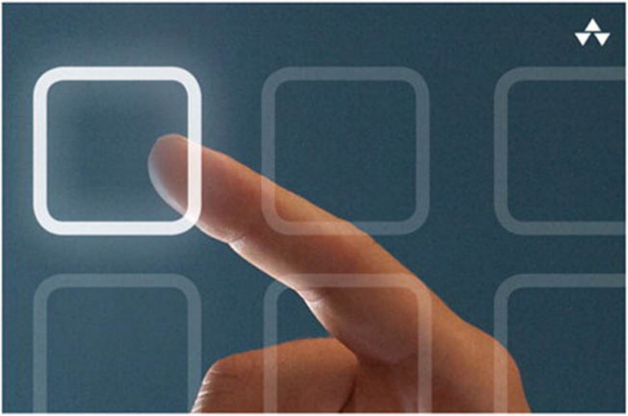
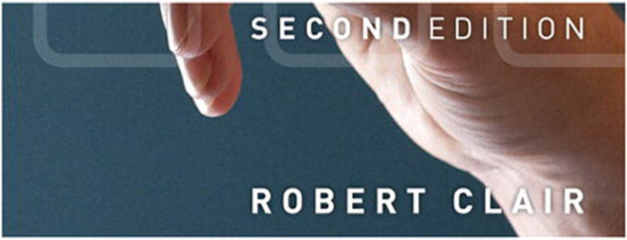


LEARNING OBJECTIVE-C 2.0

A Hands-on Guide to Objective-C for Mac and iOS Developers



Praise for the First Edition of *Learning Objective-C 2.0*

"With *Learning Objective-C 2.0*, Robert Clair cuts right to the chase and provides not only comprehensive coverage of Objective-C, but also time-saving and headache-preventing insights drawn from a depth of real-world, hands-on experience. The combination of concise overview, examples, and specific implementation details allows for rapid, complete, and well-rounded understanding of the language and its core features and concepts."

—Scott D. Yelich, Mobile Application Developer

"There are a number of books on Objective-C that attempt to cover the entire gamut of object-oriented programming, the Objective-C computer language, and application development on Apple platforms. Such a range of topics is far too ambitious to be covered thoroughly in a single volume of finite size. Bob Clair's book is focused on mastering the basics of Objective-C, which will allow a competent programmer to begin writing Objective-C code."

—Joseph E. Sacco, Ph.D., J.E. Sacco & Associates, Inc.

"Bob Clair's *Learning Objective-C 2.0* is a masterfully crafted text that provides in-depth and interesting insight into the Objective-C language, enlightening new programmers and seasoned pros alike. When programmers new to the language ask about where they should start, this is the book I now refer them to."

—Matt Long, Cocoa Is My Girlfriend (www.cimgf.com)

"Robert Clair has taken the Objective-C language and presented it in a way that makes it even easier to learn. Whether you're a novice or professional programmer, you can pick up this book and begin to follow along without knowing C as a prerequisite."

—Cory Bohon, Indie Developer and Blogger for *Mac|Life*

"I like this book because it is technical without being dry, and readable without being fluffy."

—Andy Lee, Author of AppKiDo

This page intentionally left blank

Learning Objective-C 2.0

Second Edition

Learning Objective-C 2.0

A Hands-on Guide to Objective-C for Mac and iOS Developers

Second Edition

Robert Clair

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Cataloging-in-Publication Data is on file with the Library of Congress.

ISBN-13: 978-0-321-83208-5
ISBN-10: 0-321-83208-6
Text printed in the United States on recycled paper at Edwards Brothers Malloy, Ann Arbor, Michigan.
First printing, November 2012

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Senior Development Editor
Chris Zahn

Managing Editor
John Fuller

Project Editor
Caroline Senay

Copy Editor
Barbara Wood

Indexer
Richard Evans

Proofreader
Lori Newhouse

Technical Reviewers
Duncan Champney
Joseph Sacco
Scott Yelich

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Rob Mauhar
The CIP Group

❖

To the memory of my parents,
Selma B. and Martin H. Clair,
and to Ekko

❖

This page intentionally left blank

Contents at a Glance

Contents

This page intentionally left blank

Preface

Objective-C is an object-oriented extension to C. You could call it "C with Objects." If you're reading this book, you're probably interested in learning Objective-C so that you can write applications for Mac OS X or for iOS. But there's another reason to learn Objective-C: It's a fun language and one that is relatively easy to learn. Like anything else in the real world, Objective-C has some rough spots, but on the whole it is a much simpler language than some other object-oriented languages, particularly C++. The additions that Objective-C makes to C can be listed on a page or two.

Objective-C was initially created by Brad J. Cox in the early 1980s. In 1988, NeXT Computer, the company started by Steve Jobs after he left Apple, licensed Objective-C and made it the basis of the development environment for creating applications to run under NeXT's NeXTSTEP operating system. The NeXT engineers developed a set of Objective-C libraries for use in building applications. After NeXT withdrew from the hardware business in 1993, it worked with Sun Microsystems to create OpenStep, an open specification for an object-oriented system, based on the NeXTSTEP APIs. Sun eventually lost interest in OpenStep. NeXT continued selling its version of OpenStep until NeXT was purchased by Apple in early 1997. The NeXTSTEP operating system became the basis of OS X.

In the Apple world, Objective-C does not work alone. It works in conjunction with a number of class libraries called *frameworks*. The two most important frameworks on OS X are the Foundation framework and the AppKit framework. The Foundation framework contains classes for basic entities, such as strings and arrays, and classes that wrap interactions with the operating system. The AppKit contains classes for windows, views, menus, buttons, and the assorted other widgets needed to build graphical user interfaces (GUIs). Together, the two frameworks are called Cocoa. On iOS a different framework called the UIKit replaces the AppKit. Together, Foundation and UIKit are called Cocoa Touch.

While it is technically possible to write complete OS X programs using other languages, writing a program that follows the Apple *Human Interface Guidelines*[1] and has a proper Mac "look and feel" requires the use of the Objective-C Cocoa frameworks. Even if you write the core of a Mac application in a different language, such as plain C or C++, your user interface layer should be written in Objective-C. When writing for iOS, there is no choice: An iOS app's outer layer and user interface must be written in Objective-C.

About This Book

This book concentrates on learning the Objective-C language. It will not teach you how to write Cocoa or Cocoa Touch programs. It covers and makes use of a small part of the Foundation framework and mentions the AppKit and UIKit only in passing. The book's premise is that you will have a much easier time learning Cocoa and Cocoa Touch programming if you first acquire a good understanding of the language on which Cocoa and Cocoa Touch are based.

Some computer books are written in what I like to think of as a "follow me" style. The user is invited to copy or download some code. A brief discussion of the code follows. As new features are introduced, the reader is asked to change the relevant lines of code and observe the results. After a bit of discussion it is on to the next feature. I find this style of book unsatisfying for a language book. Often there is very little explanation of how things actually work. This style of book can create a false sense of confidence that vanishes when the reader is faced with a programming task that is not a small variation on an example used in the book.

This book takes a more pedagogical approach and uses small examples to emphasize how the language works. In addition to learning the syntax of the language, you are encouraged to think about what is happening "under the hood." This approach requires a bit more mental effort on your part, but it will pay off the first time you face a novel programming task.

Who Should Read This Book

This book is intended for people with some prior programming experience who want to learn Objective-C in order to write programs for OS X or iOS. (iOS is used for the iPhone, the iPod touch, and the iPad.)

The book will also be useful for programmers who want to write Objective-C programs for other platforms using software from the GNUStep project,[2] an open-source implementation of the OpenStep libraries.

1. http://developer.apple.com/mac/library/documentation/UserExperience/Conceptual/AppleHIGuidelines.

2. www.gnustep.org/.

What You Need to Know

This book assumes a working knowledge of C. Objective-C is an extension of C; this book concentrates on what Objective-C adds to C. For those whose C is rusty and those who are adept at picking up a new language quickly, Chapters 1 and 2 form a review of the essential parts of C; those that you are likely to need to write an Objective-C program. If you have no experience with C or any C-like language (C++, Java, and C#), you will probably want to read a book on C in conjunction with this book. Previous exposure to an object-oriented language is helpful but not absolutely necessary. The required objected-oriented concepts are introduced as the book proceeds.

About the Examples

Creating code examples for an introductory text poses a challenge: how to illustrate a point without getting lost in a sea of boilerplate code that might be necessary to set up a working program. In many cases, this book takes the path of using somewhat hypothetical examples that have been thinned to help you concentrate on the point being discussed. Parts of the code that are not relevant are omitted and replaced by an ellipsis (`...`).

For example:

```
int averageScore = ...
```

The preceding line of code should be taken to mean that `averageScore` is an integer variable whose value is acquired from some other part of the program. The source of `averageScore`'s value isn't relevant to the example; all you need to consider is that it *has* a value.

About the Exercises

Most of the chapters in this book have a set of exercises at the end. You are, of course, encouraged to do them. Many of the exercises ask you to write small programs to verify points that were made in the chapter's text. Such exercises might seem redundant, but writing code and seeing the result provides a more vivid learning experience than merely reading. Writing small programs to test your understanding is a valuable habit to acquire; you should write one whenever you are unclear about a point, even if the book has not provided a relevant exercise. When I finished writing this book, I had a directory full of small test programs. When you finish with this book, you should have the same.

None of the programs suggested by the exercises require a user interface; all of them can be coded, compiled, and run either by writing the code with a text editor and compiling and running it from a command line, as shown before the exercises in Chapter 2, "More about C Variables," or by using a simple Xcode project, as shown in Chapter 4, "Your First Objective-C Program."

Objective-C—A Moving Target

Objective-C is a moving target. For the past several years Apple has been adding new features and syntax to Objective-C on a regular basis. Despite these added features, Apple has decided *not* to continue versioning the language. Objective-C 2.0 is as high as they are going to go. The only way to specify a particular version or feature set of the language is to refer to Objective-C as of a particular version of Xcode or a particular version of the LLVM compiler. This edition of the book covers Objective-C as of Xcode 4.4 (released with Mountain Lion, OS X 10.8), or, equivalently, Objective-C as implemented in the LLVM Compiler/Clang 4.0.

ARC or Not

Objective-C uses a memory management system called *retain counting*. Each object keeps a count of the number of other objects that are using it. When this count falls to zero, the object's memory is returned to the system for reuse. Historically, this system has been "manual"—you had to write code to manipulate an object's retain count at the appropriate times. The rules for this system have proved difficult for many people to follow correctly 100% of the time. The unfortunate consequences of not following the rules are memory leaks and crashes.

In the spring of 2011 Apple introduced *Automatic Reference Counting* (ARC). ARC automates the reference counting system by analyzing programs and automatically inserting code that keeps the retain count correctly. No coding on the part of the programmer is required.

Some people argue that ARC obviates the need to learn about manual memory management and how reference counting works. They say, "You don't need to know how the engine works to drive a car, and you don't need to know manual reference counting to write Objective-C programs with ARC." This is literally true. But just as some knowledge of how the car's engine works can be valuable, there are some situations where understanding manual memory management can be valuable or even essential:

- There is a lot of existing code that has not been converted to use ARC. If you are asked to work on non-ARC code or want to use an open-source project that is non-ARC, you will have to understand manual reference counting.
- There is a set of C language libraries ("Core"-level libraries) below the Objective-C frameworks. These libraries are written in an object-oriented fashion and have their own manual reference counting system. While it is best to use the Objective-C frameworks if you can, there are cases (in graphics, for example) where it is necessary to use the lower-level libraries. To use these libraries properly you must understand the concepts of manual reference counting.
- Some objects are "toll-free bridged" (see Chapter 8, "Frameworks"). A pointer to one of the low-level C objects can be cast to a pointer to an Objective-C framework object and vice versa. Doing this under ARC requires one of several

special casts. Deciding which cast to use requires an understanding of manual reference counting and what ARC is automating for you.

- There are some situations (for example, creating large numbers of temporary objects in a tight loop) where you can help keep the memory footprint of your program small by doing some manual tuning. Doing this tuning requires an understanding of how reference counting operates.
- ARC is still relatively new and there is still the odd bug or unexpected behavior in edge cases. If you encounter one of these, you need to understand what is happening behind the scenes in order to reason your way past the obstacle.

ARC presented me with a dilemma in preparing the second edition of this book. Should I abandon the sections on manual reference counting and just use ARC? I felt strongly that this would be a bad choice, but to help me decide I asked the question of a number of my colleagues. Their answers were unanimous: Understanding how reference counting works is important. Teach it first and then introduce ARC. Accordingly, this book teaches manual memory management until Chapter 16, "ARC." After you have absorbed the material in Chapter 16, you can return to the exercises in earlier chapters and do them using ARC. You will find it much easier to learn how to do manual reference counting and then enjoy the freedom of not doing it in most cases, than to have to learn it on an emergency basis because you have encountered one of the situations in the preceding list.

How This Book Is Organized

This book is organized into three sections. The first section is a review of C, followed by an introduction to object-oriented programming and Objective-C. The second section of the book covers the Objective-C language in detail, as well as an introduction to the Foundation framework. The final section of the book covers memory management in Objective-C and Objective-C blocks.

Part I: Introduction to Objective-C

- Chapter 1, "C, the Foundation of Objective-C," is a high-speed introduction to the essentials of C. It covers the parts of C that you are most likely to need when writing Objective-C programs.
- Chapter 2, "More about C Variables," continues the review of C with a discussion of the memory layout of C and Objective-C programs, and the memory location and lifetime of different types of variables. Even if you know C, you may want to read through this chapter. Many practicing C programmers are not completely familiar with the material it contains.
- Chapter 3, "An Introduction to Object-Oriented Programming," begins with an introduction to the concepts of object-oriented programming and continues with a first look at how these concepts are embodied in Objective-C.

- Chapter 4, "Your First Objective-C Program," takes you line by line through a simple Objective-C program. It also shows you how to use Xcode, Apple's integrated development environment, to create a project, and then compile and run an Objective-C program. You can then use this knowledge to do the exercises in the remainder of the book.

Part II: Language Basics

Objects are the primary entities of object-oriented programming; they group variables, called *instance variables*, and function-like blocks of code, called *methods*, into a single entity. *Classes* are the specifications for an object. A class lists the instance variables and methods that make up a given type of object and provides the code that implements those methods. An object is more tangible; it is a region of memory, similar to a C struct, that holds the variables defined by the object's class. A particular object is said to be an *instance* of the class that defines it.

- Chapter 5, "Messaging," begins the full coverage of the Objective-C language. In Objective-C, you get an object to "do something" by sending it a *message*. The message is the name of a method plus any arguments that the method takes. In response to receiving the message, the object executes the corresponding method. This chapter covers methods, messages, and how the Objective-C messaging system works.
- Chapter 6, "Classes and Objects," covers defining classes and creating and copying object instances. It also covers *inheritance*, the process of defining a class by extending an existing class, rather than starting from scratch.

  Each class used in an executing Objective-C program is represented by a piece of memory that contains information about the class. This piece of memory is called the class's *class object*. Classes can also define *class methods*, which are methods executed on behalf of the class rather than instances of the class.
- Chapter 7, "The Class Object," covers class objects and class methods. Unlike classes in some other object-oriented languages, Objective-C classes do not have class variables, variables that are shared by all instances of the class. The last sections of this chapter show you how to obtain the effect of class variables by using static variables.
- Chapter 8, "Frameworks," describes Apple's way of encapsulating dynamic link libraries. It covers the definition and structure of a framework and takes you on a brief descriptive tour of some of the common frameworks that you will encounter when writing OS X or iOS programs.
- Chapter 9, "Common Foundation Classes," covers the most commonly used Foundation classes: classes for strings, arrays, dictionaries, sets, and number objects.
- Chapter 10, "Control Structures in Objective-C," discusses some additional considerations that apply when you use Objective-C constructs with C control

structures. It goes on to cover the additional control structures added by Objective-C, including Objective-C 2.0's Fast Enumeration construct. The chapter concludes with an explanation of Objective-C's exception handling system.

- Chapter 11, "Categories, Extensions, and Security," shows you how to add methods to an existing class without having to subclass it and how to hide the declarations of methods and instance variables that you consider private. The chapter ends with a discussion of Objective-C security issues.
- Chapter 12, "Properties," introduces Objective-C 2.0's *declared properties* feature. Properties are characteristics of an object. A property is usually modeled by one of the object's instance variables. Methods that set or get a property are called *accessor methods*. Using the declared properties feature, you can ask the compiler to synthesize a property's accessor methods and its instance variable for you, thereby saving yourself a considerable amount of effort.
- Chapter 13, "Protocols," covers a different way to characterize objects. A *protocol* is a defined group of methods that a class can choose to implement. In many cases what is important is not an object's class, but whether the object's class *adopts* a particular protocol by implementing the methods declared in the protocol. (More than one class can adopt a given protocol, and a class can adopt more than one protocol.) The Java concept of an interface was borrowed from Objective-C protocols.

Part III: Advanced Concepts

The chapters in this section cover memory management in detail and Objective-C blocks.

- Chapter 14, "Memory Management Overview," is a discussion of the problem of memory management and a brief introduction to the two systems of memory management that Objective-C provides.
- Chapter 15, "Reference Counting," covers Objective-C's traditional manual reference counting system. Reference counting is also called *retain counting* or *managed memory*. In a program that uses reference counting, each object keeps a count, called a *retain count*, of the number of other objects that are using it. When that count falls to zero, the object is deallocated. This chapter covers the rules needed to keep your retain counts in good order.
- Chapter 16, "ARC," covers Automatic Reference Counting (ARC). ARC is not a completely different memory management system. Rather, ARC automates Objective-C's traditional reference counting system. ARC is a compile-time process. It analyzes your Objective-C code and inserts the appropriate memory management messages for you.
- Chapter 17, "Blocks," discusses Objective-C 2.0's *blocks* feature. A block is similar to an anonymous function, but, in addition, a block carries the values of the variables in its surrounding context with it. Blocks are a central feature of Apple's Grand Central Dispatch concurrency mechanism.

- Chapter 18, “A Few More Things,” covers a few minor items that did not fit elsewhere in the book.

Part IV: Appendices

- Appendix A, “Reserved Words and Compiler Directives,” provides a table of names that have special meaning to the compiler, and a list of Objective-C compiler directives. Compiler directives are words that begin with an `@` character; they are instructions to the compiler in various situations.
- Appendix B, “Toll-Free Bridged Classes,” gives a list of Foundation classes whose instances have the same memory layout as, and may be used interchangeably with, corresponding objects from the low-level C language Core Foundation framework.
- Appendix C, “32- and 64-Bit,” provides a brief discussion of 32-bit and 64-bit environments.
- Appendix D, “The Fragile Base Class Problem,” describes a problem that affects some object-oriented programming languages and how Objective-C avoids that problem.
- Appendix E, “Resources for Objective-C,” lists books and websites that have useful information for Objective-C developers.

We Want to Hear from You!

As a reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we’re doing right, what we could do better, what areas you’d like to see us publish in, and any other words of wisdom you’re willing to pass our way.

You can e-mail or write me directly to let me know what you did or didn’t like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book’s title and author as well as your name and phone or e-mail address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
1330 6th Avenue
New York, NY 10019

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Acknowledgments

As anyone who has ever written one knows, even a single-author book is a group effort. This book is no exception. Scott D. Yelich, Andy Lee, Matt Long, Cory Bohon, and Joachim Bean read and commented on the manuscript for the first edition. Scott and Duncan Champney performed the same services for the second edition. The readers not only found mistakes but also forced me to think more carefully about some issues that I had originally glossed over. Steve Peter started me on the path to writing this book, and Daniel Steinberg helped me with an earlier incarnation of it. At Addison-Wesley, I'd like to thank Romny French, Olivia Basegio, and my editors: Chuck Toporek (first edition) and Trina MacDonald (second edition). Chuck was especially sympathetic to my frustrations and grumpiness as a (then) first-time user of MS Word.

Everyone needs a sympathetic ear when things seem not to be going well. My friends Pat O'Brien, Michael Sokoloff, and Bill Schwartz lent one, both while I was writing this book and for several decades before I began it.

Two people deserve special mention:

- Joseph E. Sacco, Ph.D., read several drafts of this book and field-tested the exercises. Joe enthusiastically found some of the darker corners of Objective-C and encouraged me to explore them. He also provided the proverbial "many valuable technical discussions," as well as many valuable non-technical discussions, during the writing of both editions of this book.
- Ekko Jennings read some of the chapters and, in addition, provided moral support and diversions, cooked dinner even when it was my turn, and just generally put up with me while I was writing. When I finished the first edition of this book I told Ekko that if I ever did anything like that again, she could hit me with a brick. To her great credit, she graciously refrained from doing so as I wrote the second edition. Thanks, Chérie.

This page intentionally left blank

About the Author

Robert Clair holds a B.A. in Physics from Oberlin College, and an M.A. and Ph.D. in Physics from the University of California, Berkeley. He has more than twenty years of experience in commercial software development, working mainly in CAD, modeling, graphics, and mobile applications. For the past eleven years he has worked primarily in Objective-C on the Mac and now on iOS. Among other programs, he has written ZeusDraw, a vector drawing program for Mac OS X, and ZeusDraw Mobile, a drawing and painting program for iOS. Robert has been the lead programmer on several large commercial iOS apps, including The Street's iPad app. He has also made additions to, and performed surgery and repair work on, various other iOS apps. Robert lives in New York City, where he is the principal of Chromatic Bytes, LLC, an independent consulting and software development company.

This page intentionally left blank

This page intentionally left blank

1

C, the Foundation of Objective-C

Objective-C is an extension of C. Most of this book concentrates on what Objective-C adds to C. But in order to program in Objective-C, you have to know the basics of C. When you do such mundane things as add two numbers together, put a comment in your code, or use an `if` statement, you do them the identical way in both C and Objective-C. The non-object part of Objective-C isn't *similar to C*, or *C-like*, it *is* C. Objective-C 2.0 is currently based on the C99 standard for C.

This chapter begins a two-chapter review of C. The review isn't a complete description of C; it covers only the basic parts of the language. Topics such as bit operators, the details of type conversion, Unicode characters, macros with arguments, and other arcana are not mentioned. It is intended as an aide-mémoire for those whose knowledge of C is rusty, or as a quick reference for those who are adept at picking up a new language from context. The following chapter continues the review of C and treats the topics of declaring variables, variable scope, and where in memory C puts variables. If you are an expert C/C++ programmer, you can probably skip this chapter. (However, a review never hurts. I learned some things in the course of writing the chapter.) If you are coming to Objective-C from a different C-like language, such as Java or C#, you should probably at least skim the material. If your only programming experience is with a scripting language, or if you are a complete beginner, you will probably find it helpful to read a book on C in parallel with this book.

> **Note**
>
> I recommend that everyone read Chapter 2, "More about C Variables." In my experience, many who should be familiar with the material it contains are not familiar with that material.

There are many books on C. The original Kernighan and Ritchie book, *The C Programming Language*, is still one of the best.[1] It is the book many people use to learn C. For a language lawyer's view of C, or to explore some of the darker corners of the language, consult *C: A Reference Manual* by Harbison and Steele.[2]

Think for a moment about how you might go about learning a new natural language. The first thing to do is look at how the language is written: Which alphabet does it use? (If it uses an alphabet at all; some languages use pictographs.) Does it read left to right, right to left, or top to bottom? Then you start learning some words. You need at least a small vocabulary to get started. As you build your vocabulary, you can start making the words into phrases, and then start combining your phrases into complete sentences. Finally, you can combine your sentences into complete paragraphs.

This review of C follows roughly the same progression. The first section looks at the structure of a C program, how C code is formatted, and the rules and conventions for naming various entities. The subsequent sections cover variables and operators, which are roughly analogous to nouns and verbs in a natural language, and how they are combined into longer expressions and statements. The last major section covers control statements. Control statements allow a program to do more interesting things than execute statements in a linear sequence. The final section of the review covers the C preprocessor, which allows you to do some programmatic editing of source files before they are sent to the compiler, and the `printf` function, which is used for character output.

The Structure of a C Program

This chapter begins by looking at some structural aspects of a C program: the `main` routine, formatting issues, comments, names and naming conventions, and file types.

`main` Routine

All C programs have a `main` routine. After the OS loads a C program, the program begins executing with the first line of code in the `main` routine. The standard form of the `main` routine is as follows:

```
int main(int argc, const char *argv[])
{
  // The code that does the work goes here
  return 0;
}
```

1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language,* Second Edition (Englewood Cliffs, NJ: Prentice Hall, 1988).

2. Samuel P. Harbison and Guy L. Steele, *C: A Reference Manual,* Fifth Edition (Upper Saddle River, NJ: Prentice Hall, 2002).

The key features are:

- The leading `int` on the first line indicates that `main` returns an integer value to the OS as a return code.
- The name `main` is required.
- The rest of the first line refers to command line arguments passed to the program from the OS. `main` receives `argc` number of arguments, stored as strings in the array `argv`. This part isn't important for the moment; just ignore it.
- All the executable code goes between a pair of curly brackets.
- The `return 0;` line indicates that a zero is passed back to the OS as a return code. In Unix systems (including Mac OS X and iOS), a return code of zero indicates "no error" and any other value means an error of some sort.

If you are not interested in processing command line arguments or returning an error code to the OS (for example, when doing the exercises in the next several chapters), you can use a simplified form of `main`:

```
int main( void )
{

}
```

The `void` indicates that this version of `main` takes no arguments. In the absence of an explicit `return` statement, a return value of zero is implied.

Formatting

Statements are terminated by a semicolon. A whitespace character (blank, tab, or newline) is required to separate names and keywords. C ignores any additional whitespace: Indenting and extra spaces have no effect on the compiled executable; they may be used freely to make your code more readable. A statement can extend over multiple lines; the following three statements are equivalent:

```
distance = rate*time;

    distance     =      rate   *   time;

distance =
    rate *
        time;
```

Comments

Comments are notations for the programmer's edification. The compiler ignores them.

C supports two styles of comments:

- Anything following two forward slashes (//) and before the end of the line is a comment. For example:

```
// This is a comment.
```

- Anything enclosed between /* and */ is also a comment:

```
/* This is the other style of comment */
```

The second type of comment may extend over multiple lines. For example:

```
/* This is
      a longer
          comment. */
```

It can be used to temporarily "comment out" blocks of code during the development process.

This style of comment cannot be nested:

```
/*  /* WRONG - won't compile */ */
```

However, the following is legal:

```
/*
    // OK - You can nest the two slash style of comment
*/
```

Variable and Function Names

Variable and function names in C consist of letters, numbers, and the underscore character (_):

- The first character must be an underscore or a letter.
- Names are case sensitive: `bandersnatch` and `Bandersnatch` are different names.
- There cannot be any whitespace in the middle of a name.

Here are some legal names:

```
j
taxesForYear2012
bananas_per_bunch
bananasPerBunch
```

These names are not legal:

```
2012YearTaxes
rock&roll
bananas per bunch
```

Naming Conventions

As a kindness to yourself and anyone else who might have to read your code, you should use descriptive names for variables and functions. `bpb` is easy to type, but it might leave you pondering when you return to it a year later; whereas `bananas_per_bunch` is self-explanatory.

Many plain C programs use the convention of separating the words in long variable and function names with underscores:

```
apples_per_basket
```

Objective-C programmers usually use *CamelCase* names for variables. CamelCase names use capital letters to mark the beginnings of subsequent words in a name:

```
applesPerBasket
```

Names beginning with an underscore are traditionally used for variables and functions that are meant to be private, or for internal use:

```
_privateVariable
_leaveMeAlone
```

However, this is a convention; C has no enforcement mechanism to keep variables or functions private.

Files

The code for a plain C program is placed in one or more files that have a *.c* filename extension:

ACProgram.c

> **Note**
>
> Mac OS X filenames are not case sensitive. The filesystem will remember the case you used to name a file, but it treats *myfile.c*, *MYFILE.c*, and *MyFile.c* as the same filename. However, filenames on iOS *are* case sensitive.

Code that uses the Objective-C objects (the material covered starting in Chapter 3, "An Introduction to Object-Oriented Programming") is placed in one or more files that have a *.m* filename extension:

AnObjectiveCProgram.m

> **Note**
>
> Because C is a proper subset of Objective-C, it's OK to put a plain C program in a *.m* file.

There are some naming conventions for files that define and implement Objective-C classes (discussed in Chapter 3), but C does not have any formal rules for the part of the name preceding the filename extension. It is silly, but not illegal, to name the file containing the code for an accounting program

```
MyFlightToRio.m
```

C programs also use *header files*. Header files usually contain various definitions that are shared by many *.c* and *.m* files. Their contents are merged into other files by using an `#include` or `#import` preprocessor directive. (See *Preprocessor* later in this chapter.) Header files have a *.h* filename extension as shown here:

```
AHeaderFile.h
```

> **Note**
>
> It is possible to mix Objective-C and C++ code in the same program. The result is called Objective-C++. Objective-C++ code must be placed in a file with a *.mm* filename extension:
>
> ```
> AnObjectiveCPlusPlusProgram.mm
> ```
>
> The topic is beyond the scope of this book.

Variables

A variable is a name for some bytes of memory in a program. When you assign a value to a variable, what you are really doing is storing that value in those bytes. Variables in a computer language are like the nouns in a natural language. They represent items or quantities in the problem space of your program.

C requires that you tell the compiler about any variables that you are going to use by declaring them. A variable declaration has the form

```
variabletype name;
```

C allows multiple variables to be declared in a single declaration:

```
variabletype name1, name2, name3;
```

A variable declaration causes the compiler to reserve storage (memory) for that variable. The value of a variable is the contents of its memory location. The next chapter describes variable declarations in more detail. It covers where variable declarations are placed, where the variables are created in memory, and the lifetimes of different classes of variables.

Integer Types

C provides the following types to hold integers: `char`, `short`, `int`, `long`, and `long long`. Table 1.1 shows the size in bytes of the integer types on 32- and 64-bit executables on Apple systems.

The `char` type is named *char* because it was originally intended to hold characters, but it is frequently used as an 8-bit integer type.

Table 1.1 **The Sizes of Integer Types on iOS and Mac OS X**

Type	32-Bit	64-Bit
char	1 byte	1 byte
short	2 bytes	2 bytes
int	4 bytes	4 bytes
long	4 bytes	8 bytes
long long	8 bytes	8 bytes

> **Note**
> iOS executables are 32-bit. Mac OS X executables can be either 32-bit or 64-bit with 64-bit as the default. 32-bit and 64-bit executables are discussed in Appendix C, "32- and 64-Bit."

An integer type can be declared to be `unsigned`:

```
unsigned char a;
unsigned short b;
unsigned int c;
unsigned long d;
unsigned long long e;
```

When used alone, `unsigned` is taken to mean `unsigned int`:

```
unsigned a;  // a is an unsigned int
```

An `unsigned` variable's bit pattern is always interpreted as a positive number. If you assign a negative quantity to an `unsigned` variable, the result is a very large positive number. This is almost always a mistake.

Floating-Point Types

C's floating-point types are `float`, `double`, and `long double`. The sizes of the floating-point types are the same in both 32- and 64-bit executables:

```
float aFloat;   // floats are 4 bytes
double aDouble; // doubles are 8 bytes
long double aLongDouble;  // long doubles are 16 bytes
```

Floating-point values are always signed.

Truth Values

Ordinary expressions are commonly used for truth values. Expressions that evaluate to zero are considered false, and expressions that evaluate to non-zero are considered true (see the following sidebar).

_Bool, bool, and BOOL

Early versions of C did not have a defined Boolean type. Ordinary expressions were (and still are) used for Boolean values (truth values). As noted in the text, an expression that evaluates to zero is considered false and one that evaluates to non-zero is considered true. A majority of C code is still written this way.

C99, the current standard for C, introduced a `_Bool` type. `_Bool` is an integer type with only two allowed values, 0 and 1. Assigning any non-zero value to a `_Bool` results in 1:

```
_Bool  b = 35;   // b is now 1
```

If you include the file *stdbool.h* in your source code files, you can use `bool` as an alias for `_Bool` and the Boolean constants `true` and `false` (`true` and `false` are just defined as 1 and 0, respectively):

```
#include <stdbool.h>
bool b = true;
```

You will rarely see either `_Bool` or `bool` in Objective-C code, because Objective-C defines its own Boolean type, `BOOL`. `BOOL` is covered in Chapter 3, "An Introduction to Object-Oriented Programming."

Initialization

Variables can be initialized when they are declared:

```
int a = 9;

int b = 2*4;

float c = 3.14159;

char d = 'a';
```

A character enclosed in single quote marks is a character constant. It is numerically equal to the encoding value of the character. Here, the variable `d` has the numeric value of 97, which is the ASCII value of the character `a`.

Pointers

A pointer is a variable whose value is a memory address. It "points" to a location in memory.

You declare a variable to be a pointer by preceding the variable name with an `*` in the declaration. The following code declares `pointerVar` to be a variable pointing to a location in memory that holds an integer:

```
int *pointerVar;
```

The unary `&` operator ("address-of" operator) is used to get the address of a variable so it can be stored in a pointer variable. The following code sets the value of the pointer variable `b` to be the address of the integer variable `a`:

```
1  int a = 9;
2
3  int *b;
4
5  b = &a;
```

Now let's take a look at that example line by line:

- Line 1 declares `a` to be an `int` variable. The compiler reserves 4 bytes of storage for `a` and initializes them with a value of 9.
- Line 3 declares `b` to be a pointer to an `int`.
- Line 5 uses the `&` operator to get the address of `a` and then assigns `a`'s address as the value of `b`.

Figure 1.1 illustrates the process. (Assume that the compiler has located `a` beginning at memory address 1048880.) The arrow in the figure shows the concept of pointing.

The unary `*` operator (called the "contents of" or "dereferencing" operator) is used to set or retrieve the contents of a memory location by using a pointer variable that points to that location. One way to think of this is to consider the expression `*pointerVar` to be an alias, another name, for whatever memory location is stored in the contents of `pointerVar`. The expression `*pointerVar` can be used to either set or retrieve the contents of that memory location. In the following code, `b` is set to the address of `a`, so `*b` becomes an alias for `a`:

```
int a;
int c;
int *b;

a = 9;

b = &a;

c = *b;  // c is now 9

*b = 10; // a is now 10
```

Pointers are used in C to reference dynamically allocated memory (see Chapter 2, "More about C Variables"). Pointers are also used to avoid copying large chunks of

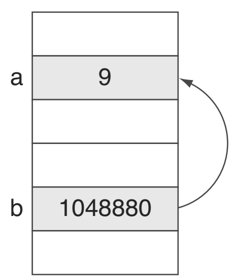


Figure 1.1 Pointer variables

memory, such as arrays and structures (discussed later in this chapter), from one part of a program to another. For example, instead of passing a large structure to a function, you pass the function a pointer to the structure. The function then uses the pointer to access the structure. As you will see later in the book, Objective-C objects are always referenced by pointer.

Generic Pointers

A variable declared as a pointer to `void` is a generic pointer:

```
void *genericPointer;
```

A generic pointer may be set to the address of any variable type:

```
int a = 9;
void *genericPointer;
genericPointer = &a;
```

However, trying to obtain a value from a generic pointer is an error because the compiler has no way of knowing how to interpret the bytes at the address indicated by the generic pointer:

```
int a = 9;
int b;
void *genericPointer;
genericPointer = &a;
b = *genericPointer;  // WRONG - won't compile
```

To obtain a value through a `void*` pointer, you must *cast* it to a pointer to a known type:

```
int a = 9;
int b;
void *genericPointer;
genericPointer = &a;
b = *((int*) genericPointer) ;  // OK - b is now 9
```

The cast operator `(int*)` forces the compiler to consider `genericPointer` to be a pointer to an integer. (See *Conversion and Casting* later in the chapter.)

C does not check to see that a pointer variable points to a valid area of memory. Incorrect use of pointers has probably caused more crashes than any other aspect of C programming.

Arrays

A C array is an ordered collection of elements of the same type. C arrays are declared by adding the number of elements in the array, enclosed in square brackets (`[]`), to the declaration, after the type and array name:

```
int a[100];
```

Individual elements of the array are accessed by placing the index of the element in `[]` after the array name:

```
a[6] = 9;
```

The index is zero-based. In the previous example, the legitimate indices run from 0 to 99. Access to C arrays is not bounds checked on either end. C will blithely let you do the following:

```
int a[100];
a[200] = 25;
a[-100] = 30;
```

Using an index outside of the array's bounds lets you trash memory belonging to other variables, resulting in either crashes or corrupted data. Taking advantage of this lack of checking is one of the pillars of mischievous malware.

The bracket notation is just a nicer syntax for pointer arithmetic. The name of an array, without the array brackets, is a pointer variable pointing to the beginning of the array. These two lines are completely equivalent:

```
a[6] = 9;

*(a + 6) = 9;
```

When compiling an expression using pointer arithmetic, the compiler takes into account the size of the type the pointer is pointing to. If `a` is an array of `int`, the expression `*(a + 2)` refers to the contents of the 4 bytes (one `int` worth) of memory at an address 8 bytes (two `int`) beyond the beginning of the array `a`. However, if `a` is an array of `char`, the expression `*(a + 2)` refers to the contents of 1 byte (one `char` worth) of memory at an address 2 bytes (two `char`) beyond the beginning of the array `a`.

Multidimensional Arrays

Multidimensional arrays are declared as follows:

```
int b[4][10];
```

Multidimensional arrays are stored linearly in memory by rows. Here, `b[0][0]` is the first element, `b[0][1]` is the second element, and `b[1][0]` is the eleventh element.

Using pointer notation:

```
b[i][j]
```

may be written as

```
*(b + i*10 + j)
```

Strings

A C string is a one-dimensional array of bytes (type `char`) terminated by a zero byte. A constant C string is coded by placing the characters of the string between double quote marks (""):

```
"A constant string"
```

When the compiler creates a constant string in memory, it automatically adds the zero byte at the end. But if you declare an array of `char` that will be used to hold a string, you must remember to include the zero byte when deciding how much space you need. The following line of code copies the five characters of the constant string `"Hello"` and its terminating zero byte to the array `aString`:

```
char aString[6] = "Hello";
```

As with any other array, arrays representing strings are not bounds checked. Overrunning string buffers used for program input is a favorite trick of hackers.

A variable of type `char*` can be initialized to point to a constant string. You can set such a variable to point at a different string, but you can't use it to modify a constant string:

```
char *aString = "Hello";
aString = "World";
aString[4] = 'q';  // WRONG - causes a crash, "World" is a constant
```

The first line points `aString` at the constant string `"Hello"`. The second line changes `aString` to point at the constant string `"World"`. The third line causes a crash, because constant strings are stored in a region of protected, read-only memory.

Structures

A structure is a collection of related variables that can be referred to as a single entity. The following is an example of a structure declaration:

```
struct dailyTemperatures
  {
    float high;
    float low;
    int   year;
    int   dayOfYear;
};
```

The individual variables in a structure are called *member variables* or just *members* for short. The name following the keyword `struct` is a *structure tag*. A structure tag identifies the structure. It can be used to declare variables typed to the structure:

```
struct dailyTemperatures today;

struct dailyTemperatures *todayPtr;
```

In the preceding example, `today` is a `dailyTemperatures` structure, whereas `todayPtr` is a pointer to a `dailyTemperatures` structure.

The dot operator (`.`) is used to access individual members of a structure from a structure variable. The pointer operator (`->`) is used to access structure members from a variable that is a pointer to a structure:

```
todayPtr = &today;

today.high = 68.0;

todayPtr->high = 68.0;
```

The last two statements do the same thing.

Structures can have other structures as members. The previous example could have been written like this:

```
struct hiLow
{
    float high;
    float low;
};

struct dailyTemperatures
{
    struct hiLow tempExtremes;
    int   year;
    int   dayOfYear;
};
```

Setting the high temperature for `today` would then look like this:

```
struct dailyTemperatures today;
today.tempExtremes.high = 68.0;
```

> **Note**
>
> The compiler is free to insert padding into a structure to force structure members to be aligned on a particular boundary in memory. You shouldn't try to access structure members by calculating their offset from the beginning of the structure or do anything else that depends on the structure's binary layout.

typedef

The `typedef` declaration provides a means for creating aliases for variable types:

```
typedef float Temperature;
```

`Temperature` can now be used to declare variables, just as if it were one of the built-in types:

```
Temperature high, low;
```

`typedef`s just provide alternate names for variable types. Here, `high` and `low` are still floats. The term *typedef* is often used as a verb when talking about C code, as in "Temperature is typedef'd to float."

Enumeration Constants

An `enum` statement lets you define a set of integer constants:

```
enum woodwind { oboe, flute, clarinet, bassoon };
```

The result of the previous statement is that `oboe`, `flute`, `clarinet`, and `bassoon` are constants with values of 0, 1, 2, and 3, respectively.

If you don't like going in order from zero, you can assign the values of the constant yourself. Any constant without an assignment has a value one higher than the previous constant:

```
enum woodwind { oboe=100, flute=150, clarinet, bassoon=200 };
```

The preceding statement makes `oboe`, `flute`, `clarinet`, and `bassoon` equal to 100, 150, 151, and 200, respectively.

The name after the keyword `enum` is called an *enumeration tag*. Enumeration tags are optional. Enumeration tags can be used to declare variables:

```
enum woodwind soloist;
soloist = oboe;
```

Enumerations are useful for defining multiple constants, and for helping to make your code self-documenting, but they aren't distinct types and they don't receive much support from the compiler. The declaration `enum woodwind soloist;` shows your intent that `soloist` should be restricted to one of `oboe`, `flute`, `clarinet`, or `bassoon`, but unfortunately, the compiler does nothing to enforce the restriction. The compiler considers `soloist` to be an `int`, and it lets you assign any integer value to `soloist` without generating a warning:

```
enum woodwind { oboe, flute, clarinet, bassoon };
enum woodwind soloist;
soloist  = 5280;  // No complaint from the compiler!
```

> **Note**
> You can't have a variable and an enumeration constant with the same name.

Operators

Operators are like verbs. They cause things to happen to your variables.

Arithmetic Operators

C has the usual binary operators `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division, respectively.

> **Note**
>
> If both operands to the division operator (/) are integer types, C does integer division. Integer division truncates the result of doing the division. The value of 7/2 is 3.
>
> If at least one of the operands is a floating-point type, C promotes any integers in the division expression to float and performs floating-point division. The values of 7.0/2, 7/2.0, and 7.0/2.0 are all 3.5.

Remainder Operator

The remainder or *modulus* operator (`%`) calculates the remainder from an integer division. The result of the following expression is 1:

```
int a = 7;
int b = 3;
int c = a%b;  // c is now 1
```

Both operands of the remainder operator must be integer types.

Increment and Decrement Operators

C provides operators for incrementing and decrementing variables:

```
a++;

++a;
```

Both lines add 1 to the value of `a`. However, there is a difference between the two expressions when they are used as a part of a larger expression. The prefix version, `++a`, increments the value of `a` *before* any other evaluation takes place. It is the incremented value that is used in the rest of the expression. The postfix version, `a++`, increments the value of `a` after other evaluations take place. The original value is used in the rest of the expression. This is illustrated by the following example:

```
int a = 9;
int b;
b = a++; // postfix increment

int c = 9;
int d;
d = ++c; // prefix increment
```

The postfix version of the operator increments the variable after its initial value has been used in the rest of the expression. After the code has executed in this example, the value of `b` is 9 and the value of `a` is 10. The prefix version of the operator increments the variable's value before it is used in the rest of the expression. In the example, the value of both `c` and `d` is 10.

The decrement operators `a--` and `--a` behave in a similar manner.

Code that depends on the difference between the prefix and postfix versions of the operator is likely to be confusing to anyone but its creator.

Precedence

Is the following expression equal to 18 or 22?

```
2 * 7 + 4
```

The answer seems ambiguous because it depends on whether you do the addition first or the multiplication first. C resolves the ambiguity by making a rule that it does multiplication and division before it does addition and subtraction; so the value of the expression is 18. The technical way of saying this is that multiplication and division have higher *precedence* than addition and subtraction.

If you need to do the addition first, you can specify that by using parentheses:

```
2 * (7 + 4)
```

The compiler will respect your request and arrange to do the addition before the multiplication.

> **Note**
>
> C defines a complicated table of precedence for all its operators (see http://en.wikipedia.org/wiki/Order_of_operations). But specifying the exact order of evaluation that you want by using parentheses is much easier than trying to remember operator precedences.

Negation

The unary minus sign (-) changes an arithmetic value to its negative:

```
int a = 9;
int b;
b  = -a;  // b is now -9
```

Comparisons

C also provides operators for comparisons. The value of a comparison is a truth value. The following expressions evaluate to `1` if they are true and `0` if they are false:

```
a > b // true, if a is greater than b

a < b // true, if a is less than b

a >= b // true, if a is greater than or equal to b

a <= b // true, if a is less than or equal to b

a == b // true, if a is equal to b

a != b // true, if a is not equal to b
```

> **Note**
>
> As with any computer language, testing for floating-point equality is risky because of rounding errors, and such a comparison is likely to give an incorrect result.

Logical Operators

The logical operators for AND and OR have the following form:

```
expression1 && expression2  // Logical AND operator

expression1 || expression2  // Logical OR operator
```

C uses *short circuit evaluation*. Expressions are evaluated from left to right, and evaluation stops as soon as the truth value for the entire expression can be deduced. If *expression1* in an AND expression evaluates to false, the value of the entire expression is false and *expression2* is not evaluated. Similarly, if *expression1* in an OR expression evaluates to true, the entire expression is true and *expression2* is not evaluated. Short circuit evaluation has interesting consequences if the second expression has any side effects. In the following example, if `b` is greater than or equal to `a`, the function `CheckSomething()`is not called (`if` statements are covered later in this chapter):

```
if ( b < a && CheckSomething() )
  {
    ...
  }
```

Logical Negation

The unary exclamation point `(!)` is the logical negation operator. After the following line of code is executed, `a` has the value `0` if *expression* is true (non-zero), and the value `1` if *expression* is false (zero):

```
a = ! expression;
```

Assignment Operators

C provides the basic assignment operator:

```
a = b;
```

`a` is assigned the value of `b`. Of course, `a` must be something that is capable of being assigned to. Entities that you can assign to are called *lvalues* (because they can appear on the *left* side of the assignment operators). Here are some examples of lvalues:

```
/* set up */
float a;
float b[100]
float *c;
struct dailyTemperatures today;
struct dailyTemperatures *todayPtr;
c = &a;
```

```
todayPtr = &today;

/* legal lvalues */
a = 76;
b[0] = 76;
*c = 76;
today.high = 76;
todayPtr->high = 76;
```

Some things are not *lvalues*. You can't assign to an array name, the return value of a function, or any expression that does not refer to a memory location:

```
float a[100];
int x;

a = 76; // WRONG
x*x = 76; // WRONG
GetTodaysHigh() = 76; // WRONG
```

Conversion and Casting

If the two sides of an assignment are of different variable types, the type of the right side is converted to the type of the left side. Conversions from shorter types to longer types don't present a problem. Going the other way, from a longer type to a shorter type, or converting between a floating-point type and an integer type, requires care. Such a conversion can cause loss of significant figures, truncation, or complete nonsense. For example:

```
int a = 14;
float b;
b = a;  // OK, b is now 14.0
        // A float can hold approximately 7 significant figures

float c = 12.5;
int d;
d = c;  // Truncation, d is now 12

char e = 99;
int f;
f = e;  // OK, f is now 99

int g = 333;
char h;
h = g;  // Nonsense, h is now 77
        // The largest number a signed char can hold is 127

int h = 123456789;
float i = h;  // loss of precision
              // A float cannot keep 9 significant figures
```

You can force the compiler to convert the value of a variable to a different type by using a *cast*. In the last line of the following example, the `(float)` casts force the compiler to convert `a` and `b` to `float` and do a floating-point division:

```
int a = 6;
int b = 4;
float c, d;

c = a / b;  // c is equal to 1.0 because integer division truncates

d = (float)a / (float)b; // Floating-point division, d is equal to 1.5
```

You can cast pointers from pointer to one type to pointer to another. Casting pointers can be a risky operation with the potential to trash your memory, but it is the only way to dereference a pointer passed to you typed as `void*`. Successfully casting a pointer requires that you understand what type of entity the pointer is "really" pointing to.

Other Assignment Operators

C also has shorthand operators that combine arithmetic and assignment:

```
a += b;

a -= b;

a *= b;

a /= b;
```

These are equivalent to the following, respectively:

```
a = a + b;

a = a - b;

a = a * b;

a = a / b;
```

Expressions and Statements

Expressions and statements in C are the rough equivalent of phrases and sentences in a natural language.

Expressions

The simplest expressions are just single constants or variables:

```
14
bananasPerBunch
```

Every expression has a *value*. The value of an expression that is a constant is just the constant itself: The value of `14` is 14. The value of a variable expression is whatever value the variable is holding: The value of `bananasPerBunch` is whatever value it was given when it was last set by initialization or assignment.

Expressions can be combined to create other expressions. The following are also expressions:

```
j + 14
a < b
distance = rate * time
```

The value of an arithmetic or logical expression is just whatever you would get by doing the arithmetic or logic. The value of an assignment expression is the value given to the variable that is the target of the assignment.

Function calls are also expressions:

```
SomeFunction()
```

The value of a function call expression is the return value of the function.

Evaluating Expressions

When the compiler encounters an expression, it creates binary code to evaluate the expression and find its value. For primitive expressions, there is nothing to do: Their values are just what they are. For more complicated expressions, the compiler generates binary code that performs the specified arithmetic, logic, function calls, and assignments.

Evaluating an expression can cause *side effects*. The most common side effects are the change in the value of a variable due to an assignment, or the execution of the code in a function due to a function call.

Expressions are used for their value in various control constructs to determine the flow of a program (see *Program Flow*). In other situations, expressions may be evaluated only for the side effects caused by evaluating them. Typically, the point of an assignment expression is that the assignment takes place. In a few situations, both the value and the side effect are important.

Statements

When you add a semicolon (`;`) to the end of an expression, it becomes a *statement*. This is similar to adding a period to a phrase to make a sentence in a natural language. A statement is the code equivalent of a complete thought. A statement is finished executing when all of the machine language instructions that result from the compilation of the statement have been executed, and all of the changes to any memory locations the statement affects have been completed.

Compound Statements

You can use a sequence of statements, enclosed by a pair of curly brackets, any place where you can use a single statement:

```
{
  timeDelta = time2 - time1;
  distanceDelta = distance2 - distance1;
  averageSpeed = distanceDelta / timeDelta;
}
```

There is no semicolon after the closing bracket. A group like this is called a *compound statement* or a *block*. Compound statements are very commonly used with the control statements covered in the next sections of the chapter.

> **Note**
>
> The use of the word *block* as a synonym for *compound statement* is pervasive in the C literature and dates back to the beginnings of C. Unfortunately, Apple has adopted the name *block* for its addition of closures to C (see Chapter 17, "Blocks"). To avoid confusion, the rest of this book uses the slightly more awkward name *compound statement*.

Program Flow

The statements in a program are executed in sequence, except when directed to do otherwise by a `for`, `while`, `do-while`, `if`, `switch`, or `goto` statement or a function call.

- An `if` statement conditionally executes code depending on the truth value of an expression.
- The `for`, `while`, and `do-while` statements are used to form loops. In a loop, the same statement or group of statements is executed repeatedly until a condition is met.
- A `switch` statement chooses a set of statements to execute based on the arithmetic value of an integer expression.
- A `goto` statement is an unconditional jump to a labeled statement.
- A function call is a jump to the code in the function's body. When the function returns, the program executes from the point after the function call.

These control statements are covered in more detail in the following sections.

> **Note**
>
> As you read the next sections, remember that every place it says *statement*, you can use a compound statement.

`if`

An **`if`** statement conditionally executes code depending on the truth value of an expression. It has the following form:

```
if ( expression )

  statement
```

If *`expression`* evaluates to true (non-zero), *`statement`* is executed; otherwise, execution continues with the next statement after the `if` statement. An `if` statement may be extended by adding an **`else`** section:

```
if ( expression )

  statement1

else

  statement2
```

If *`expression`* evaluates to true (non-zero), *`statement1`* is executed; otherwise, *`statement2`* is executed.

An `if` statement may also be extended by adding `else if` sections, as shown here:

```
if ( expression1 )

  statement1

else if ( expression2 )

  statement2

else if ( expression3 )

  statement3

...

else

  statementN
```

The expressions are evaluated in sequence. When an expression evaluates to non-zero, the corresponding statement is executed and execution continues with the next statement following the `if` statement. If the expressions are all false, the statement following the `else` clause is executed. (As with a simple `if` statement, the `else` clause is optional and may be omitted.)

Conditional Expression

A conditional expression is made up of three sub-expressions and has the following form:

```
expression1 ? expression2 : expression3
```

When a conditional expression is evaluated, *expression1* is evaluated for its truth value. If it is true, *expression2* is evaluated and the value of the entire expression is the value of *expression2*. *expression3* is not evaluated.

If *expression1* evaluates to false, *expression3* is evaluated and the value of the conditional expression is the value of *expression3*. *expression2* is not evaluated.

A conditional expression is often used as shorthand for a simple `if` statement. For example:

```
a = ( b > 0 ) ? c : d;
```

is equivalent to

```
if ( b > 0 )

  a = c;

else

  a = d;
```

while

The `while` statement is used to form loops as follows:

```
while ( expression ) statement
```

When the `while` statement is executed, *expression* is evaluated. If it evaluates to true, *statement* is executed and the condition is evaluated again. This sequence is repeated until *expression* evaluates to false, at which point execution continues with the next statement after the `while`.

You will occasionally see this construction:

```
while (1)
  {
    ...
  }
```

Since the constant `1` evaluates to true, the preceding is an infinite loop from the `while`'s point of view. Presumably, something in the body of the loop checks for a condition and breaks out of the loop when that condition is met.

do-while

The **do-while** statement is similar to the **while**, with the difference that the test comes after the statement rather than before:

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

One consequence of this is that *statement* is always executed once, independent of the value of *expression*. Situations where the program logic dictates that a loop body be executed at least once, even if the condition is false, are uncommon. As a consequence, **do-while** statements are rarely encountered in practice.

for

The **for** statement is the most general looping construct. It has the following form:

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

When a **for** statement is executed, the following sequence occurs:

1. *expression1* is evaluated once before the loop begins.
2. *expression2* is evaluated for its truth value.
3. If *expression2* is true, *statement* is executed; otherwise, the loop ends and execution continues with the next statement after the loop.
4. *expression3* is evaluated.
5. Steps 2, 3, and 4 are repeated until *expression2* becomes false.

expression1 and *expression3* are evaluated only for their side effects. Their values are discarded. They are typically used to initialize and increment a loop counter variable:

```
int j;

for (j=0; j < 10; j++)
  {
    // Something that needs doing 10 times
  }
```

> **Note**
>
> You can also declare the loop variable inside a **for** statement:
>
> ```
> for ( int j=0; j < 10; j++ )
>   {
>   }
> ```
>
> When you declare the loop variable inside a **for** statement, it is valid only inside the loop. It is undefined once the loop exits. If you are breaking out of a loop on some condition (see the next section) and you want to examine the loop variable to see what its value was when the condition was met, you must declare the loop variable outside the **for** statement as shown in the earlier example.

Any of the expressions may be omitted (the semicolons must remain). If *expression2* is omitted, the loop is an infinite loop, similar to `while( 1 )`:

```
int i;

for (i=0; ; i++)
  {
    ...
    // Check something and exit if the condition is met
  }
```

> **Note**
>
> When you use a loop to iterate over the elements of an array, remember that array indices go from zero to one less than the number of elements in the array:
>
> ```
> int j;
> int a[25];
>
> for (j=0; j < 25; j++)
>   {
>     // Do something with a[j]
>   }
> ```
>
> Writing the `for` statement in the preceding example as
>
> ```
> for (j=1; j <= 25; j++)
> ```
>
> is a common mistake.

break

The `break` statement is used to break out of a loop or a `switch` statement:

```
int j;

for (j=0;  j < 100; j++)
 {
    ...

    if ( someConditionMet ) break; //Execution continues after the loop
 }
```

Execution continues with the next statement after the enclosing `while`, `do`, `for`, or `switch` statement. When used inside nested loops, `break` only breaks out of the innermost loop. Coding a `break` statement that is not enclosed by a loop or a switch causes a compiler error:

```
error: break statement not within loop or switch
```

continue

`continue` is used inside a `while`, `do-while`, or `for` loop to abandon execution of the current loop iteration. For example:

```
int j;

for (j=0;  j < 100; j++)
 {
    ...

    if ( doneWithIteration ) continue; // Skip to the next iteration
    ...
}
```

When the `continue` statement is executed, control passes to the next iteration of the loop. In a `while` or `do-while` loop, the control expression is evaluated for the next iteration. In a `for` loop, the iteration (third) expression is evaluated and then the control (second) expression is evaluated. Coding a `continue` statement that is not enclosed by a loop causes a compiler error.

Comma Expression

A comma expression consists of two or more expressions separated by commas:

```
expression1, expression2, ..., expressionN
```

The expressions are evaluated in order from left to right, and the value of the entire expression is the value of the right-most sub-expression.

The principal use of the comma operator is to initialize and update multiple loop variables in a `for` statement. As the loop in the following example iterates, `j` goes from `0` to `MAX-1` and `k` goes from `MAX-1` to `0`:

```
int j, k;

for (j=0, k=MAX-1; j < MAX; j++, k--)
  {
    // Do something
  }
```

When a comma expression is used like this in a `for` loop, only the side effects of evaluating the sub-expressions are important. The value of the comma expression is discarded. In the preceding example a comma expression is also used to increment `j` and decrement `k` after each pass through a `for` loop.

switch

A `switch` branches to different statements based on the value of an integer expression. The form of a `switch` statement is shown here:

```
switch ( integer_expression )
  {
     case value1:
       statement
       break;

     case value2:
       statement
       break;

     ...

     default:
       statement
       break;
}
```

In a slight inconsistency with the rest of C, each case may have multiple statements without the requirement of a compound statement.

`value1`, `value2`, `...` must be either integers, character constants, or constant expressions that evaluate to an integer. (In other words, they must be reducible to an integer at compile time.) Duplicate cases with the same integer are not allowed.

When a `switch` statement is executed, *`integer_expression`* is evaluated and the switch compares the result with the integer case labels. If a match is found, execution jumps to the statement after the matching case label. Execution continues in sequence until either a `break` statement or the end of the switch is encountered. A `break` causes the execution to jump out to the first statement following the switch.

A `break` statement is not required after a case. If it is omitted, execution falls through to the following case. If you see the `break` omitted in existing code, it can be either a mistake (it is an easy one to make) or intentional (if the coder wanted a case and the following case to execute the same code).

If *`integer_expression`* doesn't match any of the case labels, execution jumps to the statement following the optional `default:` label, if one is present. If there is no match and no `default:`, the `switch` does nothing, and execution continues with the first statement after the switch.

> **Note**
>
> If you use an `enum` variable as the argument for a `switch` statement, do not supply a case for each value of the `enum`, and do not supply a `default:` case, some compilers will complain with a warning. The Clang (LLVM) compiler currently used by Apple is among those that complain.

goto

C provides a `goto` statement:

```
goto label;
```

When the `goto` is executed, control is unconditionally transferred to the statement marked with `label:`, as here:

```
label: statement
```

- Labels are not executable statements; they just mark a point in the code.
- The rules for naming labels are the same as the rules for naming variables and functions.
- Labels always end with a colon.

Using `goto` statements with abandon can lead to tangled, confusing code (often referred to as *spaghetti code*). The usual boilerplate advice is "Don't use `goto` statements." Despite this, `goto` statements are useful in certain situations, such as breaking out of nested loops (a `break` statement only breaks out of the innermost loop):

```
int i, j;

for (i=0; i < MAX_I; i++)
  for (j=0; j < MAX_J; j++)
     {
        ...
       if ( finished ) goto moreStuff;

     }

moreStuff:  statement     // more statements
```

> **Note**
> Whether to use `goto` statements is one of the longest-running debates in computer science. For a summary of the debate, see http://david.tribble.com/text/goto.html.

Functions

Functions have the following form:

```
returnType functionName( arg1Type arg1, ..., argNType argN )
{

 statements

}
```

An example of a simple function looks like this:

```
float salesTax( float purchasePrice, float taxRate )
{
  float tax = purchasePrice * taxRate;
  return tax;
}
```

> **Note**
>
> Variables that are declared inside a function are called *local variables*. Local variables are generally valid only from the point they are declared until the end of the function. The range of code for which a variable is valid is called the variable's *scope*. Variable scope is treated in detail in the next chapter.

A function is called by coding the function name followed by a parenthesized list of expressions, one for each of the function's arguments. Each expression type must match the type declared for the corresponding function argument. The following example shows a simple function call:

```
float carPrice = 20000.00;
float stateTaxRate = 0.05;

float carSalesTax = salesTax( carPrice, stateTaxRate );
```

When the line with the function call is executed, control jumps to the first statement in the function body. Execution continues until a `return` statement is encountered or the end of the function is reached. Execution then returns to the calling context. The value of the function expression in the calling context is the value set by the `return` statement.

> **Note**
>
> Functions are not required to have any arguments or to return a value. Functions that do not return a value are typed `void`:
>
> ```
> void FunctionThatReturnsNothing( int arg1 )
> ```
>
> You may omit the `return` statement from a function that does not return a value.
>
> Functions that don't take any arguments are indicated by using empty parentheses for the argument list:
>
> ```
> int FunctionWithNoArguments()
> ```

Functions are sometimes executed solely for their side effects. This function prints out the sales tax but changes nothing in the program's state:

```
void printSalesTax ( float purchasePrice, float taxRate )
{
  float tax = purchasePrice * taxRate;

  printf( "The sales tax is: %f.2\n", tax );

}
```

C functions are *call by value*. When a function is called, the expressions in the argument list of the calling statement are evaluated and their *values* are passed to the function. A function cannot directly change the value of any of the variables in the calling context. This function has no effect on anything in the calling context:

```
void salesTax( float purchasePrice, float taxRate, float carSalesTax )
{
  // Changes the local copy of carSalesTax but not the value of
  // the variable in the calling context

     carSalesTax = purchasePrice * taxRate;
     return;
}
```

To change the value of a variable in the calling context, you must pass a pointer to the variable and use that pointer to manipulate the variable's value:

```
void salesTax( float purchasePrice, float taxRate, float *carSalesTax)
{
  *carSalesTax = purchasePrice * taxRate; // this will work

  return;
}
```

> **Note**
>
> The preceding example is still call by value. The *value* of a pointer to a variable in the calling context is passed to the function. The function then uses that pointer (which it doesn't alter) to set the value of the variable it points to.

Declaring Functions

When you call a function, the compiler needs to know the types of the function's arguments and return value. It uses this information to set up the communication between the function and its caller. If the code for the function comes before the function call (in the source code file), you don't have to do anything else. If the function is coded after the function call or in a different file, you must declare the function before you use it.

A function declaration repeats the first line of the function, with a semicolon added at the end:

```
void printSalesTax ( float purchasePrice, float taxRate );
```

It is a common practice to put function declarations in a header file. The header file is then included (see the next section) in any file that uses the function.

> **Note**
>
> Forgetting to declare functions can lead to insidious errors. If you call a function that is coded in another file, and you don't declare the function, neither the compiler nor the linker will complain. But the function will receive garbage for any floating-point argument and return garbage if the function's return type is floating-point. In the absence of a declaration the compiler assumes that argument types and the return type are integers. It then interprets the bit patterns of floating-point arguments or return values as integers, resulting in (wildly) erroneous results.

Preprocessor

When C (and Objective-C) code files are compiled, they are first sent through an initial program, called the *preprocessor*, before being sent to the compiler proper. Lines that begin with a # character are directives to the preprocessor. Using preprocessor directives, you can:

- Import the text of a file into one or more other files at a specified point.
- Create defined constants.
- Conditionally compile code (compile or omit statement blocks depending on a condition).

Including Files

The following line:

```
#include "HeaderFile.h"
```

causes the preprocessor to insert the text of the file *HeaderFile.h* into the file being processed at the point of the `#include` line. The effect is the same as if you had used a text editor to copy and paste the text from *HeaderFile.h* into the file being processed.

If the included filename is enclosed in quotation marks `("")`:

```
#include "HeaderFile.h"
```

the preprocessor will look for *HeaderFile.h* in the same directory as the file being compiled, then in a list of locations that you can supply as arguments to the compiler, and finally in a series of system locations.

If the included file is enclosed in angle brackets (<>):

```
#include <HeaderFile.h>
```

the preprocessor will look for the included file only in the standard system locations.

> **Note**
>
> In Objective-C, `#include` is superseded by `#import`, which produces the same result, except that it prevents the named file from being imported more than once. If the preprocessor encounters further `#import` directives for the same header file while working on a given file, the additional `#import` directives are ignored.

`#define`

`#define` is used for textual replacement. The most common use of `#define` is to define constants, such as

```
#define MAX_VOLUME 11
```

The preprocessor will replace every occurrence of `MAX_VOLUME` in the file being compiled with an 11. A `#define` can be continued on multiple lines by placing a backslash (\) at the end of all but the last line in the definition.

> **Note**
>
> If you do this, the `\` must be the last thing on the line. Following the `\` with something else (such as a comment beginning with `//`) results in an error.

A frequently used pattern is to place the `#define` in a header file, which is then included by various source files. You can then change the value of the constant in all the source files by changing the single definition in the header file. The traditional C naming convention for defined constants is to use all capital letters. A traditional Apple naming convention is to begin the constant name with a `k` and CamelCase the rest of the name:

```
#define kMaximumVolume 11
```

You will encounter both styles, sometimes in the same code.

Conditional Compilation

The preprocessor allows for conditional compilation:

```
#if condition

  statements

#else

  otherStatements

#endif
```

Here, *`condition`* must be a constant expression that can be evaluated for a truth value at compile time. If *`condition`* evaluates to true (non-zero), *`statements`* are compiled, but *`otherStatements`* are not. If *`condition`* is false, *`statements`* are skipped and *`otherStatements`* are compiled.

The `#endif` is required, but the `#else` and the alternative code are optional. A conditional compilation block can also begin with an `#ifdef` directive:

```
#ifdef name

  statements

#endif
```

The behavior is the same as the previous example, except that the truth value of `#ifdef` is determined by whether *`name`* has been `#define`'d.

One use of `#if` is to easily remove and replace blocks of code during debugging:

```
#if 1
  statements
#endif
```

By changing the `1` to a `0`, *`statements`* can be temporarily left out for a test. They can then be replaced by changing the `0` back to a `1`.

`#if` and `#ifdef` directives can be nested, as shown here:

```
#if 0
#if 1
statements
#endif
#endif
```

In the preceding example, the compiler ignores all the code, including the other compiler directives, between the `#if 0` and its matching `#endif`. *`statements`* are not compiled.

If you need to disable and re-enable multiple statement blocks, you can code each block like this:

```
#if _DEBUG

statements

#endif
```

The defined constant `_DEBUG` can be added or removed in a header file or by using a `–D` flag in the compile command.

printf

Input and output (I/O) are not a part of the C language. Character and binary I/O are handled by functions in the C standard I/O library.

> **Note**
>
> The standard I/O library is one of a set of libraries of functions that is provided with every C environment.

To use the functions in the standard I/O library, you must include the library's header file in your program:

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

The only function covered here is **`printf`**, which prints a formatted string to your terminal window (or to the Xcode console window if you are using Xcode). The `printf` function takes a variable number of arguments. The first argument to `printf` is a *format string*. Any remaining arguments are quantities that are printed out in a manner specified by the format string:

```
printf( formatString, argument1, argument2, ... argumentN );
```

The format string consists of ordinary characters and *conversion specifiers*:

- Ordinary characters (not `%`) in the format string are sent unchanged to the output.
- Conversion specifiers begin with a percent sign (`%`). The letter following the `%` indicates the type of argument the specifier expects.
- Each conversion specification consumes, in order, one of the arguments following the format string. The argument is converted to characters that represent the value of the argument, and the characters are sent to the output.

The only conversion specifiers used in this book are `%d` for `char` and `int`, `%f` for `float` and `double`, and `%s` for C strings. C strings are typed as `char*`.

Here is a simple example:

```
int myInt = 9;
float myFloat = 3.145926;
char* myString = "a C string";

printf( "This is an integer: %d, a float: %f, and a string: %s.\n",
    myInt, myFloat, myString );
```

> **Note**
>
> The `\n` is the *newline character*. It advances the output so that any subsequent output appears on the next line.

The result of the preceding example is

```
This is an Integer: 9, a float: 3.145926, and a string: a C string.
```

If the number of arguments following the format string doesn't match the number of conversion specifications, `printf` ignores the excess arguments or prints garbage for the excess specifications.

> **Note**
>
> This book uses `printf` only for logging and debugging non-object variables, not for the output of a polished program, so this section presents only a cursory look at format strings and conversion specifiers.
>
> `printf` handles a large number of types, and it provides very fine control over the appearance of the output. A complete discussion of the available types of conversion specifications and how to control the details of formatting is available via the Unix *man* command. To see them, type the following at a terminal window:
>
> ```
> man 3 printf
> ```

Note

The Objective-C Foundation framework provides `NSLog`, another logging function. It is similar to `printf`, but it adds the capability to print out object variables. It also adds the program name, the date, and the time in hours, minutes, seconds, and milliseconds to the output. This additional information can be visually distracting if all you want to know is the value of a variable or two, so this book uses `printf` in some cases where `NSLog`'s additional capability is not required. `NSLog` is covered in Chapter 3, "An Introduction to Object-Oriented Programming."

Command Line Compiling and Debugging

When you write programs for Mac OS X or iOS, you should write, compile, and debug your programs using Xcode, Apple's integrated development environment (IDE). You'll learn how to set up a simple Xcode project in Chapter 4, "Your First Objective-C Program." However, for the simple C programs required in the exercises in this chapter and the next chapter, you may find it easier to write the programs in your favorite text editor and then compile and run them from a command line.

Compilers and Debuggers

Historically, Apple has used the open-source GNU compiler, gcc, for building iOS and OS X programs. However, in the past several years they have transitioned to using compilers from the open-source LLVM (Low Level Virtual Machine) project. LLVM is not a single compiler; it is a set of modules that can be used to build compilers, debuggers, and related tools. In the first part of the transition Apple used a compiler called llvm-gcc-4.2 that combined the front end from gcc 4.2 with the LLVM code generator and optimizer. The current compiler is the Clang compiler, which combines a new unified parser for C, Objective C, C++, and Objective C++ with the LLVM code generator and optimizer.

Note that Apple refers to the Clang compiler as "LLVM Compiler *N*," where *N* is the current version number—4.0 as of Xcode 4.4.

The Clang compiler has many advantages over llvm-gcc-4.2:

- Clang provides much more informative error messages when you make a mistake.
- Clang is faster than gcc.
- Certain newer features of Objective-C, such as automatic reference counting (ARC), are available only by using the Clang compiler.

Clang is the default compiler in the current version of Xcode. Apple has announced that llvm-gcc-4.2 is "frozen" (no new features or bug fixes) as of Xcode 4.4 and will be removed in a future version of Xcode.

The LLVM project also includes LLDB, a new debugger.

You can find more information on Clang, LLDB, and the LLVM project on the LLVM website: http://llvm.org.

To compile from the command line, you will need:

1. A terminal window. You can use the Terminal app (*/Applications/Utilities/Terminal*) that comes with Mac OS X. If you are coming from another Unix environment, and you are used to xterms, you may prefer to download and use iTerm, an OS X native terminal application that behaves similarly to an xterm (http://iterm.sourceforge.net/).
2. A text editor. Mac OS X comes with both *vi* and *emacs*, or you can use a different editor if you have one.
3. The Apple Developer tools. You can get the current version of Xcode and the Developer tools from the OS X App Store (they're free).

The first step is to check whether the command line tools are installed on your system. At the command prompt type

```
which clang
```

If the response is

```
/usr/bin/clang
```

your command line tools are already installed. If the response is

```
clang: Command not found.
```

the command line tools are not installed on your system and you must use Xcode to install them. To install the command line tools:

1. Open Xcode.
2. Choose *Xcode > Preferences...* from the menu.
3. When the *Preferences* panel opens, click the *Downloads* tab and then click on *Components*.
4. Finally, click the *Install* button for *Command Line Tools*.
5. When the installation is finished, you may close Xcode.

You are now ready to compile. If your source code file is named *MyCProgram.c*, you can compile it by typing the following at the command prompt:

```
clang -o MyCProgram MyCProgram.c
```

The `-o` flag allows you to give the compiler a name for your final executable. If the compiler complains that you have made a mistake or two, go back to fix them, and then try again. When your program compiles successfully, you can run it by typing the executable name at the command prompt:

```
MyCProgram
```

If you want to debug your program using gdb, the GNU debugger, or LLDB, you must use the `-g` flag when you compile:

```
clang -g -o MyCProgram MyCProgram.c
```

The `-g` flag causes `clang` to attach debugging information to the final executable.

To use gdb to debug a program, type `gdb` followed by the executable name:

```
gdb MyCProgram
```

Similarly, to use lldb you type `lldb` followed by the executable name:

```
lldb MyCProgram
```

Documentation for gdb is available at the GNU website, www.gnu.org/software/gdb/, or from Apple at http://developer.apple.com/mac/library/#documentation/DeveloperTools/gdb/gdb/gdb_toc.html. In addition, there are many websites with instructions for using gdb. Search for "gdb tutorial."

You can learn more about LLDB by watching the Apple video at http://devimages.apple.com/llvm/videos/LLDB_Debugging_Infrastructure.mov or by going to the LLDB website, http://lldb.llvm.org/.

Summary

This chapter has been a review of the basic parts of the C language. The review continues in Chapter 2, "More about C Variables," which covers the memory layout of a C program, declaring variables, variable scope and lifetimes, and dynamic allocation of memory. Chapter 3, "An Introduction to Object-Oriented Programming," begins the real business of this book: object-oriented programming and the object part of Objective-C.

Exercises

1. Write a function that returns the average of two floating-point numbers. Write a small program to test your function and log the output. Next, put the function in a separate source file but "forget" to declare the function in the file that has your main routine. What happens? Now add the function declaration to the file with your main program and verify that the declaration fixes the problem.
2. Write another averaging function, but this time try to pass the result back in one of the function's arguments. Your function should be declared like this:

   ```
   void average( float a, float b, float average )
   ```

   Write a small test program and verify that your function doesn't work. You can't affect a variable in the calling context by setting the value of a function parameter.

Now change the function and its call to pass a pointer to a variable in the calling context. Verify that the function can use the pointer to modify a variable in the calling context.

3. Assume that you have a function, `int flipCoin()`, that randomly returns a `1` to represent heads or a `0` to represent tails. Explain how the following code fragment works:

```
int flipResult;
if ( flipResult = flipCoin() )
  printf( "Heads is represented by %d\n", flipResult );
else
  printf( "Tails is represented by %d\n", flipResult );
```

As you will see in Chapter 6, "Classes and Objects," an `if` condition similar to the one in the preceding example is used in the course of initializing an Objective-C object.

4. An identity matrix is a square array of numbers with ones on the diagonal (the elements where the row number equals the column number) and zero everywhere else. The 2×2 identity matrix looks like this:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Write a program that calculates and stores the 4×4 identity matrix. When your program is finished calculating the matrix, it should output the result as a nicely formatted square array.

5. Fibonacci numbers (http://en.wikipedia.org/wiki/Fibonacci_number) are a numerical sequence that appears in many places in nature and in mathematics. The first two Fibonacci numbers are defined to be 0 and 1. The nth Fibonacci number is the sum of the previous two Fibonacci numbers:

$$F_n = F_{n-1} + F_{n-2}$$

Write a program that calculates and stores the first 20 Fibonacci numbers. After calculating the numbers, your program should output them, one on a line, along with their index. The output lines should be something like this:

```
Fibonacci Number 2 is: 1
```

Use a `#define` to control the number of Fibonacci numbers your program produces, so that it can be easily changed.

6. Rewrite your program from the previous exercise to use a `while` loop instead of a `for` loop.

7. What if you are asked to calculate the first 75 Fibonacci numbers? If you are using `int`s to store the numbers, there is a problem. You will find that the 47th Fibonacci number is too big to fit in an `int`. How can you fix this?

8. Judging by the number of tip calculators available in the iOS App Store, a substantial fragment of the population has forgotten how to multiply. Help out those who can't multiply but can't afford an iPhone. Write a program that calculates a 15% tip on all check amounts between $10 and $50. (For brevity, go by $0.50 increments.) Show both the check amount and the tip.
9. Now make the tip calculator look more professional. Add a column for 20% tips (Objective-C programmers eat in classy joints). Place the proper headers on each column and use a pair of nested loops so that you can output a blank line after every $10 increment.

   Using the conversion specification `%.2f` instead of `%f` will limit the check and tip output to two decimal places. Using `%%` in the format string will cause `printf` to output a single `%` character.
10. Define a structure that holds a rectangle. Do this by defining a structure that holds the coordinates of a point and another structure that represents a size by holding a width and a height. Your rectangle structure should have a point that represents the lower-left corner of the rectangle and a size. (The Cocoa frameworks define structures like these, but make your own for now.)
11. One of the basic tenets of efficient computer graphics is "Don't draw if you don't have to draw." Graphics programs commonly keep a bounding rectangle for each graphic object. When it is time to draw the graphic on the screen, the program compares the graphic's bounding rectangle with a rectangle representing the window. If there is no overlap between the rectangles, the program can skip trying to draw the graphic. Overall, this is usually a win; comparing rectangles is much cheaper than drawing graphics.

    Write a function that takes two rectangle structure arguments. (Use the structures that you defined in the previous exercise.) Your function should return `1` if there is a non-zero overlap between the two rectangles, and `0` otherwise. Write a test program that creates some rectangles and verify that your function works.

Index

A

C

K

L

N

Q

R

U

V