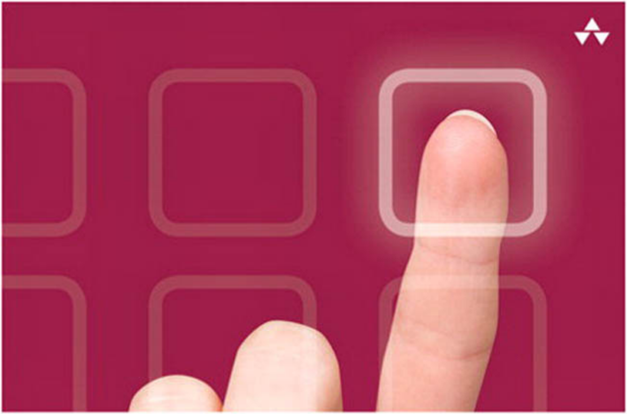

LEARNING OBJECTIVE-C 2.0

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

ROBERT CLAIR

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@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Clair, Robert, 1950–
Learning Objective-C 2.0 : a hands-on guide to Objective-C for Mac and iOS developers / Robert Clair.
p. cm.
Includes index.
ISBN-13: 978-0-321-71138-0 (pbk. : alk. paper)
ISBN-10: 0-321-71138-6 (pbk. : alk. paper)
1. Objective-C (Computer program language) 2. Object-oriented programming (Computer science) 3. Macintosh (Computer)—Programming. 4. iPhone (Smartphone)—Programming. I. Title.
QA76.73.0115C58 2011
005.1'17—dc22

2010019360

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-71138-0
ISBN-10: 0-321-71138-6
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2010

Editor-in-Chief
Karen Gettman

Senior Acquisitions Editor
Chuck Toporek

Managing Editor
John Fuller

Project Editor
Anna Popick

Copy Editor
Kelli Brooks

Indexer
Jack Lewis

Proofreader
Debbie Liehs

Technical Reviewers
Joachim Bean
Cory Bohon
Andy Lee
Matt Long
Joseph E. Sacco, Ph.D.
Scott D. Yelich

Editorial Assistant
Romny French

Compositor
Rob Mauhar

Cover Designer
Chuti Prasertsith

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 other object-oriented languages, particularly C++. The additions that Objective-C makes to C can be listed on a page or two.

In the Apple world, Objective-C does not work alone. It is used in conjunction with two class libraries called *frameworks*. 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. 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.

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 Mac OS X. The NeXT Objective-C libraries became the basis of Cocoa.

This book concentrates on the Objective-C language. It will not teach you how to write Cocoa programs or make you an expert Xcode user. 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 if you first acquire a good understanding of the language on which Cocoa is based.

Who Should Read This Book

This book is intended for programmers who want to learn Objective-C in order to write programs for Mac OS X or iOS. (iOS is used for the iPhone, the iPod touch, and the iPad.) Although 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 iPhone app's outer layer and user interface must be written in Objective-C.

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.

What You Need to Know

This book assumes a working knowledge of C. Objective-C is an extension of C; the 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 2 and 3 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.

New in Objective-C 2.0

If you already know some Objective-C and want to skip to the parts of the language that are new in the 2.0 version, they are covered in these chapters:

- *Fast Enumeration* (Chapter 10) provides a simple (and fast) way to iterate over a collection of objects.
- *Declared properties* (Chapter 12) provide an easy way to specify an object's instance variables and to have the compiler create methods to access those variables for you.
- *Garbage collection* (Chapter 15) adds automatic memory management to Objective-C.
- *Blocks* (Chapter 16) let you define function-like objects that carry their context with them.

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, and provides an introduction to

1. http://developer.apple.com/mac/library/documentation/UserExperience/Conceptual/AppleHIGuidelines
2. www.gnustep.org

the Foundation framework. The final section of the book covers the two forms of memory management used in Objective-C, and Objective-C 2.0's newly added Blocks feature.

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 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. They list the instance variables and methods that make up a given type of object and provide the code that implements those methods. An object is more tangible; it is a region of memory, similar to a C struct, which 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 executing an 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 an instance 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 new 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 that you consider private. The chapter ends with a discussion of Objective-C security issues.
- Chapter 12, "Properties," introduces Objective-C 2.0's new *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 for you, 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.) The Java concept of an interface was borrowed from Objective-C protocols.

Part III: Advanced Concepts

Objective-C provides two different systems for managing object memory: *reference counting* and automatic *garbage collection*.

- Chapter 14, "Reference Counting," covers Objective-C's traditional 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 15, "Garbage Collection," describes Objective-C 2.0's new automatic garbage collection system. Using garbage collection, a separate thread called the *garbage collector*, is responsible for determining which objects are no longer needed and freeing them. Garbage collection relieves you of most memory management chores.
- Chapter 16, "Blocks," discusses Objective-C 2.0's new 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.

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 Apple's ongoing move to 64-bit computing.
- Appendix D, "Runtimes, Old and New," describes the difference between the older "legacy" Objective-C runtime used for 32-bit OS X programs and the newer "modern" runtime used for 64-bit Objective-C programs running on OS X 10.5 or later, and for iOS programs.
- Appendix E, "Resources for Objective-C," lists books and websites that have useful information for Objective-C developers.

Compile Time and Run Time

There are two times that are significant when you create programs: compile time, when your source code is translated into machine language and linked together to form an executable program, and run time (also called execution time), when the executable program is run as a process on some computer. One of the characteristics that distinguishes Objective-C from other common languages, especially C++, is that Objective-C is a very dynamic language. "Dynamic" here means that decisions that other languages make at compile time are postponed to run time in Objective-C. The most prominent example of this is Objective-C's messaging system. The section of code that a program executes when it evaluates a message expression (the equivalent of a method call in other languages) is determined at run time.

Postponing decisions until run time has many advantages, as you will see as you read this book, but it has one important drawback. It limits the amount of checking that the compiler can do. When you code in Objective-C, some errors, which would be caught at the compile stage in other languages, only become apparent at run time.

About Memory Management

As noted earlier, Objective-C 2.0 offers you the choice between using a manual reference counting system or automatic garbage collection for managing object memory. With the exception of Chapter 15, which covers Objective-C 2.0's garbage collection system, this book uses reference counting from the beginning in all of its examples. It then provides a complete treatment of reference counting in Chapter 14.

The primary reason for this is that garbage collection is not available on iOS. If you want to write programs for the iPhone, the iPod touch, or the iPad, you must learn Objective-C's reference counting system.

Judging from the contents of various Objective-C and Cocoa mailing lists, reference counting is probably the single greatest source of confusion among people learning Objective-C. But if you learn its rules early and apply them uniformly, you will discover that reference counting isn't really difficult.

If, at a later time, you want to use garbage collection for a project, the transition should be relatively painless. Although there are some architectural issues that you need to be aware of when moving from reference counting to garbage collection (which are covered in Chapter 15), much of using garbage collection simply consists of not doing things that you have to do when using reference counting.

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 Code Listings

The examples in this book are a mixture of unnumbered and numbered code listings.

- **Unnumbered Code Listings**

  These are primarily short snippets of code that are referenced in the text that immediately precedes or follows the example.

- **Numbered Code Listings**

  The numbered code listings have captions and are numbered by chapter number and their order of appearance in the chapter (e.g., Listing 4.1 or Listing 8.3). These are primarily larger examples that are referred to in text later in the chapter or in the exercises following the chapter.

In both cases, examples that require a line-by-line explanation are given line numbers so that the explanatory text can refer to a specific line in the code.

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.

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 them from a command line, as shown before the exercises in Chapter 2, or by using a simple Xcode project, as shown in Chapter 4.

We'd Like to Hear from You

You can visit our website and register this book at www.informit.com/title/9780321711380.

Be sure to visit the book's website for convenient access to any updates, downloads, or errata that might be available for this book.

As the 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.

When you write, please be sure to include this book's title and the name of the author, as well as your name, phone, and/or email address. I will carefully review your comments and share them with the author and others who have worked on this book.

Email: chuck.toporek@pearson.com
Mail: Chuck Toporek
Senior Acquisitions Editor, Addison-Wesley
Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116
USA

For more information about our books or conferences, see our website at www.informit.com.

16

Blocks

Blocks provide a way to package up some executable code and a context (various variables) as a single entity so they can be handed off for execution at a later time or on a different thread. In other languages, blocks or similar constructs are sometimes called *closures* or *anonymous functions*. Blocks are an Apple-supplied extension to C, Objective-C 2.0, and C++. Apple has submitted blocks to the C standards working group as a proposed extension to C. At the time of this writing, blocks are only available on Mac OS X Snow Leopard (v 10.6 and on iOS 4). They are not available on earlier versions of Mac OS X or iPhone iOS.

> **Note**
>
> You can use blocks on iPhone OS 3.x and on Mac OS X Leopard (v 10.5) if you install *Plausible Blocks* (PLBlocks). Plausible Blocks, a reverse-engineered port from Apple-released open-source Darwin OS code, provides the compilers and runtime required to use blocks. You can obtain Plausible Blocks from http://code.google.com/p/plblocks/.

Handing off a package of work is useful in many situations, but one of the main driving forces behind the adoption of blocks is Apple's new *Grand Central Dispatch* (GCD) feature. GCD is designed to make concurrency easier to program and more efficient to execute. Essentially, GCD is a thread pool that is managed for you by the operating system. The idea behind GCD is that the operating system has a global view of all the processes running on your Mac, and allocates resources (CPU, GPU, and RAM) as needed to make things run more efficiently. GCD can make better decisions than a user space program can about the number of threads to use and when to schedule them for execution. You use blocks to submit units of work for GCD to execute.

> **Note**
>
> GCD provides a C interface for submitting blocks. Cocoa provides a higher-level interface to GCD through the classes `NSOperationQueue`, `NSBLockOperation`, and `NSInvocationOperation`.
>
> `NSInvocationOperation` allows you to submit units of work as `NSInvocation` objects instead of blocks, but as you will see in the section `NSInvocation`, `NSInvocation` objects are somewhat difficult to set up. Blocks are much easier to use.

This chapter is an introduction to blocks. You will learn how to define a block, how a block has access to variables in its surrounding context, how to use a block in your own code, and about the somewhat tricky topic of memory management for blocks. The chapter also explores some pitfalls that can befall an unwary user of blocks.

Before looking at blocks in detail, the chapter takes a pair of detours and looks at two earlier ways of packaging up functionality: *function pointers* and the Foundation class `NSInvocation`.

Function Pointers

When the compiler encounters a function call, it inserts a jump instruction to the code that performs the function. (A jump instruction causes the program execution to jump to the specified code instead of executing the line of code directly after the jump instruction.) To return, the function executes a jump instruction back to the line of code following the original function call. In a normal function call, the landing point of the jump instruction (and hence the function that is called) is static. It is determined at compile time. But a function call can be made dynamic through the use of a *function pointer*.

The following line declares `myFunctionPtr` as a pointer to a function that takes two `int`s as arguments and returns an `int`:

```
int (*myFunctionPtr) (int, int);
```

Figure 16.1 shows the anatomy of a function pointer.

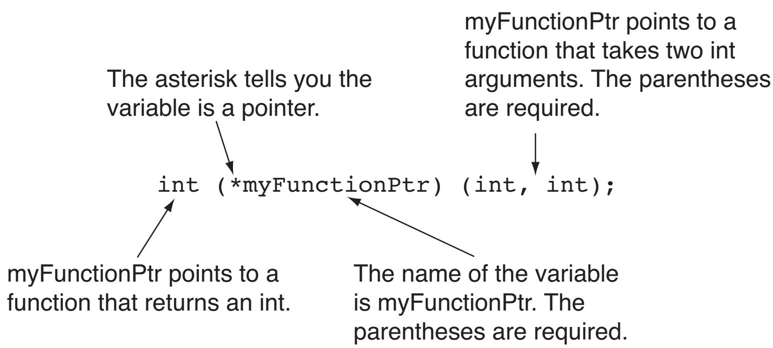

Figure 16.1 The anatomy of a function pointer

The general form of a function pointer is:

```
return_type (*name)(list of argument types);
```

Function pointers are a low point in C syntax. Instead of reading left-to-right or right-to-left, they read from the inside out. More complicated function pointer declarations can quickly turn into puzzles, as you will see in Exercise 16.1.

A Syntax Quirk in Objective-C

The "from the inside out" declaration style of function pointers doesn't fit with Objective-C's syntax for method arguments. Recall that Objective-C requires the type for a method argument to be enclosed in parentheses before the argument name. This conflict is resolved in favor of the syntax for method arguments.

When declaring a method argument that is a function pointer, the name comes outside. For example, a pointer to a function that has no arguments or return value is normally declared as follows:

```
void (*functionPtr)(void);
```

A function that takes a function pointer of preceding type as an argument is declared like this:

```
void functionWithFPArg(void (*fp)(void));
```

But a method with the same argument type is declared like this:

```
-(void) methodWithFFPArg:(float(*)(float)) fp;
```

Putting the name of the function pointer last in a function pointer declaration only works when declaring the type of a method argument. Putting the name last results in a compiler error in any other situation.

You can also declare arrays of function pointers. The following line declares `fpArray` as an array of 10 pointers to functions. Each function takes a single argument, a pointer to a `float`, and returns `void`:

```
void (*fpArray[10])(float*);
```

A function pointer can point to a function that has another function pointer as an argument or a return value:

```
void (*(*complicatedFunctionPointer)(void))(void);
```

`complicatedFunctionPointer` is a pointer to a function that takes no arguments and returns a pointer to a function that takes no arguments and returns `void`.

Declarations like the preceding one are ugly, but you can make your code cleaner by hiding the ugliness with a `typedef`:

```
typedef void (*(*complicatedFunctionPointer)(void))(void);
complicatedFunctionPointer fp;
```

Calling a Function with a Function Pointer

The following example shows how to assign a function to a function pointer and how to call the function using the function pointer:

```
void logInt( int n )
  {
    NSLog("The integer is: %d", n);
  }
```

```
void (*myFunctionPtr)(int); // Declare a function pointer

myFunctionPtr = logInt;     // Make it point to logInt
myFunctionPtr( 5 );         // Execute the function through the pointer
```

To make the function pointer refer to a function, you simply assign it the name of the function. The function must be defined or visible by a forward declaration at the point it is assigned.

To call a function through a function pointer, you simply add the arguments, encased in parentheses, to the function pointer. A function call through a function pointer is just like a normal function call except that you use the name of the function pointer variable instead of the function name, as shown in the previous code snippet.

> **Note**
>
> There is no need to use the address of operator (`&`) or the dereferencing operator (`*`) with function pointers. The compiler knows which names are functions or function pointers and which names are regular variables. However, if you would like to use them, you may, as shown here:
>
> ```
> myFunctionPtr = &logInt; // The same as myFunctionPtr = logInt;
>
> (*myFunctionPtr)( 5 ) // The same as myFunctionPtr( 5 );
> ```
>
> The compiler doesn't care which form you use.

Using Function Pointers

One of the primary uses of function pointers is for *callbacks*. Callbacks are used in situations where you have a function or method that is going to do some work for you, but you would like the opportunity to insert your own code somewhere in the process. To do this, you pass the working function or method a pointer to a function containing the code you want executed. At the appropriate time, the working function or method will call your function for you.

For example, `NSMutableArray` provides the following method for use in custom sorting:

```
- (void)sortUsingFunction:
           (NSInteger (*)(id, id, void *))compare
                  context:(void *)context
```

When you invoke `sortUsingFunction:context:`, the method sorts the contents of the receiver. To perform the sort, `sortUsingFunction:context:` must examine pairs of array elements and decide how they are ordered. To make these decisions, `sortUsingFunction:context:` calls the `compare` function that you passed in by pointer when the method was invoked.

The `compare` function must look at the two objects it receives and decide (based on whatever criterion you require) whether they are ordered `NSOrderedAscending`, `NSOrderedSame`, or `NSOrderedDescending`.

> **Note**
>
> `NSOrderedAscending`, `NSOrderedSame`, and `NSOrderedDescending` are integer constants defined by the Foundation framework.

`sortUsingFunction:context:` also passes `compare` the `void*` pointer that it received as its `context` argument. This is a pure pass-through; `sortUsingFunction:context:` doesn't look at or modify `context`. `context` may be `NULL` if `compare` doesn't require any additional information.

Listing 16.1 sorts an array containing some `NSNumber` objects. The address of a `BOOL` is passed in to control the direction of a numerical sort.

Listing 16.1 **ArraySortWithFunctionPointer.m**

```
#import <Foundation/Foundation.h>

NSInteger numericalSortFn( id obj1, id obj2, void* ascendingFlag )
{
  int value1 = [obj1 intValue];
  int value2 = [obj2 intValue];
  if ( value1 == value2 )  return NSOrderedSame;
  if ( *(BOOL*) ascendingFlag )
    {
      return  ( value1 < value2 ) ?
          NSOrderedAscending : NSOrderedDescending;
    }
  else
    {
      return  ( value1 < value2 )
       ? NSOrderedDescending : NSOrderedAscending;
    }
}

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  // Put some number NSNumber objects in an array
  NSMutableArray *numberArray = [[NSMutableArray alloc] initWithCapacity: 5];
  [numberArray addObject: [NSNumber numberWithInt: 77]];
  [numberArray addObject: [NSNumber numberWithInt: 59]];
  [numberArray addObject: [NSNumber numberWithInt: 86]];
```

```
  [numberArray addObject: [NSNumber numberWithInt: 68]];
  [numberArray addObject: [NSNumber numberWithInt: 51]];

  NSLog( @"Before sort: %@", [numberArray description] );

  // This flag controls the sort direction.
  // Change it to NO to sort in descending order.
    BOOL ascending = YES;

  // Sort the array
  [numberArray sortUsingFunction: numericalSortFn context: &ascending];

  NSLog( @"After sort: %@", [numberArray description] );

  [numberArray release];
  [pool drain];
  return 0;
}
```

Notice:

- `ascendingFlag` is passed in as `void*`. It must be cast as `BOOL*` before it can be dereferenced to get the `BOOL` value.
- The name of a function, in this case `numericalSortFn`, can serve as a properly typed pointer to that function. Here, it is used as the argument when invoking `sortUsingFunction:context:` without defining a separate function pointer variable.

The Trouble with Function Pointers

There is one large inconvenience with using function pointers as callbacks or in any situation where you are trying to hand off some code for execution by another part of the program or another thread. Any context that the function requires must be packed up and submitted as a separate variable or variables.

Most designs using callbacks use the pattern shown in Listing 16.1. The function or method that is passed the callback function accepts a blind pointer as an additional argument and then passes that pointer back to the callback function, as shown on Figure 16.2. This is only a minor inconvenience when the context is a single variable as in the preceding example. However, if your function requires a more complicated context, you must create and load a custom structure to hold the context and then pass the pointer to that structure as the context variable. As an alternative, you could package the context variables in an `NSDictionary` and then pass the dictionary as the context. Either way is awkward if the context involves many variables.

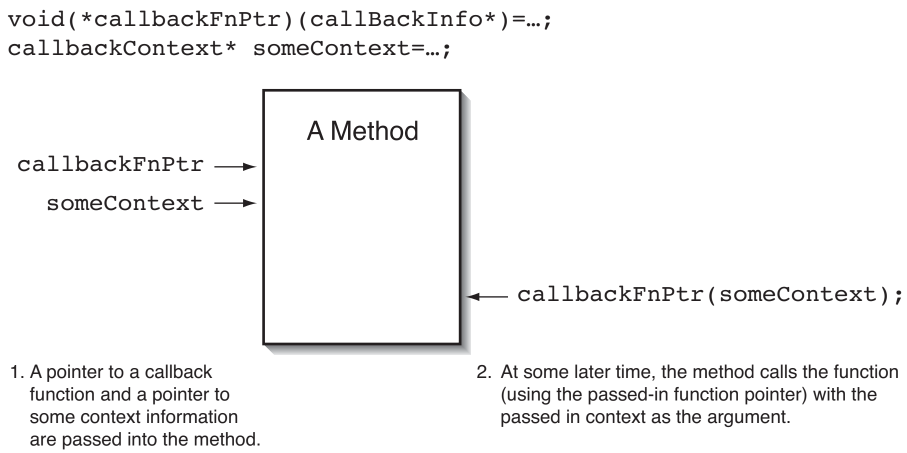

Figure 16.2 Passing a callback function to a method

NSInvocation

An `NSInvocation` object takes an Objective-C message and turns it into an object. The invocation object stores the message's receiver (called the *target* in invocation-speak), selector, and arguments. An invocation object can be saved for later execution or passed on to another part of your code. When you send the invocation object an `invoke` message, the invocation sends the target object a message using the stored selector and arguments.

As an example, consider a `LineGraphic` class with a method `drawWithColor:width:` that draws a line with a specified color and line width:

```
LineGraphic *graphic = ...
[graphic drawWithColor: [NSColor redColor] width: 2.0];
```

Listing 16.2 shows how to turn the preceding message into an invocation.

Listing 16.2 Constructing an `NSInvocation`

```
LineGraphic *graphic = ...

NSInvocation *drawInvocation =
    [NSInvocation  invocationWithMethodSignature:
        [graphic methodSignatureForSelector:
          @selector(drawWithColor:width:)]];

[drawInvocation setTarget: graphic];
[drawInvocation setSelector: @selector(drawWithColor:width:)];

[drawInvocation retainArguments];
```

```
NSColor *color = [NSColor redColor];
float linewidth = 2.0;

[drawInvocation setArgument: &color atIndex: 2];
[drawInvocation setArgument: &linewidth atIndex: 3];
```

To set up an invocation, `NSInvocation` needs to know the return type of the message being encapsulated and the types of the message's arguments. The message's selector is just a name and doesn't carry any type information, so you must obtain the type information by calling the target's `methodSignatureForSelector:`. This is a method that all classes inherit from `NSObject`. It returns an `NSMethodSignature` object, which is an encoded representation of the selector's return type and argument types. Finally, you pass the returned `NSMethodSignature` to `NSInvocation`'s `invocationWithMethodSignature:` class method to create the invocation.

> **Note**
>
> In Listing 16.2, the call to `methodSignatureForSelector:` is nested inside the call to `invocationWithMethodSignature:` so there is no explicit `NSMethodSignature` variable.

Next, you set the invocation's target and selector with `setTarget:` and `setSelector:`.

An `NSInvocation` does not retain its target or any of its arguments by default. If you are going to save an invocation for future execution, you should ask the invocation to retain its target and arguments by sending the invocation a `retainArguments` message. This prevents the target and arguments from being released before the invocation is invoked.

The arguments for the encapsulated message are set with the `setArgument:atIndex:` method:

- You pass the *address* of the variable being used for the argument, not the variable itself. You can't use a value directly in `setArgument:atIndex:` message:

  ```
  [drawInvocation setArgument: 2.0 atIndex: 3];  // WRONG!
  ```

- If the selector has an argument that is not an object (an argument that is a primitive C type such as `int` or `float`), you may use the address of the primitive type directly. There is no need to wrap the `width` argument in an `NSNumber` object.
- The arguments are identified by their position in the message. Notice that indices start at `2`. Indices `0` and `1` are reserved for the hidden method arguments `self` and `_cmd`. (For an explanation of a method's hidden arguments, see Chapter 5, "Messaging.")

Now that you have created `drawInvocation`, you can store it or hand it off to other code. When you are ready to draw the line, you simply execute the following line of code:

```
[drawInvocation invoke];
```

An invocation like the preceding one could be used as part of a display list scheme in a drawing program. Each invocation, stored in an array, encapsulates the message required to draw an element in the final scene. When you are ready to draw the scene, you loop through the array of invocations and invoke each one in turn:

```
NSMutableArray *displayList = ...
for NSInvocation invocation in displayList
{
  [invocation invoke];
}
```

Two additional points:

- An `NSInvocation` can be invoked any number of times.
- It is possible to encapsulate a message with a return value in an `NSInvocation`. To get the return value, send the invocation a `getReturnValue:` message, as illustrated here:

  ```
  double result;
  [invocationReturningDouble getReturnValue: &result];
  ```

  The argument to `getReturnValue:` must be a pointer to a variable of the same type as the invocation's return type. If you send a `getReturnValue:` message to an invocation object before it has been sent an `invoke` message, the result is undefined. The value is garbage and may cause a program crash if you attempt to it for anything.

`NSInvocation` objects are used in the Cocoa framework to schedule an operation to be performed after a time interval (`NSTimer`), and in the Cocoa undo mechanism (`NSUndoManager`). `NSInvocation` objects solve one of the problems of using function pointers; they carry at least some of their context with them in the form of the arguments to the encapsulated message. That said, `NSInvocation` objects have a major drawback—as you have seen in Listing 16.2, they are difficult to construct.

Blocks

Blocks are similar in many ways to functions, with the important exception that blocks have direct access to variables in their surrounding context. "Direct access" means that a block can use variables declared in its surrounding context even though those variables are not passed into the block through an argument list. Blocks are declared as follows:

```
^(argument_list){ body };
```

Blocks begin with a caret (`^`), followed by an argument list enclosed in parentheses and one or more statements enclosed in curly brackets. This expression is called a *block literal*, and is defined as follows:

- Blocks can return a value by using a `return` statement in the body.
- You don't have to specify the return type; the compiler determines the return type by looking at the body of the block.

- If the block doesn't have arguments, the argument list is written as `(void)`.
- Block literals don't have a name; that is why they are sometimes called *anonymous functions*.

The following is a simple block example that takes an integer, doubles it, and returns the doubled value:

```
^(int n){ return n*2; };
```

You can call a block literal directly by appending values for its arguments surrounded in parentheses:

```
int j = ^(int n){ return n*2; }( 9 ); // j is now 18
```

This works, but it is a bit silly; you could get the same result by simply coding the following:

```
int j = 2 * 9;
```

In normal use, a block literal is assigned to a variable typed as pointer to a block, or used as an argument to a function or a method that takes a pointer to block argument.

Block Pointers

A variable that is a block pointer (a pointer to a block) is declared as follows:

```
return_type (^name)(list of argument types);
```

This should look familiar; it has exactly the same form as the declaration of a function pointer, except that the `*` has been replaced with a `^`.

> **Note**
>
> Don't refer to a variable that holds a block pointer as a block variable. As you will see later in the chapter, the term *block variable* is reserved for a different entity.

The following example illustrates how to declare a block pointer, assign a block to it, and then call the block through the block pointer:

```
int (^doubler)(int);  // doubler is typed as a pointer to a block

doubler = ^(int n){ return n*2; };

int j = doubler( 9 );  // j is now 18
```

You can use block pointers as arguments to a function or a method:

```
// someFunction is a function that takes a block pointer as an argument

void someFunction( int (^blockArg)(int) );
```

```
int (^doubler)(int)= ^(int n){ return n*2; };

someFunction( doubler );
```

You can also use a block literal directly in a function call or an Objective-C message expression:

```
void someFunction( int (^blockArg)(int) );

someFunction( ^(int n){ return n*2; } );
```

> **Note**
>
> Objective-C method declarations have the same quirk with block pointers as they do with function pointers: When declaring a method that takes a block pointer argument, the name comes outside the type declaration:
>
> ```
> - (void) doSomethingWithBlockPointer:
>           (float (^)(float)) blockPointer;
> ```

Access to Variables

A block has:

- Read-only access to automatic variables visible in its enclosing scope[1]
- Read/write access to static variables declared in a function and to external variables
- Read/write access to special variables declared as *block variables*

Here is a simple example of a block accessing a local variable in its enclosing scope:

```
int j = 10;

int (^blockPtr)(int) = ^(int n){ return j+n; };

int k = blockPtr( 5 );  // k is now 15
```

The value that a block uses for a local variable from its context is the value that local variable has when the flow of execution passes over the block literal, as shown here:

```
int j = 10;

int (^blockPtr)(int) = ^(int n){ return j+n; };

j = 20;

int k = blockPtr ( 5 );  // k is 15, not 25 as you might expect
```

1. The scope of various classes of variables is discussed in Chapter 2, *More About C Variables*.

In the preceding code:

- Line 1: The local variable `j` is set to 10.
- Line 3: The block `blockPtr` is defined. `blockPtr` has access to the local variable `j`. The value of `j` that `blockPtr` uses is bound to the value that `j` has when the program execution passes over this line. In effect, `blockPtr` now has a private copy of `j` that is set to 10.
- Line 5: To show that `blockPtr`'s value of `j` was bound in Line 3, `j` is reset to 20.
- Line 7: The `blockPtr` is evaluated with an argument of 5 resulting in a return value of 10 (the value of `j` at the time Line 3 is executed) + 5 (the argument) = 15.

A block's access to local variables is read-only. The following code, which attempts to set the value of `j` from inside `blockPtr`, results in a compiler error because `blockPtr`'s access to the local variable `j` is read-only:

```
int j = 10;

void (^blockPtr)(void) = ^(void){ j = 20; };
```

> **Note**
>
> If a local variable holds a pointer to an object, a block cannot change the variable to point to a different object. But the object that the variable points to can still be modified by the block:
>
> ```
> NSMutableArray *localArray =  ...
>
> void (^shortenArray)(void) = ^(void){ [localArray removeLastObject]; };
>
> // Removes the last object in localArray
>
> shortenArray();
> ```

The compiler gives blocks access to `static` and external variables by pointer. The value the block sees for this type of variable is the value the variable has when the block is executed, not when the block is defined:

```
static int j = 10;

int (^blockPtr)(int) = ^(int n){ return j+n; };

j = 20;

int k = blockPtr ( 5 );  // k is 25
```

Block Variables

Variables declared with the new type modifier `__block` are called *block variables*:

```
__block int integerBlockVariable;
```

Block variables are visible to, and are shared and mutable by, any block defined in their scope. For example:

```
1 __block int j = 10;
2
3 void (^blockPtr_1)(void) = ^(void){ j += 15; };
4 void (^blockPtr_2)(void) = ^(void){ j += 25; };
5
6 blockPtr_1(); // j is now 25
7 blockPtr_2(); // j is now 50
```

In the preceding code:

- Line 1: `j` is declared as a `__block` variable and set to 10.
- Line 3: `blockPtr_1` is defined. Because `j` has been declared as a `__block` variable, `blockPtr_1` is permitted to set the value of `j`.
- Line 4: Similarly, `blockPtr_2` is permitted to set the value of `j`. Both `blockPtr_1` and `blockPtr_2` share read-write access to the variable `j`.
- Line 6: `blockPtr_1` is evaluated, incrementing `j` by 15, resulting in a value of 25 for `j`.
- Line 7: `blockPtr_2` is evaluated, incrementing `j` by 25, resulting in a value of 50 for `j`.

Block variables start out on the stack like any other automatic variable. But if you copy a block that references a block variable, the block variable is moved from the stack to the heap along with the block (see the section *Copying Blocks*).

> **Note**
>
> The term *block variable* refers to a variable that is declared with the `__block` modifier, as described in the preceding paragraphs. Don't confuse "block variable" and "variable that holds a pointer to a block." A block variable is not a block.

Blocks Are Stack Based

When you define a block literal inside a function or a method, the compiler creates a structure on the stack that holds the values of any local variables the block references, the addresses of read/write variables it references, and a pointer to block's executable code.

> **Note**
>
> The block structure is created on the stack, but the block's executable code is not on the stack. It is in the text portion of the program along with all the other executable code.

Blocks have the same lifetime as automatic variables. When the block literal goes out of scope, it is undefined, just like an automatic variable that has gone out of scope. Scope for a block literal is defined in the same way as scope for an automatic variable (see

Chapter 2, "More About C Variables"): If a block literal is defined inside a compound statement (between a pair of curly brackets), the block literal goes out of scope when the program execution leaves the compound statement. If a block literal is defined in a function, it goes out of scope when the function returns. The following code is incorrect:

```
int (^doubler)(int);
{
   ...
   doubler = ^(int n){ return n*2; };
   ...
}
...
int j = doubler( 9 );  // WRONG! Bug!
```

In the preceding example, `j` is undefined. At the point where `doubler(9)` is executed, the block that the `doubler` variable points to has gone out of scope and the block may have been destroyed.

> **Note**
>
> If you try the preceding example, it may appear to work correctly. `j` may very well be set to 18. But this would be an accident of the way the complier has arranged the code in this instance. After the block is out of scope, the compiler is free to reuse the space the block occupied in the stack frame. If the compiler has reused the space, the result of trying to execute the out-of-scope block would be an incorrect value of `j` or, more likely, a crash.

Global Blocks

You can also assign a block literal to a file-scope block pointer variable:

```
#import <Foundation/Foundation.h>

void (^logObject)(id) =
  ^(id obj){ NSLog( @"Object Description: %@", [obj description]); };

// logObject( someObj ) may be used anywhere in the file.
```

The compiler creates global-scope blocks in low memory like any other file-scope variable. Global blocks never go out of scope.

Blocks Are Objective-C Objects

It may seem surprising, but blocks are also Objective-C objects. A newly created block is the only example of an Objective-C object that is created on the stack. Blocks are instances of one of several private subclasses of `NSObject`. Apple doesn't provide the header for the block classes so you can't subclass them or do much of anything with them in an Objective-C sense except send them `copy`, `retain`, `release`, and `autorelease` messages. Copying and memory management for blocks are covered in the next sections.

Copying Blocks

One of the main uses of blocks is to pass a chunk of work (some code plus some context) out of the current scope for processing at a later time. Passing a block to a function or method that you call is safe (as long as that function or method is going to execute on the same thread). But what happens if you want to pass a block to a different thread or pass a block out of the current scope as a return value? When the current function or method returns, its stack frame is destroyed. Any blocks that were defined in its scope become invalid.

To preserve a block, you must copy it. When you copy a block, the copy is created on the heap. The heap-based copy is then safe to return up the stack to the calling function or pass off to another thread.

If you are coding in straight C, you can use the `Block_copy()` function, as follows:

```
int(^doublerCopy)(int) = Block_copy( ^(int n){ return n * 2; } );
```

In Objective-C, you can send a `copy` message to the block:

```
int(^doublerCopy)(int) = [^(int n){ return n * 2; } copy];
```

The two preceding examples are equivalent. In either statement, you could use a block pointer instead of the block literal.

When you copy a block, the new block gets copies of the values of any automatic variables that the block references. (The block accesses automatic variables by value. The value of the variable is copied into the block object when it is created.)

But What About Block Variables?

Block variables are accessed by reference. It wouldn't be very useful to copy a block and then leave the copied block referring to a variable that is destroyed when the program execution leaves the current scope.

To remedy this, when you copy a block, the compiler also moves any block variables that the block references from the stack to a location on the heap. The compiler then updates any blocks that reference the block variable so they have the variable's new address.

One consequence of the compiler's behavior in this situation is that it is a *very* bad idea to take the address of a block variable and use it for anything. After the copy operation, the original address refers to a memory location that may now be garbage.

Memory Management for Blocks

If you copy a block with `Block_copy()`, you must eventually balance that call with a call to `Block_release()`. If you use the Objective-C copy message and you are using reference counting, you must balance the `copy` message with a `release` or an `autorelease`:

```
int(^getDoublerBlock())(int)
{
  int(^db)(int) = ^(int n){ return 2*n; };

  // The returned block is autoreleased. This balances the copy
  // and makes getDoublerBlock conform to the naming convention
  // for memory management.
  return [[db copy] autorelease];
}
...

int(^doubler )(int) = getDoublerBlock();  // Get the block

int sevenDoubled = doubler(7); // Use the block
```

Don't mix calls to `Block_copy()` and `Block_release()` with the Objective-C's `copy` and `release` messages.

If a block references a variable that holds an object, that object is retained when the block is copied and released when the block is released.

> **Note**
>
> An object held in `__block` variable is *not* retained when a block that references it is copied.

When copying a block inside a method body, the rules are slightly more complicated:

- A direct reference to `self` in a block that is being copied causes `self` to be retained.
- A reference to an object's instance variable (either directly or through an accessor method) in a block that is being copied causes `self` to be retained.
- A reference to an object held in a local variable in a method causes that object, but *not* `self`, to be retained.

You should be careful when copying a block. If the code that copies the block is inside a method and the block refers to any of the object's instance variables, the copy causes `self` to be retained. It is easy to set up a retain cycle that prevents the object from ever being deallocated,

Listing 16.3 shows the interface section for a class that has an instance variable `name` to store a name, and a method `logMyName` to log that name. `logMyName` uses a block stored in the instance variable `loggingBlock` to do the actual logging.

Listing 16.3 **ObjectWithName.h**

```
#import <Foundation/Foundation.h>

@interface ObjectWithName : NSObject
{
```

```
  NSString *name;
  void (^loggingBlock)(void);
}

- (void) logMyName;
- (id) initWithName:(NSString*) inName;

@end
```

Listing 16.4 shows the corresponding implementation file.

Listing 16.4 **ObjectWithName.m**

```
 1 #import "ObjectWithName.h"
 2
 3 @implementation ObjectWithName
 4
 5 - (id) initWithName:(NSString*) inputName
 6 {
 7   if (self = [super init] )
 8     {
 9       name = [inputName copy];
10       loggingBlock = [^(void){ NSLog( @"%@", name ); } copy];
11     }
12   return self;
13 }
14
15 - (void) logMyName
16 {
17   loggingBlock();
18 }
19
20 - (void) dealloc
21 {
22   [loggingBlock release];
23   [name release];
24   [super dealloc];
25 }
```

`ObjectWithName` is a very simple class. However, this version of `ObjectWithName` has a retain cycle. If you create an `ObjectWithName` object, it won't be deallocated when you release it.

The problem is Line 10 of Listing 16.4:

```
loggingBlock = [^(void){ NSLog( @"%@", name ); } copy];
```

To store the block in the instance variable `loggingBlock`, you must copy the block literal and assign the copy to the instance variable. This is because the block literal goes out of scope when `initWithName:` returns. Copying the block puts the copy on the heap (like a normal Objective-C object). However, the block literal references the instance variable `name`, so the `copy` causes `self` to be retained, setting up a retain cycle. The block now has ownership of the object and the object has ownership of the block (because it has copied the block). The object's reference count never goes to zero and its `dealloc` method is never called.

You can fix this problem by changing Line 10 of Listing 16.4 so it reads as follows:

```
loggingBlock = [^(void){ NSLog( @"%@", inputName ); } copy];
```

With this change, the block copying operation retains the input argument `inputName` rather than the instance variable `name`. Because the block no longer references any of the object's instance variables, `self` is not retained and there is no retain cycle. The object will still have the same behavior because `name` and `inputName` have the same content.

> **Note**
>
> The preceding rules are presented separately for blocks copied inside a method to emphasize the consequences of a block accessing an object's instance variable. But there is really no significant difference between copying a block inside or outside a method. The only difference is that, outside of a method, there is no way to reference an object's instance variables without referencing the object itself.

Traps

Because blocks are stack-based objects, they present some traps for the unwary programmer. The following snippet of code is incorrect:

```
void(^loggingBlock)(void);

BOOL canWeDoIt = ...

// WRONG

if ( canWeDoIt )
  loggingBlock = ^(void){ NSLog( @"YES" ); };
else
  loggingBlock = ^(void){ NSLog( @"NO" ); };

// Possible crash
loggingBlock();
```

At the end of this snippet, `loggingBlock` is undefined. The `if` and `else` clauses of an `if` statement and the bodies of loops are separate lexical scopes, *even if they are single statements and not compound statements.* When the program execution leaves the scope, the compiler is free to destroy the block and leave `loggingBlock` pointing at garbage.

To fix this code, you must `copy` the block, and then remember to release it when you are finished:

```
void(^loggingBlock)(void);

BOOL canWeDoIt = ...

if ( canWeDoIt )
  loggingBlock = [^(void){ NSLog( @"YES" ); } copy];
else
  loggingBlock = [^(void){ NSLog( @"NO" ); } copy];

// Remember to release loggingBlock when you are finished
```

This example is also incorrect:

```
NSMutableArray *array = ...

// WRONG!

[array addObject: ^(void){ doSomething; }];
return array; //
```

Recall that objects added to collection objects receive a retain message; however in this case, the retain doesn't help because retain is a no-op for a stack-based block. Again, to fix the problem, you must `copy` the block:

```
NSMutableArray *array = ...
[array addObject: [[^(void){ doSomething; } copy] autorelease]];
return array;
```

In the preceding code snippet, the `copy` message puts a copy of the block on the heap. The `autorelease` message balances the `copy`. The `retain` message that the copied block receives when it is placed in the array is balanced by a `release` message when the block is later removed from the array or when the array is deallocated.

Blocks in Cocoa

Beginning with Mac OS X Snow Leopard (v 10.6), Apple has started deploying blocks throughout the Cocoa frameworks. This section briefly describes three areas where Apple has added features that use blocks.

Concurrency with `NSOperationQueue`

Concurrent (multithreaded) programming is very difficult to do correctly. To make it easier for programmers to write error-free multithreaded programs, Apple has introduced Grand Central Dispatch (GCD). GCD implements concurrency by creating and managing a *thread pool*. A thread pool is a group of threads that can be assigned to various tasks and reused when the task is finished. GCD hides the details of managing the thread pool and presents a relatively simple interface to programmers.

The Cocoa class `NSOperationQueue` provides a high-level interface to GCD. The idea is simple: You create an `NSOperationQueue` and add units of work, in the form of blocks, for the queue to execute. Underneath `NSOperationQueue`, GCD arranges to execute the block on a separate thread:

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];

[queue addOperationWithBlock: ^(void){ doSomething; } ];

// doSomething will now execute on a separate thread
```

- A block passed to GCD (either through `NSOperationQueue` or through the low-level C interface) must have the form:

  ```
  void (^block)(void)
  ```

  It must not take arguments or return a value.
- The GCD mechanism takes care of copying blocks submitted to it and releases them when no longer needed.

> **Note**
>
> Programming concurrency is a complex topic. For a complete discussion of `NSOperationQueue` and GCD, see Apple's *Concurrency Programming Guide.*[2]

Collection Classes

The Foundation collection classes now have methods that enable you to apply a block to every object in the collection. `NSArray` has the following method:

```
- (void)enumerateObjectsUsingBlock:
      (void (^)(id obj, NSUInteger idx, BOOL *stop))block
```

This method calls `block` once for each object in the array; the arguments to `block` are:

- `obj`, a pointer to the current object.
- `idx`, the index of the current object (`idx` is the equivalent of the loop index in an ordinary `for` loop).
- `stop`, a pointer to a `BOOL`. If the block sets stop to `YES`, `-enumerateObjectsUsingBlock:` terminates when the block returns. It is the equivalent of a `break` statement in an ordinary C loop.

Listing 16.5 uses `-enumerateObjectsUsingBlock:` to log a description of every object in an array.

2. http://developer.apple.com/mac/library/documentation/General/Conceptual/ConcurrencyProgrammingGuide

Listing 16.5 **DescribeArrayContents.m**

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  NSArray *array =
      [NSArray arrayWithObjects: @"dagger", @"candlestick",
           @"wrench", @"rope", nil];

  void (^loggingBlock)(id obj, NSUInteger idx, BOOL *stop) =
      ^(id obj, NSUInteger idx, BOOL *stop)
          { NSLog( @"Object number %d is a %@",
               idx, [obj description] ); };

  [array enumerateObjectsUsingBlock: loggingBlock];

  [pool drain];
  return 0;
}
```

If you build and run this program, you should see the following result:

```
DescribeArrayContents [50642:a0b] Object number 0 is a dagger
DescribeArrayContents [50642:a0b] Object number 1 is a candlestick
DescribeArrayContents [50642:a0b] Object number 2 is a wrench
DescribeArrayContents [50642:a0b] Object number 3 is a rope
```

Did-End Callbacks

I haven't said much about AppKit in this book, but I'll assume that you are familiar with saving files on Mac OS X. You select **File > Save** in an app, and if this is the first time the file is saved, a save sheet appears so you can name the file and select the location where it will be saved. You make your choices and click Save, or if you've changed your mind, you can click Cancel. After clicking one of the buttons, the sheet slides up and disappears.

When you invoke the method that begins the sheet, Cocoa gives you the chance to register some code to be executed when the user dismisses the sheet. (This is where you put the code that actually saved the file to disk.)

Prior to Mac OS X Snow Leopard (v 10.6), a Save sheet was started with this rather formidable method:

```
- (void)beginSheetForDirectory:(NSString *)path
        file:(NSString *)name
        modalForWindow:(NSWindow *)docWindow
```

```
modalDelegate:(id)modalDelegate
didEndSelector:(SEL)didEndSelector
contextInfo:(void *)contextInfo
```

When the user dismisses the sheet, the sheet sends the object registered as `modalDelegate`, the message represented by the selector `didEndSelector`. Typically, the `modalDelegate` is the object that initiates the panel. `didEndSelector` has the form:

```
- (void)savePanelDidEnd:(NSSavePanel *)sheet
            returnCode:(int)returnCode
           contextInfo:(void *)contextInfo;
```

- `sheet` is a pointer to the `NSSavePanel` object itself.
- `returnCode` is an integer that specifies which button the user clicked on.
- `contextInfo` is a blind pointer to the information passed to `beginSheetForDirectory: ...` when it was invoked. This is how you pass information from the object that invoked the sheet to the code responsible for acting on the user's input.

For Mac OS X Snow Leopard and beyond, the preceding method has been deprecated[3] and replaced with the following method:

```
- (void)beginSheetModalForWindow:(NSWindow *)window
             completionHandler:(void (^)(NSInteger result))handler
```

You simply pass in a block to be executed when the sheet is dismissed. The block can capture any required context so the blind `contexInfo` pointer is not required.

> **Note**
>
> The `file` and `directoryPath` arguments were removed as part of a separate cleanup that doesn't involve blocks.

Style Issues

Placing the statements of a block literal on a single line makes debugging difficult; for example:

```
^(void){doStuff; doMoreStuff; evenMore; keepGoing; lastStatement;}
```

You can set a debugger breakpoint on `doStuff;` but there is no way to step through or set a breakpoint on any of the other statements in the block. If you stop on `doStuff;` and try to step, the debugger jumps to the line following the block literal—making it

3. A method whose status is changed to deprecated in a given major OS release is still available in that release but may be withdrawn in a future major OS release. For example, a method marked as deprecated in Mac OS X 10.6 may not be available in Mac OS X 10.7.

impossible to debug the subsequent lines in the literal. If your block literal is non-trivial and may require debugging, you should put the statements in the block's body on separate lines, as follows:

```
^(void){doStuff;
        doMoreStuff;
        evenMore;
        keepGoing;
        lastStatement;}
```

As noted earlier, you can place a block literal directly in a function or method call:

```
someFunction( otherArgs, ^(void){ doStuff;
                                  doMoreStuff;
                                  evenMore;
                                  keepGoing;
                                  lastStatement;} );
```

You could also assign the block to a block pointer variable and use the block pointer as the argument. Which you choose is a matter of preference: Some people are annoyed at creating an extra variable (which the compiler will probably optimize away), whereas others find that putting the block literal inside the function call makes the code hard to read.

Some Philosophical Reservations

Blocks are very versatile and they are clearly an important part of Apple's plans for the future of Objective-C and Mac OS X. However, blocks come with a few issues that are worth a moment or two of thought:

- The term "block" was already in use. It is used as interchangeable with "compound statement" in almost every book on the C language. This might cause confusion in some circumstances.
- Blocks are *function oriented* and not very *object oriented*. This may be an issue if you are strongly attached to an ideal of object-oriented purity.
- Blocks completely break encapsulation. A block's access to variables that are not accessed through an argument list or an accessor method presents many of the same issues as using global variables.

  Using `__block` variables and copying blocks can result in entangled objects: You can create separate objects (potentially belonging to different classes) communicating via a variable on the heap that is not visible to anything else.
- As with operator overloading in C++, blocks can be used in ways that lead to *Design Your Own Language Syndrome*, code that is very terse but very difficult for others (or yourself, several months later) to read and understand. This may not be an issue for independent developers, but it can be a problem if you are part of a programming team.

Summary

This chapter looked at several ways of packaging functionality to be executed at a later time or on a different thread. Function pointers let you hand off functions but require that you provide an extra variable to go with the function pointer if you need to pass some context to go along with the function. `NSInvocation` objects wrap the target, the selector, and the arguments of an Objective-C message expression in a single object that can then be stored or handed off for later execution. They are easy to use but difficult to construct.

Blocks, an Apple-added extension to C, Objective-C 2.0, and C++, wrap a series of statements and the variables in their surrounding context in a single entity. Grand Central Dispatch, Apple's system for managing concurrency, uses blocks as the medium for submitting tasks to be executed on other threads. Beginning with Mac OS X Snow Leopard (v 10.6), Apple is deploying blocks throughout the Cocoa frameworks to replace older methods that used `NSInvocation` objects or required separate target, selector, and context arguments for callbacks.

Exercises

1. This is more of a puzzle than anything else, but it will test your understanding of function pointer (and, by extension, block pointer) declarations. Consider the following declaration:

   ```
   int (*(*myFunctionPointer)(int (*)(int))) (int);
   ```

   What (in words) is `myFunctionPointer`?

2. Rewrite the `HelloObjectiveC` program from Chapter 4, "Your First Objective-C Program," to use an `NSInvocation`:

   Instead of passing the `Greeter` the greeting text as an `NSString`, create a `Greeting` class that encapsulates the greeting and a method that issues the greeting. (The method should take the greeting string as an argument and log it.)

   Package up issuing the greeting as an `NSInvocation` and pass it to the `Greeter`.

   The `Greeter` should then issue the greeting by sending the invocation object the `-invoke` message.

3. Write a program that uses some simple blocks and verify for yourself that:
   - The value for an ordinary automatic value that a block sees is fixed when execution passes over the block literal, and is unchanged if the value of the variable is changed later in the code.
   - A block cannot modify the value of an ordinary automatic variable in its scope.
   - A block can both read and set a variable declared with the `__block` type modifier.

4. Rewrite the program in Listing 16.1 to use a block instead of a function. Use the `NSMutableArray` method:

   ```
   - (void)sortUsingComparator:(NSComparator)cmptr
   ```

   `NSComparator` is a typedef for a pointer to a block that takes two object arguments and returns the same integer constants as the function in Listing 16.1.

5. Write a program that looks for a name in an array of names and reports back the name's index in the array:
   - Create an `NSArray` with some names (use `NSString` objects).
   - Create a local `NSString` variable to hold the name you are searching for and an integer block variable to report back at what index the name was found.
   - Search the array using `-enumerateObjectsUsingBlock:`.
   - Make sure your block uses the `stop` argument to stop looking when the name is found.
   - If the name you are looking for isn't in the array, the block variable holding the index should have a value of -1.

6. Write a program that uses the `ObjectWithName` class (see Listings 16.3 and 16.4):
   - Add a logging statement to `ObjectWithName`'s `dealloc` routine.
   - In your main program, allocate an instance of `ObjectWithName`.
   - Release the object and verify that it is never deallocated.
   - Make the fix suggested in the text and verify that the object now deallocates.

Index

Symbols

Numbers

A

F

H

I

O

P

Q

R