/***************************> WriteFile <*/
void WriteFile( void )
{
 FILE              *fp;
 struct CDInfo     *infoPtr;
 int               num;

 if ( gFirstPtr == NULL )
    return;
```

Next, we'll open the file *cdData* for writing. If **`fopen()`** returns **`NULL`**, we know it couldn't open the file and we'll print out an error message and return.

```
    if ( ( fp = fopen( kCDFileName, "w" ) )
              == NULL )
    {
       printf( "***ERROR: Could not write CD
              file!" );
       return;
    }
```

This **for** loop steps through the linked list, setting **infoPtr** to point to the first **struct** in the list, then moving it to point to the **next struct**, and so on, until **infoPtr** is equal to **NULL**. Since the last **struct** in our list sets its **next** pointer to **NULL**, **infoPtr** will be equal to **NULL** when it points to the last **struct** in the list.

```
    for ( infoPtr=gFirstPtr; infoPtr!=NULL;
              infoPtr=infoPtr->next )
    {
```

Each time through the list, we call **fprintf()** to print the **artist** string followed by a carriage return and then the **title** string followed by a carriage return. Remember, each of these strings was **0**-terminated, a requirement if you plan on using the **%s** format specifier.

```
       fprintf( fp, "%s\n", infoPtr->artist );
       fprintf( fp, "%s\n", infoPtr->title );
```

Finally, we convert the **rating** field to an **int** by assigning it to the **int num**, then print it (as well as a following carriage return) to the file using **fprintf()**. We converted the **char** to an **int** because the **%d** format specifier was designed to work with an **int**, and not a **char**.

```
       num = infoPtr->rating;
       fprintf( fp, "%d\n", num );
    }
```

Once we finish writing the linked list into the file, we'll close the file by calling **fclose()**.

```
    fclose( fp );
  }
```

ReadFile() starts by opening the file *cdData* for reading. If we can't open the file, we'll print an error message and return, leaving the list empty.

```
/***************************> ReadFile <*/
void ReadFile( void )
{
  FILE              *fp;
  struct CDInfo     *infoPtr;
  int               i;

  if ( ( fp = fopen( kCDFileName, "r" ) )
            == NULL )
  {
        printf( "***ERROR: Could not read CD
                    file!" );
        return;
    }
```

With the file open, we'll enter a loop that continues as long as **`ReadStructFromFile()`** returns **`true`**. By using the **`do`**-**`while`** loop, we'll execute the body of the loop before we call **`ReadStructFromFile()`** for the first time. This is what we want. The body of the loop attempts to allocate a block of memory the size of a **`CDInfo struct`**. If the **`malloc()`** fails, we'll bail out of the program.

```
    do
    {
      infoPtr = malloc( sizeof( struct CDInfo ));

      if ( infoPtr == NULL )
      {
        printf( "Out of memory!!!  Goodbye!\n" );
        exit( 0 );
      }
    }
    while ( ReadStructFromFile( fp, infoPtr ) );
```

`ReadStructFromFile()` will return **`false`** when it hits the end of the file, when it can't read another set of **`CDInfo`** fields. In that case, we'll close the file and free up the last block we just allocated, since we have nothing to store in it.

```
    fclose( fp );
    free( infoPtr );
  }
```

`ReadStructFromFile()` uses a funky form of **`fscanf()`** to read in the first two **`CDInfo`** fields. Notice the use of the format descriptor **`"%[^\n]\n"`**. This tells **`fscanf()`** to read characters from the specified file until it hits a **`'\n'`**, then to read the **`'\n'`** character and stop. The characters **`[^\n]`** represent the set of all characters except **`'\n'`**. Note that the **`%[`** format specifier places a terminating **`0`**-byte at the end of the characters it reads in.

```
/*******************> ReadStructFromFile <*/
char ReadStructFromFile( FILE *fp, struct
          CDInfo *infoPtr )
{
  int       num;

  if ( fscanf( fp, "%[^\n]\n",
          infoPtr->artist ) != EOF )
  {
```

The square brackets inside a format specifier give you much greater control over **`scanf()`**. For example, the format specifier **`"%[abcd]"`** would tell **`scanf()`** to keep reading as long as it was reading either an **`'a'`**, a **`'b'`**, a **`'c'`**, or a **`'d'`**. The first non-**`[abcd]`** character would be left in the input buffer for the next part of the format specifier or for the next read operation to pick up.

If the first character in the set is the character "^", The set represents the characters that do not belong to the set. In other words, the format specifier **`"%[^abcd]"`**, tells **`scanf()`** to continue reading as long as it doesn't encounter any of the characters **`'a'`**, **`'b'`**, **`'c'`**, or **`'d'`**.

If **`fscanf()`** hits the end of the file, we'll return **`false`**, letting the calling function know there are no more fields to read. If **`fscanf()`** succeeds, we'll move on to the **`title`** field using the same technique. If this second **`fscanf()`** fails, we've got a problem, since we read an **`artist`**, but couldn't read a **`title`**.

```
    if ( fscanf( fp, "%[^\n]\n",
                infoPtr->title ) == EOF )
    {
          printf( "Missing CD title!\n" );
          return false;
    }
```

Assuming we got both the **`artist`** and **`title`**, we'll use a more normal format specifier to pick up an **`int`** and the third carriage return.

```
    else if ( fscanf( fp, "%d\n", &num ) ==
                EOF )
    {
          printf( "Missing CD rating!\n" );
          return false;
    }
```

Assuming we picked up the **`int`**, we'll use the assignment operator to convert the **`int`** to a **`char`** and add the now complete **`struct`** to the list by passing it to **`AddToList()`**.

```
    else
    {
          infoPtr->rating = num;
          AddToList( infoPtr );
          return true;
    }
  }
  else
    return false;
}
```

Working With Files, Part Three

Now that you've mastered the basics of file reading and writing, there are a few more topics worth exploring before we leave this chapter. We'll start off with a look at some additional file opening modes.

The "Update" Modes

So far, you've encountered the three basic file opening modes: **`"r"`**, **`"w"`**, and **`"a"`**. Each of these modes has a corresponding **update** mode, specified by adding a "**+**" to the mode. The three update modes, **`"r+"`**, **`"w+"`**, and **`"a+"`**, each allow you to open a file for both reading and writing.

Though the three update modes do allow you to switch between read and write operations without reopening the file, you must first call either **`fsetpos()`**, **`fseek()`**, **`rewind()`**, or **`fflush()`** before you make the switch.

In other words, if your file is opened using one of the update modes, you can't call **`fscanf()`** and then call **`fprintf()`** (or call **`fprintf()`** followed by **`fscanf()`**) unless you call **`fsetpos()`**, **`fseek()`**, **`rewind()`**, or **`fflush()`** in between.

There is a great chart in Harbison and Steele's *C: A Reference Manual* which summarizes these modes quite nicely. My version of the chart is found in Figure 10.3. Before you read on, take a minute to look the chart over to be sure you understand the different file modes.

Mode Rules	"r"	"w"	"a"	"r"	"w"	"a"
Named file must already exist	yes	no	no	yes	no	no
Existing file's contents are lost	no	yes	no	no	yes	no
Read OK	yes	no	no	yes	yes	yes
Write OK	no	yes	yes	yes	yes	yes
Write begins at end of file	no	no	yes	no	no	yes

Figure 10.3 *My version of the Harbison and Steele file mode chart showing the rules associated with each of the 6 basic file opening modes.*

C also allows a file mode to specify whether a file is limited to ASCII characters (text mode) or is allowed to hold any type of data at all (binary mode). To open a file in text mode, just append a "**`t`**" at the end of the mode string (like **`"rt"`** or **`"w+t"`**). To open a file in binary mode, append a "**`b`**" at the end of the mode string (like **`"rb"`** or **`"w+b"`**).

If you use a file mode that doesn't include a "**`t`**" or a "**`b`**", check your development environment doc to find out which of the two types is the default.

Random File Access

So far, each of the examples presented in this chapter have treated files as a **sequential stream of bytes**. When **`cdFiler`** read from a file, it started from the beginning of the file and read the contents, one byte at a time or in larger chunks, but from the beginning straight through until the end. This sequential approach works fine if you intend to read or write the entire file all at once. As you might have guessed, there is another model.

Instead of starting at the beginning and streaming through a file, you can use a technique called **random file access**. The Standard Library provides a set of functions that let you reposition the file position indicator to any location within the file, so that the next read or write you do occurs exactly where you want it to.

Imagine a file filled with 100 **`long`**s, each of which was 4 bytes long. The file would be 400 bytes long. Now suppose you wanted to retrieve the 10th **`long`** in the file. Using the sequential model, you would have to do 10 reads to get the 10th **`long`** into memory. Unless you read the entire file into memory, you'll constantly be reading a series of **`long`**s to get to the **`long`** you want.

Using the random access model, you would first calculate where in the file the 10th **`long`** starts, jump to that position in the file, then just read that **`long`**. To move the file position indicator just before the 10th **`long`**, you'd skip over the first 9 **`long`**s (9*4 = 36 bytes).

fseek(), ftell(), and rewind()

There are five functions that you'll need to know about in order to randomly access your files. **`fseek()`** moves the file position indicator to an offset you specify, relative to either the beginning of the file, the current file position, or the end of the file:

```
int  fseek( FILE *fp, long offset, int
 wherefrom );
```

You'll pass your **`FILE`** pointer as the first parameter, a **`long`** offset as the second parameter, and one of **`SEEK_SET`**, **`SEEK_CUR`**, or **`SEEK_END`** as the third parameter. **`SEEK_SET`** represents the beginning of the file, **`SEEK_CUR`** represents the current position, and **`SEEK_END`** represents the end of the file (in which case you'll probably use a negative **`offset`**).

`ftell()` takes a **`FILE`** pointer as a parameter and returns a **`long`** containing the value of the file position indicator:

```
long ftell( FILE *fp );
```

`rewind()` takes a **`FILE`** pointer as a parameter and resets the file position indicator to the beginning of the file:

```
void rewind( FILE *fp );
```

The functions **`fsetpos()`** and **`fgetpos()`** were introduced as part of ISO C and allow you to work with file offsets that are larger than will fit in a **`long`**. You can look these two functions up in the usual places.

dinoEdit.μ

The last sample program in this chapter, **`dinoEdit`**, is a simple example of random file access. It allows you to edit a series of dinosaur names stored in a file named *My Dinos*. Each dinosaur name in *My Dinos* is 20 characters long. If the actual dinosaur name is shorter than 20 characters, the appropriate number of spaces is added to the name to bring the length up to 20. This is done to make the size of each item in the file a fixed length. You'll see why this is important as we go through the source code. For now, let's take **`dinoEdit`** for a spin.

Open the *Learn C Projects* folder, go inside the folder *10.03 - dinoEdit*, and open and run *dinoEdit.xcode*. **`dinoEdit`** will count the number of dinosaur names in the file *My Dinos* and will use that number to prompt you for a dinosaur number to edit:

```
Enter number from 1 to 5 (0 to exit):
```

Since the file *My Dinos* has 5 dinosaurs, enter a number from 1 to 5:

```
Enter number from 1 to 5 (0 to exit): 3
```

If you enter the number 3, for example, **`dinoEdit`** will fetch the third dinosaur name from the file, then ask you to enter a new name for the third dinosaur. When you type a new name, **`dinoEdit`** will overwrite the existing name with the new name.

```
Dino #3: Galimimus
Enter new name: Euoplocephalus
```

Either way, **`dinoEdit`** will prompt you to enter another dinosaur number. Reenter the same number, so you can verify that the change was made in the file.

```
Enter number from 1 to 5 (0 to exit): 3
Dino #3: Euoplocephalus
Enter new name: Galimimus
Enter number from 1 to 5 (0 to exit): 0
Goodbye...
```

Let's take a look at the source code...

Stepping Through the Source Code

The file *dinoEdit.h* starts off with a few **`#define`**s. **`kDinoRecordSize`** defines the length of each dinosaur record. Note that the dinosaur file doesn't contain any carriage returns, just 5 * 20 = 100 bytes of pure dinosaur pleasure!

kMaxLineLength defines the length of an array of **char**s we'll use to read in any new dinosaur names. **kDinoFileName** is the name of the dinosaur file.

```
/***********/
/* Defines */
/***********/
#define kDinoRecordSize         20
#define kMaxLineLength          100
#define kDinoFileName           "../My Dinos"
```

Next come the function prototypes for the functions in *main.c*.

```
/********************************/
/* Function Prototypes - main.c */
/********************************/
int  GetNumber( void );
int  GetNumberOfDinos( void );
void ReadDinoName( int number, char
            *dinoName );
char GetNewDinoName( char *dinoName );
void WriteDinoName( int number, char
            *dinoName );
void Flush( void );
void DoError( char *message );
```

main.c starts with four **#include**s. **<stdlib.h>** gives us access to the function **exit()**. **<stdio.h>** gives us access to a number of functions, including **printf()** and all the file manipulation functions, types and constants. **<string.h>** gives us access to the function **strlen()**. You've already seen what **"dinoEdit.h"** brings to the table.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "dinoEdit.h"
```

If you ever want to find out which of the functions you call are dependent on which of your include files, just comment out the **#include** statement in question and recompile. The compiler will spew out an error message (or a whole bunch of messages) telling you it couldn't find a prototype for a function you called.

main() basically consists of a loop that first prompts for a dinosaur number at the top of the loop, then processes the selection in the body of the loop.

```
/********************************> main <*/
int  main( void )
{
 int number;
 FILE      *fp;
 char      dinoName[ kDinoRecordSize+1 ];
```

`GetNumber()` prompts for a dinosaur number between 0 and the number of dinosaur records in the file. If the user types 0, we'll drop out of the loop and exit the program.

```
   while ( (number = GetNumber() ) != 0 )
   {
```

If we made it here, **`GetNumber()`** must have returned a legitimate record number. **`ReadDinoName()`** takes the dinosaur number and returns the corresponding dinosaur name from the file. The returned dinosaur name is then printed.

```
      ReadDinoName( number, dinoName );

      printf( "Dino #%d: %s\n", number,
                     dinoName );
```

`GetNewDinoName()` prompts the user for a new dinosaur name to replace the existing name. **`GetNewDinoName()`** returns **`true`** if a name is entered and **`false`** if the user just hit a return. If the user entered a name, we'll pass it on to **`WriteDinoName()`**, which will write the name in the file, overwriting the old name.

```
      if ( GetNewDinoName( dinoName ) )
            WriteDinoName( number, dinoName );
   }
```

```
   printf( "Goodbye..." );

   return 0;
 }
```

`GetNumber()` starts off with a call to **`GetNumberOfDinos()`**. As its name implies, **`GetNumberOfDinos()`** goes into the dinosaur file and returns the number of records in the file.

```
/***************************> GetNumber <*/
int  GetNumber( void )
{
 int       number, numDinos;

 numDinos = GetNumberOfDinos();
```

`GetNumber()` then continuously prompts for a dinosaur number until the user enters a number between 0 and **`numDinos`**.

```
 do
 {
    printf( "Enter number from 1 to %d (0 to
          exit): ", numDinos );
    scanf( "%d", &number );
    Flush();
 }
 while ( (number < 0) || (number > numDinos));

 return( number );
}
```

`GetNumberOfDinos()` starts our file management adventure. First, we'll open *My Dinos* for reading only.

```
/********************> GetNumberOfDinos <*/
int  GetNumberOfDinos( void )
{
 FILE      *fp;
 long      fileLength;

 if ( (fp = fopen( kDinoFileName, "r" )) ==
           NULL )
    DoError("Couldn't open file...Goodbye!");
```

Notice that we've passed an error message to a function called **`DoError()`** instead of printing it with **`printf()`**. There are several reasons for doing this. First, since **`DoError()`** executes two lines of code (calls of **`printf()`** and **`exit()`**), each **`DoError()`** call saves a bit of code.

More importantly, this approach encapsulates all our error handling in a single function. If we want to send all error messages to a log file, all we have to do is edit **`DoError()`** instead of hunting down all the error messages and attaching a few extra lines of code.

Next, we'll call **`fseek()`** to move the file position indicator to the end of the file. Can you see what's coming?

```
 if ( fseek( fp, 0L, SEEK_END ) != 0 )
    DoError( "Couldn't seek to end of file...
             Goodbye!" );
```

Now we'll call **`ftell()`** to retrieve the current file position indicator, which also happens to be the file length! Cool!

```
 if ( (fileLength = ftell( fp )) == -1L )
    DoError( "ftell() failed...Goodbye!" );
```

Now that we have the file length, we can close the file.

```
 fclose( fp );
```

Finally, we'll calculate the number of dinosaur records by dividing the file length by the number of bytes in a single record. For simplicities' sake, we'll convert the number of records to an **`int`** before we return it. That means that we can't deal with a file that contains more than 32,767 dinosaur records. How many dinosaurs can you name?

```
 return( (int)(fileLength / kDinoRecordSize) );
}
```

`ReadDinoName()` first opens the file for reading only.

```
/***********************> ReadDinoName <*/
void ReadDinoName( int number, char
                  *dinoName )
{
 FILE      *fp;
 long      bytesToSkip;

 if ( (fp = fopen( kDinoFileName, "r" )) ==
                NULL )
    DoError("Couldn't open file...Goodbye!");
```

Since we'll be reading the **`number`**th dinosaur, we have to move the file position indicator to the end of the (**`number-1`**)th dinosaur. That means we'll need to skip over (**`number-1`**) dinosaur records.

```
 bytesToSkip = (long)((number-1) *
          kDinoRecordSize);
```

We'll use **`fseek()`** to skip that many bytes from the beginning of the file (that's what the constant **`SEEK_SET`** is for).

```
 if ( fseek( fp, bytesToSkip, SEEK_SET )
                != 0 )
    DoError( "Couldn't seek in file...
                Goodbye!" );
```

Finally, we'll call **`fread()`** to read the dinosaur record into the array of **`char`**s pointed to by **`dinoName`**. The first **`fread()`** parameter is the pointer to the block of memory where the data will be read. The second parameter is the number of bytes in a single record. **`fread()`** expects both the second and third parameters to be of type **`size_t`**, so we'll use a typecast to make the compiler happy. Gee, by the time we talk about typecasting in Chapter 11, you'll already be an expert! The third parameter is the number of records to read in. We want to read in 1 record of **`kDinoRecordSize`** bytes. The last parameter is the **`FILE`** pointer we got from **`fopen()`**.

`fread()` returns the number of records read. Since we asked **`fread()`** to read 1 record, we expect **`fread()`** to return a value of 1. If that doesn't happen, something is dreadfully wrong (perhaps the file got corrupted, or that Pepsi you spilled in your hard drive is finally starting to take effect).

```
 if ( fread( dinoName,
          (size_t)kDinoRecordSize,
          (size_t)1, fp ) != 1 )
    DoError( "Bad fread()...Goodbye!" );
```

Once again, we close the file when we're done working with it.

```
  fclose( fp );
}
```

`GetNewDinoName()` starts by prompting for a new dinosaur name, then calling **`fgets()`** to read in a line of text. We'll use our **`strlen()`** trick to replace the **`'\n'`** with a **`'\0'`**.

```
/**********************> GetNewDinoName <*/
char GetNewDinoName( char *dinoName )
{
  char      line[ kMaxLineLength ];
  int       i, nameLen;
  char      *result;

  printf( "Enter new name: " );

  result = fgets( line, kMaxLineLength,
                  stdin );
  line[ strlen( line ) - 1 ] = '\0';
```

Our next step is to fill the **`dinoName`** array with spaces. We'll then call **`strlen()`** to find out how many characters the user typed in. We'll copy those characters back into the **`dinoName`** array, leaving **`dinoName`** with a dinosaur name followed by a bunch of spaces.

```
  for ( i=0; i<kDinoRecordSize; i++ )
    dinoName[i] = ' ';
```

`strlen()` takes a pointer to a **`0`** terminated string and returns the length of the string, not including the **`0`** terminator.

```
  nameLen = strlen( line );
```

If the user typed a dinosaur name larger than 20 characters long, we'll only copy the first 20 characters.

```
  if ( nameLen > kDinoRecordSize )
    nameLen = kDinoRecordSize;
```

Here's where we copy the characters from **`line`** into **`dinoName`**.

```
  for ( i=0; i<nameLen; i++ )
    dinoName[i] = line[i];
```

Finally, we'll return **`true`** to let the calling function know that the name is ready.

```
  return true;
}
```

`WriteDinoName()` opens the file for reading and writing. Since we used a mode of **`"r+"`** instead of **`"w+"`**, we won't lose the contents of *My Dinos* (in other words, *My Dinos* won't be deleted and recreated).

```
/************************> WriteDinoName <*/
void WriteDinoName( int number, char
                   *dinoName )
{
 FILE      *fp;
 long      bytesToSkip;

 if ( (fp = fopen( kDinoFileName, "r+" )) ==
                  NULL )
    DoError( "Couldn't open file...Goodbye!"
 );
```

Next, we calculate the number of bytes we need to skip to place the file position indicator at the beginning of the record we want to overwrite, then call **`fseek()`** to move the file position indicator.

```
 bytesToSkip = (long)((number-1) *
                  kDinoRecordSize);

 if ( fseek( fp, bytesToSkip, SEEK_SET )
                  != 0 )
    DoError( "Couldn't seek in file...
                  Goodbye!" );
```

We then call **`fwrite()`** to write the dinosaur record back out. **`fwrite()`** works exactly the same way as **`fread()`**, including returning the number of records written.

```
 if ( fwrite( dinoName,
            (size_t)kDinoRecordSize,
            (size_t)1, fp ) != 1 )
    DoError( "Bad fwrite()...Goodbye!" );

 fclose( fp );
}
```

You've seen this function before...

```
/*******************************> Flush <*/
void Flush( void )
{
 while ( getchar() != '\n' )
    ;
}
```

`DoError()` prints the error message, adding a carriage return, then exits.

```
/*****************************> DoError <*/
void DoError( char *message )
{
 printf( "%s\n", message );
 exit( 0 );
}
```

What's Next?

Chapter 11 tackles a wide assortment of programming topics. We'll look at typecasting, the technique used to translate from one type to another. We'll cover recursion, the ability of a function to call itself. We'll also examine function pointers, variables that can be used to pass a function as a parameter.

Exercises

1) What's wrong with each of the following code fragments:

a)

```
FILE      *fp;

fp = fopen( "w", "My Data File" );
if ( fp != NULL )
   printf( "The file is open." );
```

b)

```
char      myData = 7;
FILE      *fp;

fp = fopen( "r", "My Data File" );
fscanf( "Here's a number: %d", &myData );
```

c)

```
FILE      *fp;
char      *line;

fp = fopen( "My Data File", "r" );
fscanf( fp, "%s", &line );
```

d)

```
FILE      *fp;
char      line[100];

fp = fopen( "My Data File", "w" );
fscanf( fp, "%s", line );
```

2) Write a program that reads in and prints a file with the following format:

- The first line in the file contains a single **`int`**. Call it x.
- All subsequent lines contain a list of x **`int`**s separated by tabs.

For example, if the first number in the file is 6, all subsequent lines will have 6 **`int`**s per line. There is no limit to the number of lines in the file. Keep reading and printing lines until you hit the end of the file.

You can print each **`int`** as you encounter it or, for extra credit, allocate an array of **`int`**s large enough to hold one line's worth of **`int`**s, then pass that array to a function that prints an **`int`** array.

3) Modify **`cdFiler`** so memory for the **`artist`** and **`title`** lines is allocated as the lines are read in. First, you'll need to change the **`CDInfo struct`** declaration as follows:

```
struct CDInfo
{
   char                  rating;
   char                  *artist
   char                  *title;
   struct CDInfo *next;
};
```

Not only will you call **`malloc()`** to allocate a **`CDInfo struct`**, you'll also call **`malloc()`** to allocate space for the **`artist`** and **`title`** strings. Don't forget to leave enough space for the terminating **`0`** at the end of each string.

Chapter 11 Advanced Topics

Congratulations! By now you've mastered most of the fundamental C programming concepts. This chapter will fill you in on some useful C programming tips, tricks, and techniques that will enhance your programming skills. We'll start with a look at typecasting, C's mechanism for translating one data type to another.

What is Typecasting?

There often will be times when you find yourself trying to convert a variable of one type to a variable of another type. For example, this code fragment:

```
float f;
int  i;

f = 3.5;
i = f;

printf( "i is equal to %d", i );
```

causes this line:

```
i is equal to 3
```

to appear in the console window. Notice that the original value assigned to **`f`** was truncated from 3.5 to 3 when the value in **`f`** was assigned to **`i`**. This truncation was caused when the compiler saw an **`int`** on the left side and a **`float`** on the right side of this assignment statement:

```
i = f;
```

The compiler automatically translated the **`float`** to an **`int`**. In general, the right side of an assignment statement is always translated to the type on the left side when the assignment occurs. In this case, the compiler handled the type conversion for you.

Typecasting is a mechanism you can use to translate the value of an expression from one type to another. A **typecast**, or just plain **cast**, always takes this form:

```
(type) expression
```

where **`type`** is any legal C type. In this code fragment:

```
float f;

f = 1.5;
```

the variable **`f`** gets assigned a value of 1.5. In this code fragment:

```
float f;

f = (int)1.5;
```

the value of 1.5 is cast as an **`int`** before being assigned to **`f`**. Just as you might imagine, casting a **`float`** as an **`int`** truncates the **`float`**, turning the value 1.5 into 1. In this example, two casts were performed. First, the **`float`** value 1.5 was cast to the **`int`** value 1. When this **`int`** value was assigned to the **`float f`**, the value was cast to the **`float`** value 1.0.

Cast With Care

Use caution when you cast from one type to another. Problems can arise when casting between types of a different size. Consider this example:

```
int  i;
char c;

i = 500;
c = i;
```

Here, the value 500 is assigned to the **`int i`**. So far, so good. Next, the value in **`i`** is cast to a **`char`** as it is assigned to the **`char c`**. See the problem? Since a **`char`** can only hold values between -128 and 127, assigning a value of 500 to **`c`** doesn't make sense.

So what happens to the extra byte or bytes when a larger type is cast to a smaller type? The matching bytes are typecast and the value of any extra bytes is lost.

For example, when a 2 byte **`int`** is cast to a 1 byte **`char`**, the leftmost byte of the **`int`** (the byte with the more significant bits, the bits valued 2^8 through 2^{15}) is dropped and the rightmost byte (the bits valued 2^0 through 2^7) is copied into the **`char`**.

In this case:

```
int  i;

char c;

i = 500;

c = i;
```

the **`int i`** has a value of **`0x01E4`**, which is hex for 500. After the second assignment, the **`char`** ends up with the value **`0xE4`**, which has a value of 244 if the **`char`** was **`unsigned`** or -12 if the **`char`** is **`signed`**.

Casting With Pointers

Typecasting can also be used when working with pointers. This notation:

```
(int *) myPtr
```

casts the variable **`myPtr`** as a pointer to an **`int`**.

Casting with pointers allows you to link together **`struct`**s of different types. For example, suppose you declared two **`struct`** types, as follows:

```
struct Dog
{
 struct Dog       *next;
} ;

struct Cat
{
 struct Cat *next;
} ;
```

By using typecasting, you could create a linked list that contains both **`Cat`**s and **`Dog`**s. Figure 11.1 shows a **`Dog`** whose **`next`** field points to a **`Cat`**. Imagine the source code you'd need to implement such a linked list.

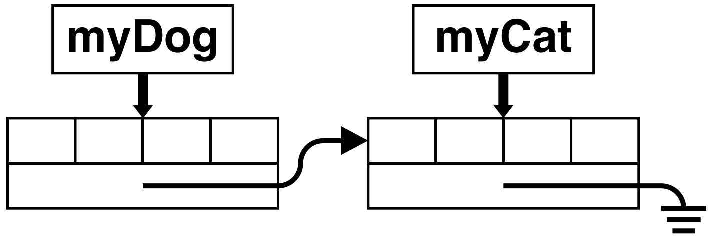


Figure 11.1 **`myDog.next`** *points to* **`myCat`**. **`myCat.next`** *points to* **`NULL`**.

Consider this source code:

```
struct Dog myDog;
struct Cat myCat;

myDog.next = &myCat; /* <--Compiler complains
 */
myCat.next = NULL;
```

In the first assignment statement, a pointer of one type is assigned to a pointer of another type. **`&myCat`** is a pointer to a **`struct`** of type **`Cat`**. **`myDog.next`** is declared to be a pointer to a **`struct`** of type **`Dog`**. To make this code compile, we'll need a typecast:

```
struct Dog myDog;
struct Cat myCat;

myDog.next = (struct Dog *)(&myCat);
myCat.next = NULL;
```

If both sides of an assignment operator are arithmetic types (like **`float`**, **`int`**, **`char`**, etc.), the compiler will automatically cast the right side of the assignment to the type of the left side. If both sides are pointers, you'll have to perform the typecast yourself.

There are a few exceptions to this rule. If the pointers on both sides of the assignment are the same type, no typecast is necessary. If the pointer on the right side is either **`NULL`** or of type **`(void *)`**, no typecast is necessary. Finally, if the pointer on the left side is of type **`(void *)`**, no typecast is necessary.

The type **`(void *)`** is sort of a wild card for pointers. It matches up with any pointer type. For example, here's a new version of the **`Dog`** and **`Cat`** code:

```
struct Dog
{
 void      *next;
} ;

struct Cat
{
 void      *next;
} ;

struct Dog myDog;
struct Cat myCat;

myDog.next = &myCat;
myCat.next = NULL;
```

This code lets **`Dog.next`** point to a **`Cat struct`** without a typecast. If you are not sure what type your pointers will be pointing to, declare your pointers as **`(void *)`**.

Unions

C offers a special data type, known as a **union**, which allows a single variable to disguise itself as several different data types. **union**s are declared just like **struct**s. Here's an example:

```
union Number
{
 int  i;
 float f;
 char *s;
}   myNumber;
```

This declaration creates a **union** type named **Number**. It also creates an individual **Number** named **myNumber**. If this were a **struct** declaration, you'd be able to store three different values in the three fields of the **struct**. A **union**, on the other hand, lets you store one and only one of the **union**'s fields in the **union**. Here's how this works.

When a **union** is declared, the compiler allocates the space required by the largest of the **union**'s fields, sharing that space with all of the **union**'s fields. If an **int** requires 2 bytes, a **float** 4 bytes, and a pointer 4 bytes, **myNumber** is allocated exactly 4 bytes. You can store an **int**, a **float**, or a **char** pointer in **myNumber**. The compiler allows you to treat **myNumber** as any of these types. To refer to **myNumber** as an **int**, refer to:

```
myNumber.i
```

To refer to **myNumber** as a **float**, refer to:

```
myNumber.f
```

To refer to **myNumber** as a **char** pointer, refer to:

```
myNumber.s
```

You are responsible for remembering which form the **union** is currently occupying.

If you store an **int** in **myUnion** by assigning a value to **myUnion.i**, you'd best remember that fact. If you proceed to store a **float** in **myUnion.f**, you've just trashed your **int**. Remember, there are only 4 bytes allocated to the entire **union**.

In addition, storing a value as one type then reading it as another can produce unpredictable results. For example, if you stored a **float** in **myNumber.f**, the field **myNumber.i** would *not* be the same as **(int)(myNumber.f)**.

One way to keep track of the current state of the **union** is to declare an **int** to go along with the **union**, as well as a **#define** for each of the union's fields:

```
#define kUnionContainsInt     1
#define kUnionContainsFloat   2
#define kUnionContainsPointer 3

union Number
{
 int  i;
 float f;
 char *s;
} myNumber;

int  myUnionTag;
```

If you are currently using **`myUnion`** as a **`float`**, assign the value **`kUnionContainsFloat`** to **`myUnionTag`**. Later in your code you can use **`myUnionTag`** when deciding which form of the **`union`** you are dealing with:

```
if ( myUnionTag == kUnionContainsInt )
 DoIntStuff( myUnion.i );
else if ( myUnionTag == kUnionContainsFloat )
 DoFloatStuff( myUnion.f );
else
 DoPointerStuff( myUnion.s );
```

Why Use Unions?

In general, **`union`**s are most useful when dealing with two data structures that share a set of common fields, but differ in some small way. For example, consider these two **`struct`** declarations:

```
struct Pitcher
{
 char      name[ 40 ];
 int       team;
 int       strikeouts;
 int       runsAllowed;
} ;

struct Batter
{
 char      name[ 40 ];
 int       team;
 int       runsScored;
 int       homeRuns;
} ;
```

These **`struct`**s might be useful if you were tracking the pitchers and batters on your favorite baseball team. Both **`struct`**s share a set of common fields, the array of **`char`**s named **`name`** and the **`int`** named **`team`**. Both **`struct`**s have their own unique fields as well. The **`Pitcher struct`** contains a pair of fields appropriate for a pitcher, **`strikeouts`** and **`runsAllowed`**. The **`Batter struct`** contains a pair of fields appropriate for a batter, **`runsScored`** and **`homeRuns`**.

One solution to your baseball-tracking program would be to maintain two types of **`struct`**s, a **`Pitcher`** and a **`Batter`**. There is nothing wrong with this approach. There is an alternative, however. You can declare a single **`struct`** that contains the fields common to **`Pitcher`** and **`Batter`**, with a

`union` for the unique fields:

```
#define kMets  1
#define kReds  2

#define kPitcher  1
#define kBatter   2

struct Pitcher
{
 int   strikeouts;
 int   runsAllowed;
} ;

struct Batter
{
 int   runsScored;
 int   homeRuns;
} ;

struct Player
{
 int   type;
 char  name[ 40 ];
 int   team;
 union
 {
    struct Pitcher   pStats;
    struct Batter bStats;
 } u;
};
```

Here's an example of a **`Player`** declaration:

```
struct Player   myPlayer;
```

Once you created the **`Player struct`**, you would initialize the **`type`** field with one of either **`kPitcher`** or **`kBatter`**:

```
myPlayer.type = kBatter;
```

You would access the name and team fields like this:

```
myPlayer.team = kMets;
printf( "Stepping up to the plate:  %s",
 myPlayer.name );
```

Finally, you'd access the **`union`** fields like this:

```
if ( myPlayer.type == kPitcher )
 myPlayer.u.pStats.strikeouts = 20;
```

The **`u`** was the name given to the union in the declaration of the **`Player`** type. Every **`Player`** you declare will automatically have a **`union`** named **`u`** built into it. The **`union`** gives you access to either a **`Pitcher struct`** named **`pStats`** or a **`Batter struct`** named **`bStats`**. The example above references the **`strikeouts`** field of the **`pStats`** field.

`union`s provide an interesting alternative to maintaining multiple data structures. Try them. Write your next program using a **`union`** or two. If you don't like them, you can return them for a full refund.

Function Recursion

Some programming problems are best solved by repeating a mathematical process. For example, to learn whether a number is prime (see Chapter 6) you might step through each of the even integers between 2 and the number's square root, one at a time, searching for a factor. If no factor is found, you have a prime. The process of stepping through the numbers between 2 and the number's square root is called **iteration**.

In programming, iterative solutions are fairly common. Almost every time you use a **`for`** loop, you are applying an iterative approach to a problem. An alternative to the iterative approach is known as **recursion**. In a recursive approach, instead of repeating a process in a loop, you embed the process in a function and have the function call itself until the process is complete. The key to recursion is a function calling itself.

Suppose you wanted to calculate 5 factorial (also known as **`5!`**). The factorial of a number is the product of each integer from 1 up to the number. For example, 5 factorial is:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Using an iterative approach, you might write some code like this:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
 int   i, num;
 long  fac;

 num = 5;
 fac = 1;

 for ( i=1; i<=num; i++ )
    fac *= i;

 printf( "%d factorial is %ld.", num, fac );

 return 0;
}
```

If you are interested in trying this code, you'll find it in the *Learn C Projects* folder, under the subfolder named *11.01 - iterate*.

If you ran this program, you'd see this line printed in the console window:

```
5 factorial is 120.
```

As you can see from the source code, the algorithm steps through (iterates) the numbers 1 through 5, building the factorial with each successive multiplication.

A Recursive Approach

You can use a recursive approach to solve the same problem. For starters, you'll need a function to act as a base for the recursion, a function that will call itself. There are two things you'll need to build into your recursive function. First, you'll need a mechanism to keep track of the depth of the recursion. In other words, you'll need a variable or parameter that changes, depending on the number of times the recursive function calls itself.

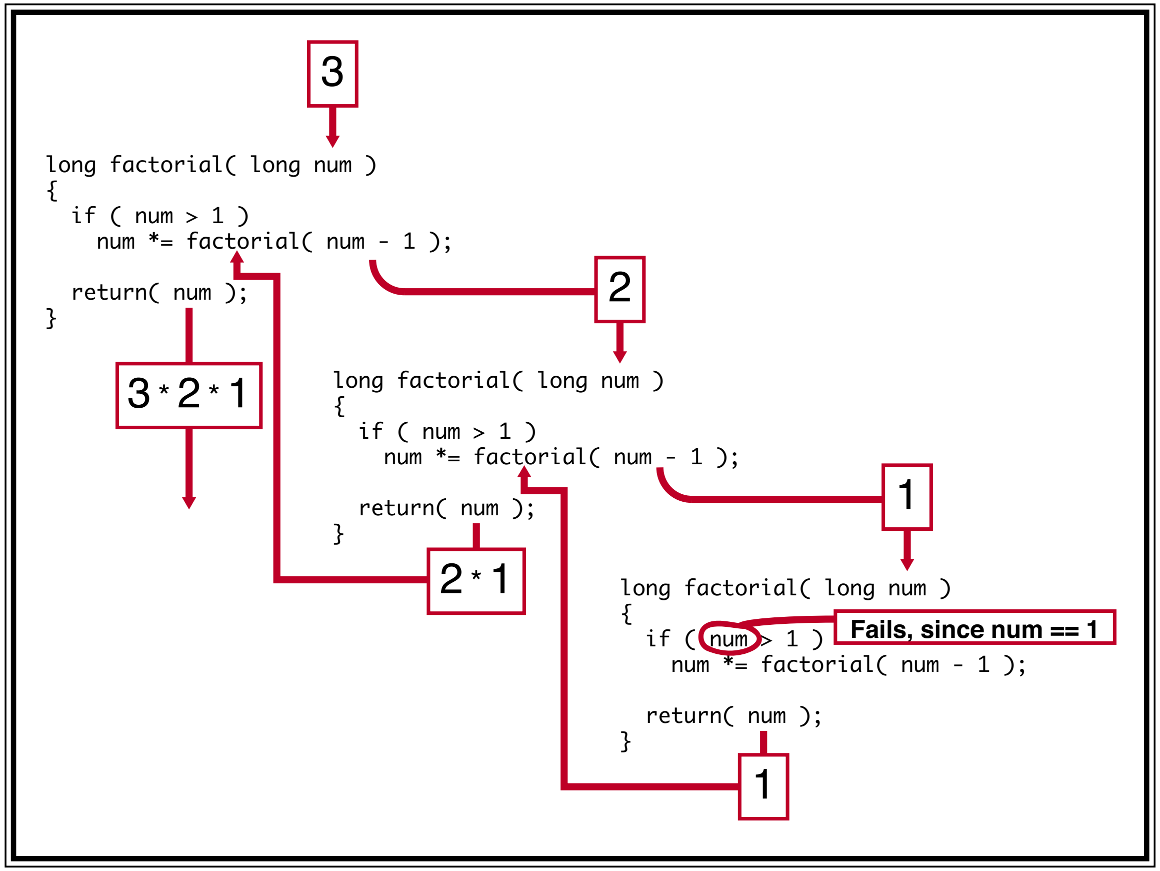
Second, you'll need a terminating condition, something that tells the recursive function when it's gone deep enough. Here's one version of a recursive function that calculates a factorial:

```
long factorial( long num )
{
 if ( num > 1 )
    num *= factorial( num - 1 );

 return( num );
}
```

`factorial()` takes a single parameter, the number whose factorial you are trying to calculate. **`factorial()`** first checks to see whether the number passed to it is greater than 1. If not, **`factorial()`** calls itself, passing 1 less than the number passed into it. This strategy guarantees that, eventually, **`factorial()`** will get called with a value of 1.

Figure 11.2 shows this process in action. The process starts with a call to **`factorial()`**:

```
result = factorial( 3 );
```



```
long factorial( long num )
{
  if ( num > 1 )
    num *= factorial( num - 1 );

  return( num );
}
```

Figure 11.2 The recursion process caused by the call **`factorial(3)`**.

Take a look at the leftmost **`factorial()`** source code in Figure 11.2. **`factorial()`** is called with a parameter of 3. The **`if`** statement checks to see if the parameter is greater than 1. Since 3 is greater than 1, the statement:

```
num *= factorial( num - 1 );
```

is executed. This statement calls **`factorial()`** again, passing a value of **`n-1`**, or 2, as the parameter. This second call of **`factorial()`** is pictured in the center of Figure 11.2.

It's important to understand that this second call to **`factorial()`** is treated just like any other function call that occurs in the middle of a function. The calling function's variables are preserved while the called function runs. In this case, the called function is just another copy of **`factorial()`**.

This second call of **`factorial()`** takes a value of 2 as a parameter. The **`if`** statement compares this value to 1 and, since 2 is greater than 1, executes the statement:

```
num *= factorial( num - 1 );
```

This statement calls **`factorial()`** yet again, passing **`num-1`**, or 1, as a parameter. The third call of **`factorial()`** is portrayed on the rightmost side of Figure 11.2.

The third call of **`factorial()`** starts with an **`if`** statement. Since the input parameter was 1, the **`if`** statement fails. Thus, the recursion termination condition is reached. Now, this third call of **`factorial()`** returns a value of 1.

At this point, the second call of **`factorial()`** resumes, completing the statement:

```
num *= factorial( num - 1 );
```

Since the call of **`factorial()`** returned a value of 1, this statement is equivalent to:

```
num *= 1;
```

leaving **`num`** with the same value it came in with, namely 2. This second call of **`factorial()`** returns a value of 2.

At this point, the first call of **`factorial()`** resumes, completing the statement:

```
num *= factorial( num - 1 );
```

Since the second call of **`factorial()`** returned a value of 2, this statement is equivalent to:

```
num *= 2;
```

Since the first call of **`factorial()`** started with the parameter **`num`** taking a value of 3, this statement sets **`num`** to a value of 6. Finally, the original call of **`factorial()`** returns a value of 6. This is as it should be, since 3 factorial = 3 * 2 * 1 = 6.

The recursive version of the factorial program is in the *Learn C Projects* folder, under the subfolder named *11.02 - recurse*. Open the project and follow the program through, line by line.

Binary Trees

As you learn more about data structures, you'll discover new applications for recursion. For example, one of the most-used data structures in computer programming is the **binary tree** (Figure 11.3). As you'll see later, binary trees were just made for recursion. The binary tree is similar to the linked list. Both consist of **`struct`**s connected by pointers embedded in each **`struct`**.

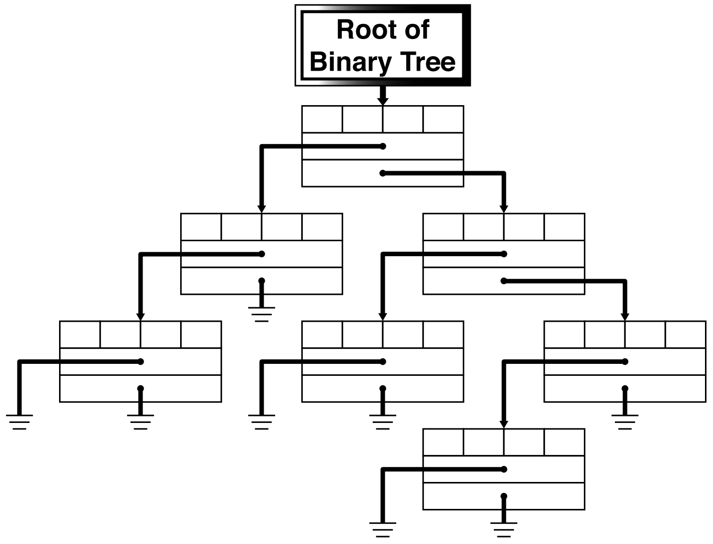


Figure 11.3 *A binary tree. Why binary? Each node in the tree contains two pointers.*

Linked lists are linear. Each **`struct`** in the list is linked by pointers to the **`struct`** behind it and in front of it in the list. Binary trees always start with a single **`struct`**, known as the root **`struct`** or **root node**. Where the linked-list **`struct`**s we've been working with contain a single pointer, named **`next`**, binary-tree **`struct`**s each have two pointers, usually known as **`left`** and **`right`**.

Check out the binary tree in Figure 11.3. Notice that the root node has a left **child** and a right child. The left child has its own left child but its **`right`** pointer is set to **`NULL`**. The left child's left child has two **`NULL`** pointers. A node with two **`NULL`** pointers is known as a **leaf node** or **terminal node**.

Binary trees are extremely useful. They work especially well when the data you are trying to sort has a **comparative relationship**. This means that if you compare one piece of data to another, you'll be able to judge the first piece as greater than, equal to, or less than the second piece. For example, numbers are comparative. Words in a dictionary can be comparative, if you consider their alphabetical order. The word *iguana* is greater than *aardvark*, but less than *xenophobe*.

Here's how you might store a sequence of words, one at a time, in a binary tree. We'll start with this list of words:

```
opulent
entropy
salubrious
ratchet
coulomb
yokel
tortuous
```

Figure 11.4 shows the word **`opulent`** added to the root node of the binary tree. Since it is the only word in the tree so far, both the left and right pointers are set to **`NULL`**.

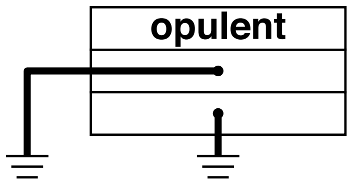


Figure 11.4 *The word `opulent` is entered into the binary tree.*

Figure 11.5 shows the word **`entropy`** added to the binary tree. Since **`entropy`** is less than **`opulent`** (i.e., comes before it alphabetically), **`entropy`** is stored as **`opulent`**'s left child.

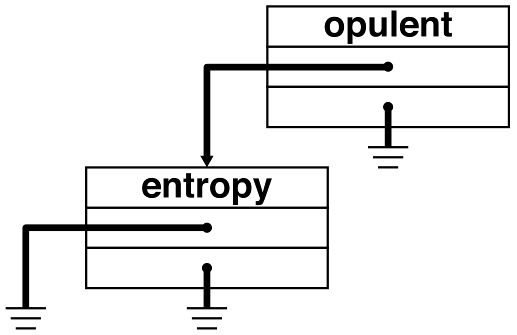


Figure 11.5 *The word* `entropy` *is less than the word* `opulent` *and is added as its left child in the binary tree.*

Next, Figure 11.6 shows the word **`salubrious`** added to the tree. Since **`salubrious`** is greater than **`opulent`**, it becomes **`opulent`**'s right child.

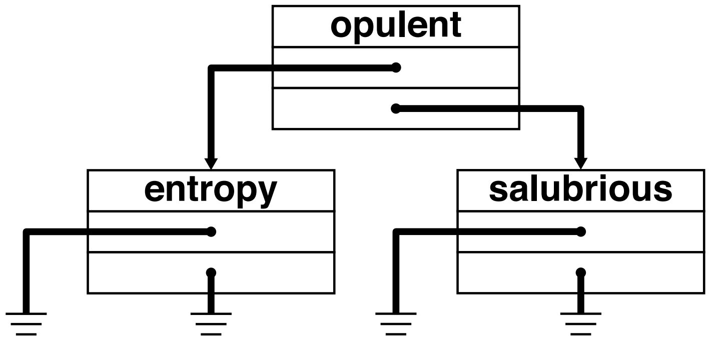


Figure 11.6 *The word* **`salubrious`** *is greater than the word* **`opulent`** *and is added to its right in the tree.*

Figure 11.7 shows the word **`ratchet`** added to the tree. First, **`ratchet`** is compared to **`opulent`**. Since **`ratchet`** is greater than **`opulent`** we follow the right pointer. Since there's a word there already, we'll have to compare **`ratchet`** to this word. Since **`ratchet`** is less than **`salubrious`**, we'll store it as **`salubrious`**'s left child.

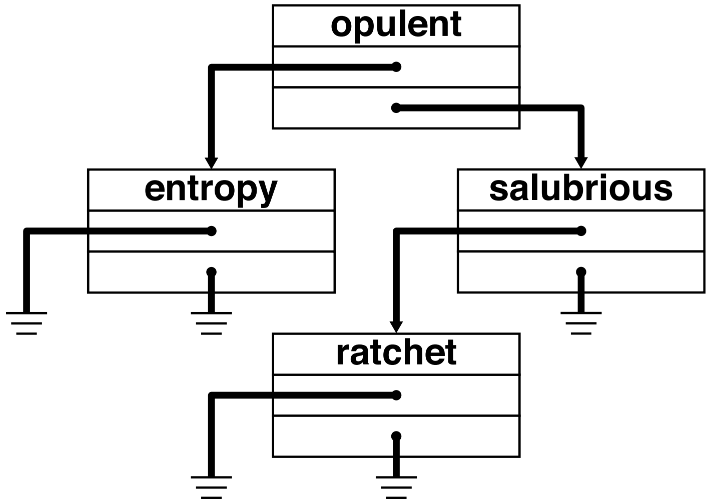


Figure 11.7 The word **`ratchet`** *is greater than* **`opulent`** *but less than* **`salubrious`** *and is placed in the tree accordingly.*

Figure 11.8 shows the binary tree after the remainder of the word list has been added. Do you understand how this scheme works? What would the binary tree look like if **`coulomb`** was the first word on the list? The tree would have no left children and would lean heavily to the right. What if **`yokel`** was the first word entered? As you can see, this particular use of binary trees depends on the order of the data. Randomized data that starts with a value close to the average produces a **balanced tree**. If the words had been entered in alphabetical order, you would have ended up with a binary tree that looked like a linked list.

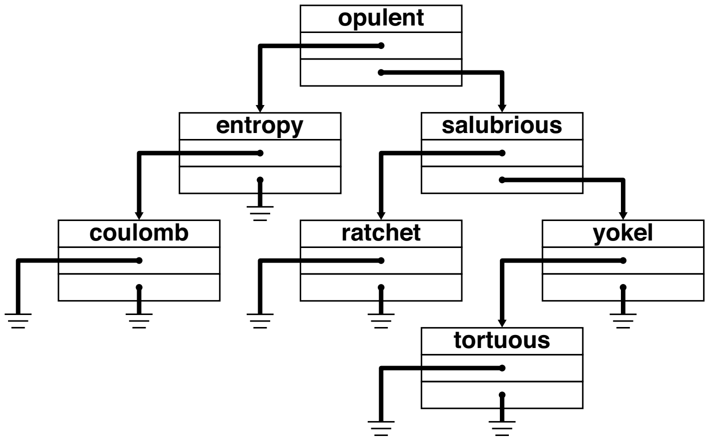


***Figure 11.8** The words `coulomb`, `yokel`, and `tortuous` are added to the tree.*

Searching Binary Trees

Now that your word list is stored in the binary tree, the next step is to look up a word in the tree. This is known as **searching** the tree. Suppose you wanted to look up the word **`tortuous`** in your tree. You'd start with the root node, comparing **`tortuous`** with **`opulent`**. Since **`tortuous`** is greater than **`opulent`**, you'd follow the right pointer to **`salubrious`**. You'd follow this algorithm down to **`yokel`** and finally **`tortuous`**.

Searching a binary tree is typically much faster than searching a linked list. In a linked list, you search through your list of nodes, one at a time, until you find the node you are looking for. On average, you'll end up searching half of the list. In a list of 100 nodes, you'll end up checking 50 nodes on average. In a list of 1000 nodes, you'll end up checking 500 nodes on average.

In a balanced binary tree, you reduce the search space in half each time you check a node. Without getting into the mathematics (check Knuth's *The Art of Computer Programming, Volume 3* for more info), the maximum number of nodes searched is approximately **$\log_2 n$**, where **`n`** is the number of nodes in the tree. On average, you'll search **$\log_2 n/2$** nodes. In a list of 100 nodes, you'll end up searching 3.32 nodes on average. In a list of 1000 nodes, you'll end up checking about 5 nodes on average.

As you can see, a binary tree provides a significant performance advantage over a linked list.

A binary tree that contained just words may not be that interesting, but imagine that these words were names of great political leaders. Each **`struct`** might contain a leader's name, biographical information, perhaps a pointer to another data structure containing great speeches. The value, name, or word that determines the order of the tree is said to be the **key**.

You don't always search a tree based on the key. Sometimes, you'll want to step through every node in the tree. For example, suppose your tree contained the name and birth date of each of the presidents of the United States. Suppose also that the tree was built using each president's last name as a key. Now suppose you wanted to compose a list of all presidents born in July. In this case, searching the tree alphabetically won't do you any good. You'll have to search every node in the tree. This is where recursion comes in.

Recursion and Binary Trees

Binary trees and recursion were made for each other. To search a tree recursively, the recursing function has to visit the current node, as well as call itself with each of its two child nodes. The child nodes will do the same thing with themselves and their child nodes. Each part of the recursion stops when a terminal node is encountered.

Check out this piece of code:

```
struct Node
{
 int     value;
 struct Node    *left;
 struct Node    *right;
} myNode;


Searcher( struct Node *nodePtr )
{
 if ( nodePtr != NULL )
 {
    VisitNode( nodePtr );
    Searcher( nodePtr->left );
    Searcher( nodePtr->right );
 }
}
```

The function **`Searcher()`** takes a pointer to a tree node as its parameter. If the pointer is **`NULL`**, we must be at a terminal node and there's no need to recurse any deeper. If the pointer points to a **`Node`**, the function **`VisitNode()`** is called. **`VisitNode()`** performs whatever function you want performed for each node in the binary tree. In our current example, **`VisitNode()`** could check to see if the president associated with this node was born in July. If so, **`VisitNode()`** might print the president's name in the console window.

Once the node is visited, **`Searcher()`** calls itself twice, once passing a pointer to its left child and once passing a pointer to its right child. If this version of **`Searcher()`** were used to search the tree in Figure 11.8, the tree would be searched in the order described in Figure 11.9. This type of search is known as a **preorder search**, because the node is visited before the two recursive calls take place.

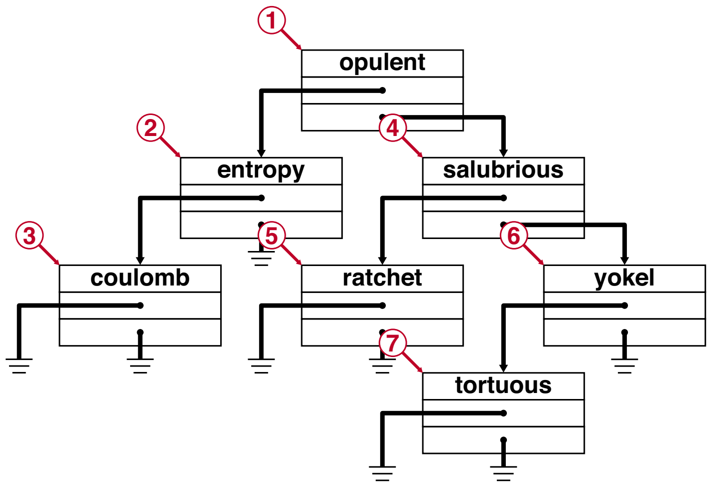


Figure 11.9 *A preorder search of a binary tree. This search was produced by the first version of* `Searcher()`*.*

Here's a slightly revised version of **`Searcher()`**. Without looking at Figure 11.10, can you predict the order that the tree will be searched? This version of **`Searcher()`** performs an **inorder search** of the tree:

```
Searcher( struct Node *nodePtr )
{
 if ( nodePtr != NULL )
 {
    Searcher( nodePtr->left );
    VisitNode( nodePtr );
    Searcher( nodePtr->right );
 }
}
```

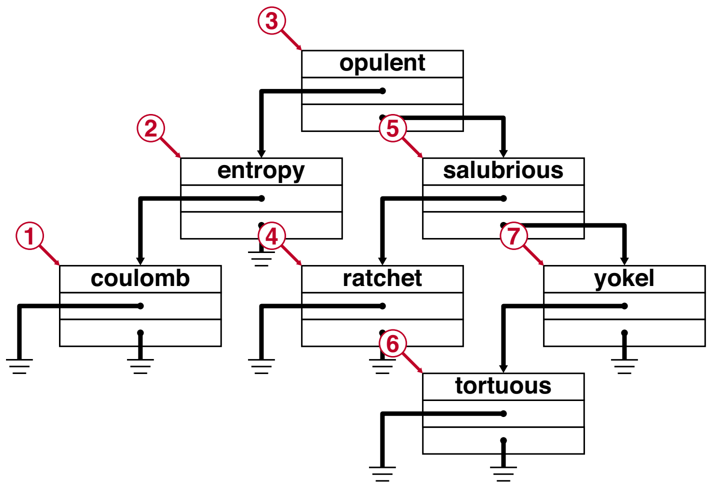


Figure 11.10 *An inorder search of the same tree.*

Here's a final look at **`Searcher()`**. This version performs a **postorder search** of the tree (Figure 11.11):

```
Searcher( struct Node *nodePtr )
{
 if ( nodePtr != NULL )
 {
    Searcher( nodePtr->left );
    Searcher( nodePtr->right );
    VisitNode( nodePtr );
 }
}
```

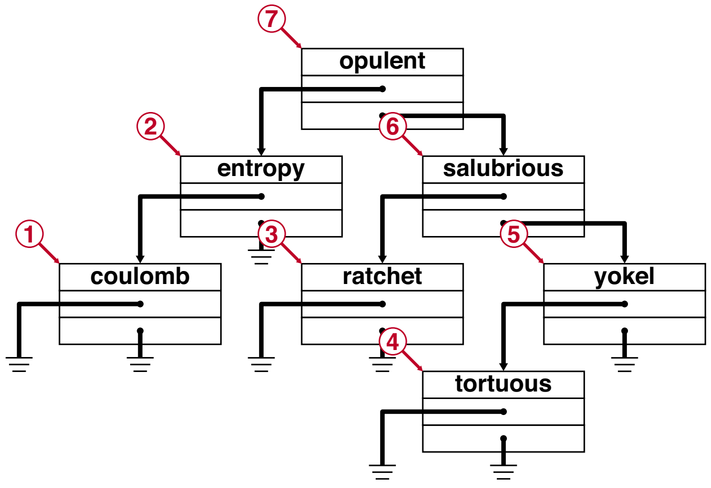


Figure 11.11 *A postorder search of the same tree.*

Recursion and binary trees are two extremely powerful programming tools. Learn how to use them—they'll pay big dividends.

Function Pointers

Next on the list is the subject of **function pointers**. Function pointers are exactly what they sound like: pointers that point to functions. Up to now, the only way to call a function was to place its name in the source code:

```
MyFunction();
```

Function pointers give you a new way to call a function. Function pointers allow you to say, "Execute the function pointed to by this variable." Here's an example:

```
int  (*myFuncPtr)( float );
```

This line of code declares a function pointer named **`myFuncPtr`**. **`myFuncPtr`** is a pointer to a function that takes a single parameter, a **`float`**, and that returns an **`int`**. The parentheses in the declaration are all necessary. The first pair tie the **`*`** to **`myFuncPtr`**, ensuring that **`myFuncPtr`** is declared as a pointer. The second pair surround the parameter list and distinguish **`myFuncPtr`** as a function pointer.

Suppose we had a function called **`DealTheCards()`** that took a **`float`** as a parameter and returned an **`int`**. This line of code assigns the address of **`DealTheCards()`** to the function pointer **`myFuncPtr`**:

```
myFuncPtr = DealTheCards;
```

Notice that the parentheses were left off the end of **`DealTheCards()`**. This is critical. If the parentheses were there, the code would have called **`DealTheCards()`**, returning a value to **`myFuncPtr`**. You may also have noticed that the **`&`** operator wasn't used. When you refer to a function without using the parentheses at the end, the compiler knows you are referring to the address of the function.

Now that you have the function's address in the function pointer, there's only one thing left to do—call the function. Here's how it's done:

```
int  result;

result = (*myFuncPtr)( 3.5 );
```

This line calls the function **`DealTheCards()`**, passing it the parameter 3.5, and returning the function value to the **`int result`**. You could also have called the function this way:

```
int  result;

result = myFuncPtr( 3.5 );
```

Some older (non-ISO compliant) compilers can't handle this form, but it is easier on the eye.

There's a lot you can do with function pointers. You can create an array of function pointers. How about a binary tree of function pointers? You can pass a function pointer as a parameter to another function. Taking this one step further, you can create a function that does nothing but call other functions. Cool!

For your enjoyment, there's a function-calling project in the *Learn C Projects* folder, inside the *11.03 - funcPtr* subfolder. The program is pretty simple, but it should serve as a useful reference when you start using function pointers in your own programs.

Initializers

When you declare a variable, you can also provide an initial value for the variable at the same time. The format for integer types, floating point types, and pointers is as follows:

```
type variable = initializer;
```

In this case, the initializer is just an expression. Here are a few examples:

```
float  myFloat = 3.14159;
int    myInt = 9 * 27;
int    *intPtr = &myInt;
```

If you plan on initializing a more complex variable, like an array, **`struct`**, or **`union`**, you'll use a slightly different form of initializer, embedding the elements used to initialize the variable between pairs of curly braces. Consider these two array declarations:

```
int   myInts[] = { 10, 20, 30, 40 };
float myFloats[ 5 ] = { 1.0, 2.0, 3.0 };
```

The first line of code declares an array of 4 **`int`**s, setting **`myInts[0]`** to 10, **`myInts[1]`** to 20, **`myInts[2]`** to 30, and **`myInts[3]`** to 40. If you leave out the array dimension, the compiler makes it just large enough to contain the listed data.

The second line of code includes a dimension, but not enough data to fill the array. The first three array elements are filled with the specified values, but **`myFloats[3]`** and **`myFloats[4]`** are initialized to 0.0.

If you don't provide enough values in your initializer list, the compiler initializes all the remaining elements to their **default initialization value**. For integers, the default initialization value is 0. For **`float`**s, it's 0.0. For pointers, it's **`NULL`**.

Here's another example:

```
char s[ 20 ] = "Hello";
```

What a convenient way to initialize an array of **`char`**s! Here's another way to accomplish the same thing:

```
char s[ 20 ] = { 'H', 'e', 'l', 'l', 'o', '\0'
 };
```

Once again, if you leave out the dimension, the compiler will allocate just enough memory to hold your text string, including a byte to hold the 0 terminator. If you include the dimension, the compiler will allocate that many array elements, then fill the array with whatever data you provide. If you provide more data than will fit in the array, your code won't compile.

Here's a **`struct`** example:

```
struct Numbers
{
 int   i, j;
 float  f;
}

struct Numbers myNums = { 1, 2, 3.01 };
```

As you can see, the three initializing values were wrapped in a pair of curly braces. This leaves **`myNums.i`** with a value of 1, **`myNums.j`** with a value of 2, and **`myNums.f`** with a value of 3.01. If you have a **`struct`**, **`union`**, or array embedded in your **`struct`**, you can nest a curly-wrapped list of values inside another list. For example:

```
struct Numbers
{
 int   i, j;
 float  f[ 4 ];
}

struct Numbers myNums1 = { 1, 2, {3.01, 4.01,
 5.01, 6.01} };
```

The Remaining Operators

If you go back to Chapter 5 and review the list of operators shown in Figure 5.7, you'll likely find a few operators you are not yet familiar with. Most of the ones we've missed were designed specifically to set the individual bits within a byte. For example, the | operator (not to be confused with its comrade, the logical || operator) takes two values and "ORs" their bits together, resolving to a single value. This operator is frequently used to set a particular bit to 1.

Check out this code:

```
short   myShort;

myShort = 0x0001 | myShort;
```

This code sets the rightmost bit of **`myShort`** to 1, no matter what its current value. This line of code, based on the **|=** operator, does the exact same thing:

```
myShort |= 0x0001;
```

The **&** operator takes two values and "ANDs" their bits together, resolving to a single value. This operator is frequently used to set a particular bit to 0 (more frequently referred to as **clearing** a bit).

Check out this code:

```
short              myShort;

myShort = 0xFFFE & myShort;
```

This code sets the rightmost bit of **`myShort`** to 0, no matter what its current value. It might help to think of **`0xFFFE`** as **`1111111111111110`** in binary.

This line of code, based on the **`&=`** operator, does the exact same thing:

```
myShort &= 0xFFFE;
```

The **^** operator takes two values and "XORs" their values together. It goes along with the **`^=`** operator. The **~** operator takes a single value and turns all the 1's into 0's and all the 0's into 1's. The **&**, **|**, **^**, and **~** operators are summarized in Figure 11.12.

A	B	A & B	A | B	A ^ B	~A
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

Figure 11.12 *A summary of the* **&**, **|**, **^**, *and* **~** *operators.*

The previous examples assumed that a **`short`** is two bytes (16 bits) long. Of course, this makes for some implementation dependent code. Here's a more portable example.

This code:

```
short   myShort;

myShort = (~1) & myShort;
```

sets the rightmost bit of **`myShort`**, no matter how many bytes are used to implement a **`short`**. You could also write this as:

```
myShort &= (~1);
```

The last of the binary operators, **`<<`**, **`>>`**, **`<<=`**, and **`>>=`** are used to **shift** bits within a variable, either to the left or to the right. The left operand is usually an **`unsigned`** variable and the right operand is a positive integer specifying how far to shift the variable's bits.

For example, this code shifts the bits of **`myShort`** 2 bits to the right:

```
unsigned short   myShort = 0x0100;

myShort = myShort >> 2;  /* equal to myShort
 >>= 2; */
```

`myShort` starts off with a value of **`0000000100000000`** and ends up with a value of `0000000001000000` (in hex, that's 0x0040). Notice that zeros get shifted in to make up for the leftmost bits that are getting shifted over and the rightmost bits are lost when they shift off the end.

These operators were designed to work with **`unsigned`** values only. Check with your compiler to see how it handles shifting of **`signed`** values.

The last two operators we need to cover are the **`,`** and **`:?`** operators. The **`,`** operator gives you a way to combine two expressions into a single expression. The **`,`** operator is binary and both operands are expressions. The left expression is evaluated first and the result is discarded. The right expression is then evaluated and its value is returned.

Here's an example:

```
for ( i=0, j=0; i<20 && j<40; i++,j+=2 )
 DoSomething( i, j );
```

This **`for`** loop is based on two variables instead of one. Before the loop is entered, **`i`** and **`j`** are both set to 0. The loop continues as long as **`i`** is less than 20 and **`j`** is lass than 40. Each time through the loop, **`i`** is incremented by 1 and **`j`** is incremented by 2.

The **`?`** and **`:`** operators combine to create something called a **conditional expression**. A conditional expression consists of a logical expression (an expression that evaluates to either **`true`** or **`false`**), followed by the **`?`** operator, followed by a second expression, followed by the **`:`** operator, followed by a third expression:

```
logical-expression ? expression2 : expression3
```

If the logical expression evaluates to **`true`**, **`expression2`** gets evaluated and the entire expression resolves to the value of **`expression2`**. If the logical expression evaluates to **`false`**, **`expression3`** gets evaluated and the entire expression resolves to the value of **`expression3`**.

Here's an example:

```
IsPrime( num ) ? DoPrimeStuff( num ) :
 DoNonPrimeStuff( num );
```

As you can see, a conditional expression is really a shorthand way of writing an **`if-else`** statement. Here's the **`if-else`** version of the previous example:

```
if ( IsPrime( num ) )
 DoPrimeStuff( num );
else
 DoNonPrimeStuff( num );
```

Some people like the brevity of the **`?:`** operator combination. Others find it hard to read. As always, make your choice and stick with it.

A word of advice: Don’t overuse the **?:** operator. For example, suppose you wanted to use **?:** to generate a number’s absolute value. You might write code like this:

```
int  value;

value - (value<0) ? (-value) : (value);
```

Though this code works, take a look at this code translated into its **if-else** form:

```
int  value;

if ( value<0 )

  value = (-value);

else

  value = (value);
```

As you can see, the **?:** operator can lead you to write source code that you would otherwise consider pretty darn silly.

Creating Your Own Types

The **typedef** statement lets you use existing types to create brand new types you can then use in your declarations. You’ll declare this new type just as you would a variable, except you’ll precede the declaration with the word **typedef** and the name you declare will be the name of a new type. Here’s an example:

```
typedef  int   *IntPointer;

IntPointer    myIntPointer;
```

The first line of code creates a new type named **IntPointer**. The second line declares a variable named **myIntPointer** which is a pointer to an **int**.

Here’s another example:

```
typedef  float   (*FuncPtr)( int * );

FuncPtr  myFuncPtr;
```

The first line of code declares a new type named **FuncPtr**. The second line declares a variable named **myFuncPtr** which is a pointer to a function which returns a **float** and which takes a single **int** as a parameter.

Enumerated Types

In a similar vein, the **`enum`** statement lets you declare a new type known as an enumerated type. An enumerated type is a set of named integer constants, collected under a single type name. A series of examples will make this clear.

```
enum Weekdays
{
 Monday,
 Tuesday,
 Wednesday,
 Thursday,
 Friday
};

enum Weekdays  whichDay;

whichDay = Thursday;
```

This code starts off with an **`enum`** declaration. The **`enum`** is given the name **`Weekdays`** and consists of the constants **`Monday`**, **`Tuesday`**, **`Wednesday`**, **`Thursday`**, and **`Friday`**. The second line of code uses this new enumerated type to declare a variable named **`whichDay`**. **`whichDay`** is an integer variable that can take on any of the **`Weekdays`** constants, as evidenced by the last line of code, which assigns the constant **`Thursday`** to **`whichDay`**.

Here's another example:

```
enum Colors
{
 red,
 green = 5,
 blue,
 magenta,
 yellow = blue + 5
} myColor;

myColor = blue;
```

This code declares an enumerated type named **`Colors`**. Notice that some of the constants in the **`Colors`** list are accompanied by initializers. When the compiler creates the enumeration constants, it numbers them sequentially, starting with 0. In the previous example, **`Monday`** has a value of 0, **`Tuesday`** has a value of 1, and so on, with **`Friday`** having a value of 4.

In this case, the constant **`red`** has a value of 0. But the constant **`green`** has a value of 5. Things move along from there, with **`blue`** and **`magenta`** having values of 6 and 7, respectively. Next, **`yellow`** has a value of **`blue+5`**, which is 11.

This code also declares an enumeration variable named **`myColor`**, which is then assigned a value of **`blue`**.

You can declare an enumerated type without the type name:

```
enum

{

  chocolate,

  strawberry,

  vanilla

};

int iceCreamFlavor = vanilla;
```

This code declares a series of enumeration constants with values of 0, 1, and 2. We can assign the constants to an **`int`**, as we did with **`iceCreamFlavor`**. This comes in handy when you need a set of integer constants but have no need for a tag name.

Static Variables

Normally, when a function exits, the storage for its variables is freed up and their values are no longer available. By declaring a local variable as **`static`**, the variable's value is maintained across multiple calls of the same function.

Here's an example:

```
int  StaticFunc( void )
{
 static int  myStatic = 0;

 return myStatic++;
}
```

This function declares an **`int`** named **`myStatic`** and initializes it to a value of 0. The function returns the value of **`myStatic`** and increments **`myStatic`** after the return value is determined. The first time this function is called, it returns 0 and **`myStatic`** is left with a value of 1. The second time **`StaticFunc()`** is called, it returns 1 and **`myStatic`** is left with a value of 2.

Take a few minutes and try this code out for yourself. You'll find it in the *Learn C Projects* folder in the subfolder *11.04 - static*.

One of the keys to this function is the manner in which **`myStatic`** received its initial value. Imagine if the function looked like this:

```
int  StaticFunc( void )
{
 static int   myStatic;

 myStatic = 0;    /* <-- Bad idea.... */

 return myStatic++;
}
```

Each time through the function, we'd be setting the value of **`myStatic`** back to 0. This function will always return a value of 0. Not what we want, eh?

The difference between the two functions? The first version sets the value of **`myStatic`** to 0 by initialization (the value is specified within the declaration). The second version sets the value of **`myStatic`** to 0 by assignment (the value is specified after the declaration). If a variable is marked as **`static`**, any initialization is done once and once only. Be sure you set the initial value of your **`static`** variable in the declaration and not in an assignment statement.

One way to think of **`static`** variables is as global variables that are limited in scope to a single function.

More on Strings

The last topic we'll tackle in this chapter is **string manipulation**. Although we've done some work with strings in previous chapters, there are a number of Standard Library functions that haven't been covered. Each of these functions requires that you include the file **`<string.h>`**. Here are a few examples...

strcpy()

`strcpy()` is declared as follows:

```
char *strcpy( char *dest, const char *source
 );
```

`strcpy()` copies the string pointed to by **`source`** into the string pointed to by **`dest`**. **`strcpy()`** copies each of the characters in **`source`**, including the terminating 0 byte. That leaves **`dest`** as a properly terminated string. **`strcpy()`** returns the pointer **`dest`**.

An important thing to remember about **`strcpy()`** is that you are responsible for ensuring that **`source`** is properly terminated, and that enough memory is allocated for the string returned in **`dest`**. Here's an example of **`strcpy()`** in action:

```
char name[ 20 ];

strcpy( name, “Dave Mark” );
```

This example uses a string literal as the source string. The string is copied into the array **`name`**. The return value was ignored.

strcat()

`strcat()` is declared as follows:

```
char *strcat( char *dest, const char *source
 );
```

`strcat()` appends a copy of the string pointed to by **`source`** onto the end of the string pointed to by **`dest`**. As was the case with **`strcpy()`**, **`strcat()`** returns the pointer **`dest`**. Here's an example of **`strcat()`** in action:

```
char name[ 20 ];

strcpy( name, “Dave “ );
strcat( name, “Mark” );
```

The call of **`strcpy()`** copies the string **`“Dave “`** into the array **`name`**. The call of **`strcat()`** copies the string **`“Mark”`** onto the end of **`dest`**, leaving **`dest`** with the properly terminated string **`“Dave Mark”`**. Again, the return value was ignored.

strcmp()

`strcmp()` is declared as follows:

```
int strcmp( const char *s1, const char *s2 );
```

`strcmp()` compares the strings **`s1`** and **`s2`**. **`strcmp()`** returns 0 if the strings are identical, a positive number if **`s1`** is greater than **`s2`**, and a negative number if **`s2`** is greater than **`s1`**. The strings are compared one byte at a time. If the strings are not equal, the first byte that is not identical determines the return value.

Here's a sample:

```
if ( strcmp( “Hello”, “Goodbye” ) )
 printf( “The strings are not equal!” );
```

Notice that the **`if`** succeeds when the strings are not equal.

strlen()

`strlen()` is declared as follows:

```
size_t  strlen( const char *s );
```

`strlen()` returns the length of the string pointed to by **`s`**. As an example, this call:

```
  length = strlen( "Aardvark" );
```

returns a value of 8, the number of characters in the string, not counting the terminating zero.

More Standard Library

There is a lot more to the Standard Library than what we've covered in the book. Having made it this far, consider yourself an official C programmer. You now have a sworn duty to dig in to the *C Standard Library* page we've referred to throughout the book. In case you haven't bookmarked it yet:

http://www.infosys.utas.edu.au/info/documentation/C/CStdLib.html

A good place to start is with the functions declared in **`<string.h>`**. Find out what the difference is between **`strcmp()`** and **`strncmp()`**. Wander around. Get to know the Standard Library.

What's Next?

Chapter 12 answers the question, "Where do you go from here?" Do you want to learn to create programs with that special Macintosh look and feel? Would you like more information on data structures and C programming techniques? Chapter 12 offers some suggestions to help you find your programming direction.

Exercises

1) What's wrong with each of the following code fragments:

a)

```
struct Dog
{
 struct Dog *next;
} ;

struct Cat
{
 struct Cat *next;
} ;

struct DogmyDog;
struct CatmyCat;

myDog.next = (struct Dog)&myCat;
myCat.next = NULL;
```

b)

```
int *MyFunc( void );
typedef   int (*FuncPtr)();

FuncPtr   myFuncPtr = MyFunc;
```

c)

```
union Number
{
 int      i;
 float    f;
 char     *s;
} ;

Number    myUnion;

myUnion.f = 3.5;
```

d)

```
struct Player
{
 int      type;
 char     name[ 40 ];
 int      team;
 union
 {
   int    myInt;
   float  myFloat;
 } u;
} myPlayer;

myPlayer.team = 27;
myPlayer.myInt = -42;
myPlayer.myFloat = 5.7;
```

e)

```
int       *myFuncPtr( int );

myFuncPtr = main;
*myFuncPtr();
```

f)

```
char      s[ 20 ];

strcpy( s, "Hello " );

if ( strcmp( s, "Hello" ) )
 printf( "The strings are the same!" );
```

g)

```
char *s;

s = malloc( 20 );
strcpy( "Heeeers Johnny!", s );
```

h)

```
char *s;

strcpy( s, "Aardvark" );
```

i)

```
void DoSomeStuff( void )
{
 /* stuff done here */
}

int main( void )
{
 int      ii;

 for ( ii = 0; ii < 10; ii++ )
   DoSomeStuff;

 return 0;
}
```

2) Write a program that reads in a series of integers from a file, storing the numbers in a binary tree in the same fashion as the words were stored earlier in the chapter. Store the first number as the root of the tree. Next, store the second number in the left branch if it is less than the first number, right branch if it is greater than or equal to the first number. Continue this process until all the numbers are stored in the tree.

Now write a series of functions that print the contents of the tree using preorder, inorder, and postorder recursive searches.

Chapter 12 Where Do I Go From Here?

Now that you've mastered the fundamentals of C, you're ready to dig into the specifics of Macintosh programming. As you've run the example programs in the previous chapters, you've probably noticed that none of the programs sport the look and feel that make a Mac program a Mac program.

For one thing, all of the interaction between you and your program focuses on the keyboard and the console window. None of the programs take advantage of the mouse. None offer color graphics, pull-down menus, buttons, checkboxes, scrolling windows or any of the thousand things that make Mac OS X applications so special. These things are all part of the Macintosh user interface.

The Macintosh User Interface

User interface is the part of your program that interacts with the user. So far, your user interface skills have focused on writing to and reading from the console window, using functions such as `printf()`, `scanf()`, and `getchar()`. The advantage of this type of user interface is that each of the aforementioned functions is available on every machine that supports the C language. Programs written using the Standard C Library are extremely portable.

On the down side, console-based user interfaces tend to be limited. With a console-based interface, you can't use an elegant graphic to make a point. Text-based interfaces can't provide animation or digital sound. In a nutshell, the console-based interface is simple and, at the same time, simple to program. Mac OS X's **graphical user interface** (**GUI**) offers an elegant, more sophisticated method of working with a computer.

Objective C and Cocoa

Your Mac just wouldn't be the same without windows, pull-down and pop-up menus, icons, push buttons, and scroll bars. You can and should add these user interface elements to your programs. Fortunately, the set of Apple developer tools you downloaded and installed at the beginning of this book include everything you need to build world-class applications with all the elements that make the Mac great!

The key to working with these elements is understanding Objective-C and Cocoa. The Objective-C language is a superset of C, developed by the same folks who designed and built Mac OS X. There are a number of excellent resources available for learning Objective-C. One of them is right there on your hard drive.

At the top level of your hard drive, find the *Developer* folder, the same one that holds the Xcode application. Follow this path to find the file named *ObjC.pdf*:

```
Developer/Documentation/Cocoa/Conceptual/
 ObjectiveC/ObjC.pdf
```

Another way to find this document is to use the search field in the upper right corner of your Finder window. Open a new Finder window, then type *objc.pdf* in the search field. You should see something similar to the result shown in Figure 12.1. Click on the file and the path to it will be shown at the bottom of the window. You can also double-click on the file to open it in your default PDF reader.

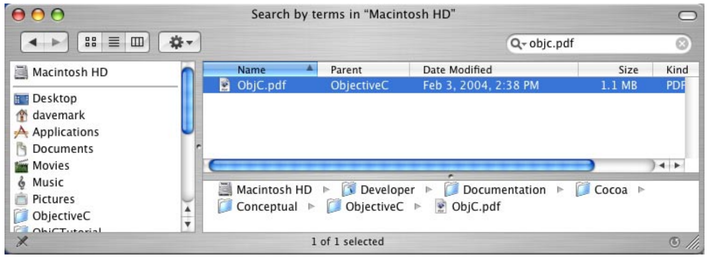


***Figure 12.1** Finding the Objective C documentation on your hard drive.*

I *love* this document. It is very well written, detailed and, best of all, it is free! Take a few minutes to read through the first few pages. If you feel comfortable with the language and the tone, you've found your path to learning Objective-C.

If this doc makes your eyes glaze over and you start to feel a bit gassy, there are plenty of other ways to learn Objective C. If you like the experience you had reading this book, check out the sequel from SpiderWorks (http://www.spiderworks.com), called *Learn Objective-C* by Mark Dalrymple. Mark is one of the smartest people I know and he does an excellent job explaining the concepts behind the Objective-C language. Not free, but at $9.95, still a great deal.

So what's Objective-C got that regular old C doesn't? In a word, objects. Just as a **`struct`** brings variables together under a single name, an object can bring together variables *as well as* functions, binding them together under a single class name.

Objects are incredibly powerful. Every part of the Mac user interface has a set of objects associated with it. Want to create a new window? Just create a new window object and the object will take care of all the housekeeping associated with maintaining a window. The window object's functions will draw the contents of the window for you, perhaps communicating with other objects to get them to draw themselves within the window.

There are pull-down menu objects, icon objects, scrollbar objects, file objects, even objects that can organize other objects. Chances are, if you can imagine it, there's a set of objects that will help you build it.

Learning Cocoa

Learning Objective-C will teach you the mechanics of working with objects. Once you get that down, you'll turn your attention to Cocoa, Apple's object library. Cocoa is an extensive collection of objects that will allow you to implement pretty much every aspect of the Mac OS X experience.

As you might expect, Apple's developer tools contain some excellent Cocoa documentation. For starters, check out:

```
Developer/Documentation/Cocoa/Conceptual/
 CocoaOverview/index.html
```

Again, some excellent doc here, and it's free.

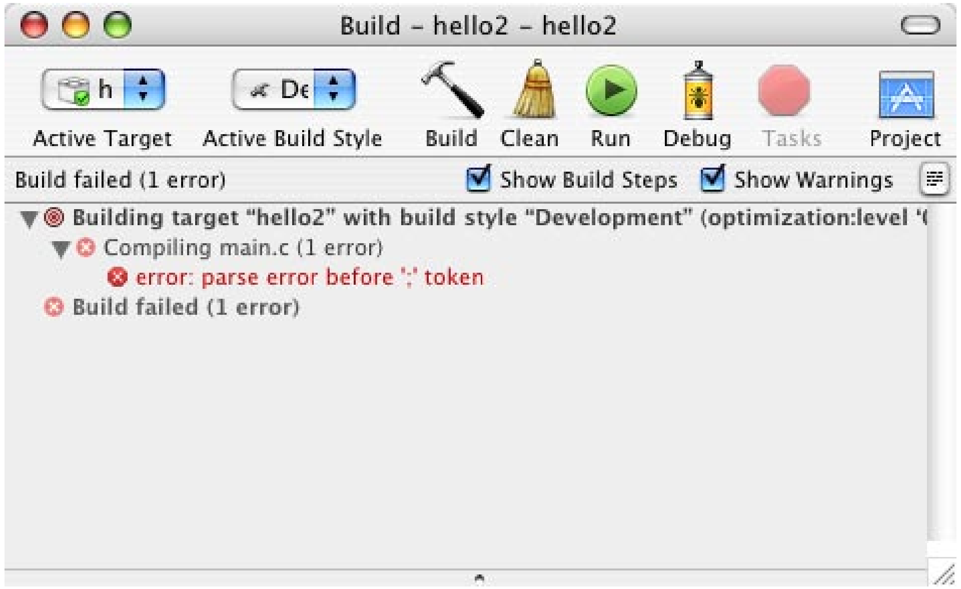
Go Get 'Em

Well, that's about it. I hope you enjoyed reading this book as much as I enjoyed writing it. Above all, I hope you are excited about your newfound programming capabilities. By learning C, you've opened the door to an exciting new adventure. You can move on to Objective-C and Cocoa, tackle web programming with PHP, move into the Windows universe with C#, or do it all with Java. There are so many choices out there. And they are all based on C.

Go on out there and write some code. And keep in touch!

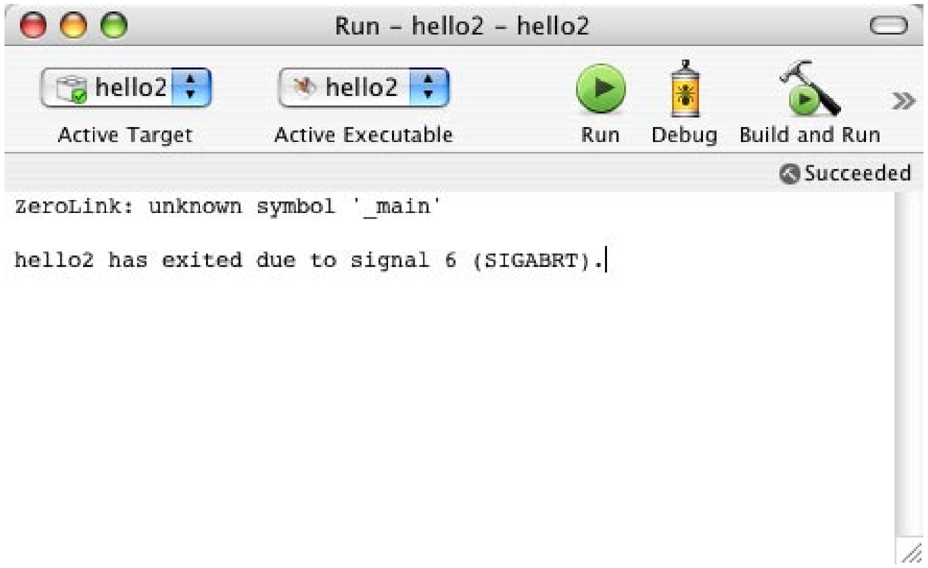
Chapter 4

1)



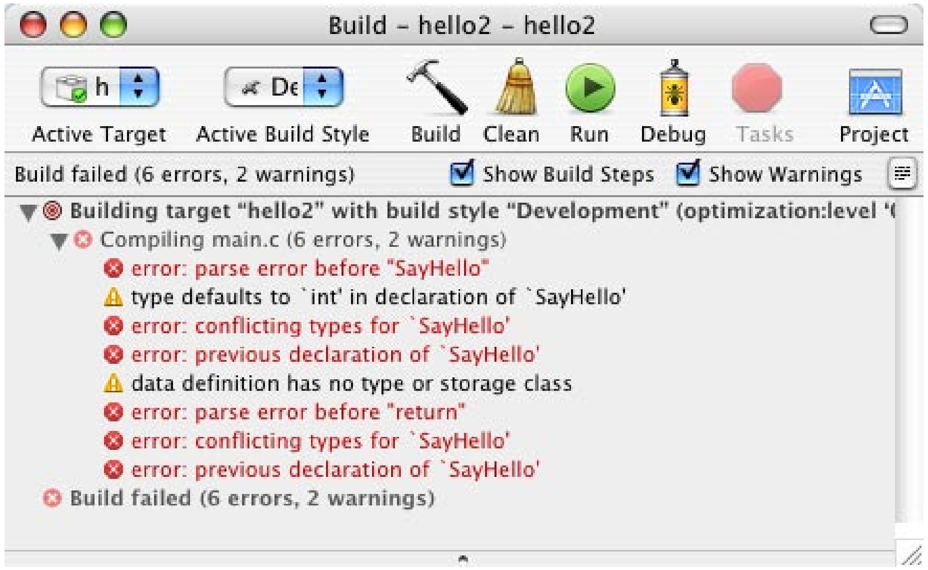
```
Build failed (1 error)
▼ Building target "hello2" with build style "Development" (optimization:level '
  ▼ Compiling main.c (1 error)
      error: parse error before ';' token
  Build failed (1 error)
```

2)

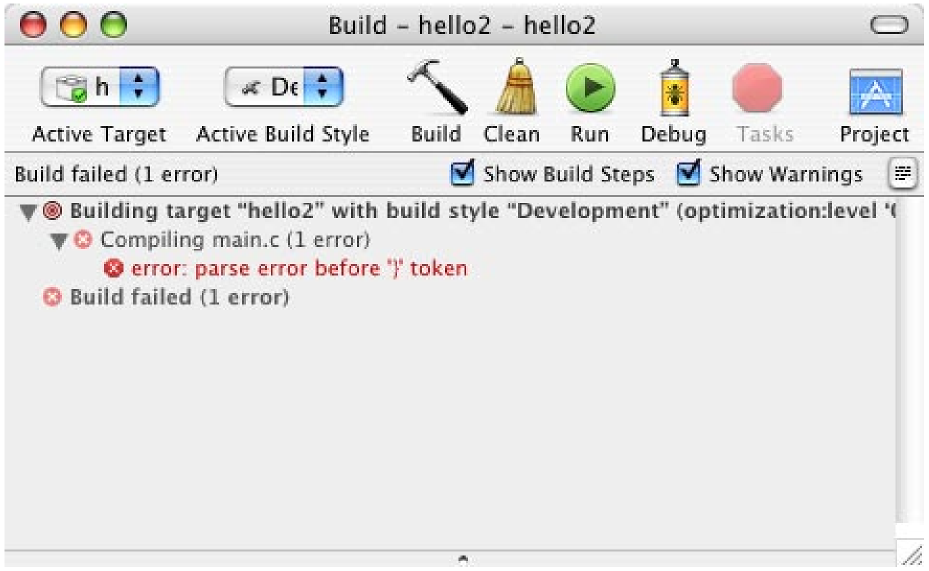


```
ZeroLink: unknown symbol '_main'

hello2 has exited due to signal 6 (SIGABRT).
```

3)



```
Build failed (6 errors, 2 warnings)
▼ Building target "hello2" with build style "Development" (optimization:level '(
  ▼ Compiling main.c (6 errors, 2 warnings)
      error: parse error before "SayHello"
      type defaults to `int' in declaration of `SayHello'
      error: conflicting types for `SayHello'
      error: previous declaration of `SayHello'
      data definition has no type or storage class
      error: parse error before "return"
      error: conflicting types for `SayHello'
      error: previous declaration of `SayHello'
  Build failed (6 errors, 2 warnings)
```

4) 

```
Build failed (1 error)
Building target "hello2" with build style "Development" (optimization:level '
  Compiling main.c (1 error)
    error: parse error before '}' token
Build failed (1 error)
```

Chapter 5

1) Find the error in each of the following code fragments:

a. Missing quotes around **`"Hello, World"`**

b. Missing comma between two variables

c. **`=+`** should be **`+=`** (though this will compile with some older compilers)

d. Missing 2nd parameter to **`printf()`**. Note that this error won't be caught by the compiler and is known as a run-time error.

e. Another run-time error. This time, you are missing the **`%d`** in the first argument to **`printf()`**.

f. This time we've either got an extra **`\`** or are missing an **`n`** following the **`\`** in the first **`printf()`** parameter.

g. The left and right-hand-side of the assignment are switched.

h. The declaration of **`anotherInt`** is missing.

2) Compute the value of **`myInt`** after each code fragment is executed:

a. 70

b. -6

c. -1

d. 4

e. -8

f. 2

g. 14

h. 1

Chapter 6

1) What's wrong with each of the following code fragments?

a. The **`if`** statement's expression should be surrounded by parentheses.

b. We increment **`i`** inside the **`for`** loop's expression, then decrement it in the body of the loop. This loop will never end!

c. The **`while`** loop has parentheses, but is missing an expression.

d. The **`do`** statement should follow this format:

```
do
   statement
while ( expression ) ;
```

e. Each **`case`** in this **`switch`** statement contains a text string, which is illegal. Also, **`case default`** should read **`default`**.

f. The **`printf()`** will never get called.

g. This is probably the most common mistake made by C programmers. The assignment operator (**`=`**) is used instead of the logical equality operator (**`==`**). Since the assignment operator is perfectly legal inside an expression, the compiler won't find this error. An annoying little error you'll encounter again and again!

h. Once again, this code will compile, but it likely is not what you wanted. The third expression in the **`for`** loop is usually an assignment statement - something to move **`i`** towards its terminating condition. The expression **`i*20`** is useless here, since it doesn't change anything.

2) Look in the folder *06.05 - nextPrime2*.

3) Look in the folder *06.06 - nextPrime3*.

Chapter 7

1) Predict the result of each of the following code fragments:

   a. Final value is 25.

   b. Final value is 512. Try changing the **for** loop from 2 to 3. Notice that this generates a number too large for a 2-byte **int** to hold. Now change the **for** loop from 3 to 4. This generates a number too large for even a 4-byte **int** to hold. Be aware of the size of your types!

   c. Final value is 1024.

2) Look in the folder *07.06 - power2*.

3) Look in the folder *07.07 - nonPrimes*.

Chapter 8

1) What's wrong with each of the following code fragments:

   a. If the **char** type defaults to **signed** (very likely), **c** can only hold values from -128 to 127. Even if your **char** does default to **unsigned**, this is dangerous code. At the very least, use an **unsigned char**. Even better, use a **short**, **int**, or **long**.

   b. Use **%f**, **%g**, or **%e** to print the value of a **float**, not **%d**.

   c. The text string **"a"** is composed of two characters, both **'a'** and the terminating zero byte. The variable **c** is only a single byte in size. Even if **c** were 2 bytes long, you can't copy a text string this way. Try copying the text one byte at a time into a variable large enough to hold the text string and its terminating zero byte.

   d. Once again, this code uses the wrong approach to copying a text string, and once again there is not enough memory allocated to hold the text string and its zero byte.

   e. The **#define** of **kMaxArraySize** must come before the first non-**#define** reference to it.

   f. This definition:

```
char c[ kMaxArraySize ];
```

creates an array ranging from **`c[0]`** to **`c[kMaxArraySize-1]`**. The reference to **`c[kMaxArraySize]`** is out of bounds.

g. The problem occurs in the line:

```
cPtr++ = 0;
```

This line assigns the pointer variable **`cPtr`** a value of **`0`** (making it point to location **`0`** in memory) then increments it to 1 (making it point to location 1 in memory). This code will not compile. Here's a more likely scenario:

```
*cPtr++ = 0;
```

This code sets the **`char`** that **`cPtr`** points to to **`0`**, then increments **`cPtr`** to point to the next **`char`** in the array.

h. The problem here is with the statement:

```
c++;
```

You can't increment an array name. Even if you could, if you increment **`c`**, you no longer have a pointer to the beginning of the array! A more proper approach is to declare an extra **`char`** pointer, assign **`c`** to this **`char`** pointer, then increment the copy of **`c`**, rather than **`c`** itself.

i. You don't need to terminate a **`#define`** with a semicolon. This statement defines "**`kMaxArraySize`**" to "**`200;`**", probably not what we had in mind.

2) Look in the folder *08.08 - dice2*.

3) Look in the folder *08.09 - wordCount2*.

Chapter 9

1) What's wrong with each of the following code fragments:

a. The semicolon after **`employeeNumber`** is missing.

b. This code is really pretty useless. If the first character returned by **`getchar()`** is **`'\n'`**, the **`;`** will get executed, otherwise the loop just exits. Try changing the **`==`** to **`!=`** and see what happens.

c. This code will actually work, since the double-quotes around the header file name tell the compiler to search the local directory in addition to the places it normally searches for system header files. On the other hand, it is considered better form to place angle brackets around a system header file like **`<stdio.h>`**.

d. The **`name`** field is missing its type. As it turns out, this code will compile, but it might not do what you think it does. Since the type is missing, the C compiler assumes you want an array of **`int`**s. Even though it compiles, this is bad form!

e. **`next`** and **`prev`** should be declared as pointers.

f. There are several problems with this code. First, the **`while`** loop is completely useless. Also, the code should use **`'\0'`** instead of **`0`** (though that's really a question of style). Finally, by the time we get to the **`printf()`**, **`line`** points beyond the end of the string!

2) Look in the folder *09.06 - cdTracker2*.

3) Look in the folder *09.07 - cdTracker3*.

Chapter 10

1) What's wrong with each of the following code fragments:

   a. The arguments to **`fopen()`** appear in reverse order.

   b. Once again, the arguments to **`fopen()`** are reversed. In addition, the first parameter to **`fscanf()`** contains a prompt, as if you were calling **`printf()`**. Also, the second parameter to **`fscanf()`** is defined as a **`char`**, yet the **`%d`** format specifier is used, telling **`fscanf()`** to expect an **`int`**. This will cause **`fscanf()`** to store an **`int`**-sized value in the space allocated for a **`char`**. Not good!

   c. **`line`** is declared as a **`char`** pointer instead of as an array of **`char`**s. No memory was allocated for the string being read in by **`fscanf()`**. Also, since **`line`** is a pointer, the **`&`** in the **`fscanf()`** call shouldn't be there.

   d. This code is fine except for one problem. The file is opened for writing, yet we are trying to read from the file using **`fscanf()`**.

2) Look in the folder *10.04 - fileReader*

3) Look in the folder *10.05 - cdFiler2*

Chapter 11

1) What's wrong with each of the following code fragments:

   a. In the next to last line, the address of **`myCat`** is cast to a **`struct`**. Instead, the address should be cast to a **`(struct Dog *)`**.

   b. The **`typedef`** defines **`FuncPtr`** to be a pointer to a function that returns an **`int`**. **`MyFunc()`** is declared to return a pointer to an **`int`**, not an **`int`**.

   c. The declaration of **`Number`** is missing the keyword **`union`**. Here's the corrected declaration:

```
union     Number myUnion;
```

   d. The **`Player union`** fields must be accessed using **`u`**. Instead of **`myPlayer.myInt`**, refer to **`myPlayer.u.myInt`**. Instead of **`myPlayer.myFloat`**, refer to **`myPlayer.u.myFloat`**.

   e. First off, **`myFuncPtr`** is not a function pointer and not a legal l-value. As is, the declaration just declares a function named **`myFuncPtr`**. This declaration fixes that problem:

```
int       (*myFuncPtr)( int );
```

Next, **`main()`** doesn't take a single **`int`** as a parameter. Besides that, calling **`main()`** yourself is a questionable practice. Finally, to call the function pointed to by **`myFuncPtr`**, use either **`myFuncPtr();`** or **`(*myFuncPtr)();`** instead of **`*myFuncPtr();`**

f. **`strcmp()`** returns zero if the strings are equal. The **`if`** would fail if the strings were the same. The message passed to **`printf()`** is wrong.

g. The parameters passed to **`strcpy()`** should be reversed.

h. No memory was allocated for **`s`**. When **`strcpy()`** copies the string, it will be writing over unintended memory.

i. This is a common problem that tons of people, including battle-scarred veterans, run into. The function call in the loop is not actually a function call. Instead, the address of the function **`DoSomeStuff`** is evaluated. Because this address is not assigned to anything or used in any other way, the result of the evaluation is discarded. The expression "**`DoSomeStuff;`**" is effectively a no-op, making the entire loop a no-op.

2) Look in the folder *11.05 - treePrinter*.

License Agreement