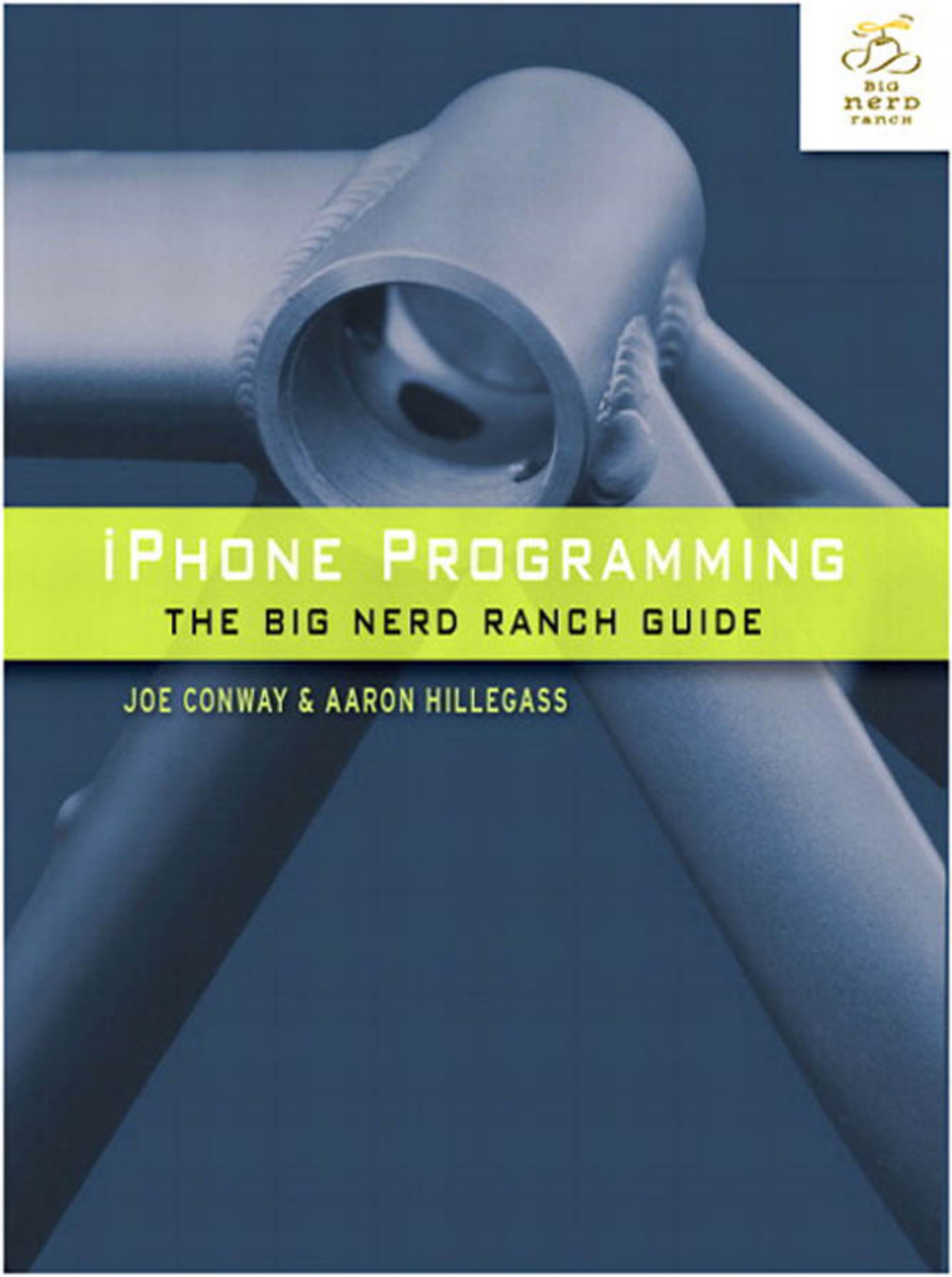

Big nerd ranch
iPhone Programming
THE BIG NERD RANCH GUIDE
JOE CONWAY & AARON HILLEGASS

iPhone Programming

THE BIG NERD RANCH GUIDE

JOE CONWAY & AARON HILLEGASS

Big nerd ranch

iPhone Programming: The Big Nerd Ranch Guide

by Joe Conway and Aaron Hillegass

Big Nerd Ranch, Inc.
1963 Hosea L. Williams Drive SE
Suite 209
Atlanta, GA 30317
(404) 478-9005
http://www.bignerdranch.com/
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, Inc.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-13 978-0321706249
ISBN-10 0321706242

Library of Congress Control Number: 2010903421

Second printing, August 2010

Table of Contents

Introduction

An aspiring iPhone developer faces three basic hurdles:

- *You must learn the Objective-C language*. Objective-C is a small and simple extension to the C language. After the first four chapters of this book, you will have a working knowledge of Objective-C.

- *You must master the big ideas*. These include things like memory management techniques, delegation, archiving, and the proper use of view controllers. The big ideas take a few days to understand. When you reach the halfway point of this book, you will understand these big ideas.

- *You must master the frameworks*. The eventual goal is to know how to use every method of every class in every framework on the iPhone. This is a project for a lifetime: there are over 3000 methods and more than 200 classes available for the iPhone. To make things even worse, Apple adds new classes and new methods with every release of the iPhone OS. In this book, you will be introduced to each of the subsystems that make up the iPhone SDK, but we will not study each one deeply. Instead, our goal is get you to the point where you can search and understand Apple's reference documentation.

We have used this material many times at our iPhone Development Bootcamp at Big Nerd Ranch. It is well-tested and has helped hundreds of people become iPhone application developers. We sincerely hope that it proves useful to you.

Prerequisites

This book assumes that you are already motivated to learn to write iPhone apps. We won't spend any time convincing you that the iPhone is a compelling piece of technology.

We also assume that you know the C programming language and something about object-oriented programming. If this is not true, you should probably start with an introductory book on C and Objective-C. We recommend Kochan's *Programming in Objective-C*.

Our Teaching Philosophy

This book is based on our iPhone Development Bootcamp course. It will teach you the essential concepts of iPhone programming. At the same time, you'll type in a lot of code and build a bunch of applications. By the end of the book, you'll have knowledge *and* experience. However, all the knowledge shouldn't (and, in this book, won't) come first. That's sort of the traditional way we've all come to know and hate. Instead, we take a learn-while-doing approach. Development concepts and actual coding go together.

Here's what we've learned over the years of teaching iPhone programming:

- We've learned what ideas people must have to get started programming, and we focus on that subset.

- We've learned that people learn best when these concepts are introduced *as they are needed*.

- We've learned that programming knowledge and experience grow best when they grow together.
- We've learned that "going through the motions" is much more important than it sounds. Many times we'll ask you to start typing in code before you understand it. We get that you may feel like a trained monkey typing in a bunch of code that you don't fully grasp. But the best way to learn coding is to find and fix your typos. Far from being a drag, this basic debugging is where you really learn the ins and outs of the code. That's why we encourage you to type in the code yourself. You could just download it, but copying and pasting is not programming. We want better for you and your skills.

What does this mean for you, the reader? To learn this way takes some trust. And we appreciate yours. It also takes patience. As we lead you through these chapters, we will try to keep you comfortable and tell you what's happening. However, there will be times when you'll have to take our word for it. (If you think this will bug you, keep reading — we've got some ideas that might help.) Don't get discouraged if you run across a concept that you don't understand right away. Remember that we're intentionally *not* providing all the knowledge you will ever need all at once. If a concept seems unclear, we will likely discuss it in more detail later when it becomes necessary. And some things that aren't clear at the beginning will suddenly make sense when you implement them the first (or the twelfth) time.

People learn differently. It's possible that you will love how we hand out concepts on an as-needed basis. It's also possible that you'll find it frustrating. In case of the latter, here are some options:

- Take a deep breath and wait it out. We'll get there, and so will you.
- Check the index. We'll let it slide if you look ahead and read through a more advanced discussion that occurs later in the book.
- Check the online Apple documentation. This is an essential developer tool, and you'll want plenty of practice using it. Consult it early and often.
- If it's Objective-C or object-oriented programming concepts that are giving you a hard time (or if you think they will), try Kochan's *Programming in Objective-C*. It's a great book that presents these concepts in a more traditional way.

How To Use This Book

This book is based on the class we teach at Big Nerd Ranch. As such, it was designed to be consumed in a certain manner.

Set yourself a reasonable goal, like "I will do one chapter every day." When you sit down to attack a chapter, find a quiet place where you won't be interrupted for at least an hour. Shut down your email, your Twitter client, and your chat program. This is not a time for multi-tasking; you will need to concentrate.

Do the actual programming. You can read through a chapter first, if you'd like. But the real learning comes when you sit down and code as you go. You will not really understand the idea until you have written a program that uses it and, perhaps more importantly, debugged that program.

A couple of the exercises require supporting files. For example, the SQLite exercise is a lot more fun if you have some data to browse. Thus, we have made a script that inserts data into a SQLite file. You can download these resources and solutions to the exercises from `http://www.bignerdranch.com/solutions/iPhoneProgramming.zip`.

There are two types of learning. When you learn about the Civil War, you are simply adding details to a scaffolding of ideas that you already understand. This is what we will call "Easy Learning". Yes, learning about the Civil War can take a long time, but you are seldom flummoxed by it. Learning iPhone programming, on the other hand, is "Hard Learning," and you may find yourself quite baffled at times, especially in the first few days. In writing this book, we have tried to create an experience that will ease you over the bumps in the learning curve. Here are two things you can do to make the journey easier:

- Find someone who already knows iPhone programming and will answer your questions. In particular, getting your application onto the device the first time is usually very frustrating if you are doing it without the help of an experienced developer.
- Get enough sleep. Sleepy people don't remember what they have learned.

How This Book Is Organized

In this book, each chapter addresses one or more ideas of iPhone development followed by hands-on practice. For more coding practice, we issue challenges towards the end of each chapter. We encourage you to take on at least some of these. They are excellent for firming up the concepts introduced in the chapter and making you a more confident iPhone programmer. Finally, most chapters conclude with one or two "For the More Curious" sections that explain certain consequences of the concepts that were introduced earlier.

Chapter 1 introduces you to iPhone programming as you build and deploy a tiny application. You'll get your feet wet with Xcode, Interface Builder, and the iPhone simulator along with all the steps for creating projects and files. The chapter includes a discussion of Model-View-Controller and how it relates to iPhone development.

Chapters 2 and 3 provide an overview of Objective-C and memory management. Although you won't create an iPhone application in these two chapters, you will build and debug a tool called RandomPossessions to ground you in these concepts. (You will reuse this tool and its related class in the Homepwner application introduced in Chapter 10.)

In Chapters 4 and 5, you will learn about the Core Location and MapKit frameworks and create a mapping application called Whereami. You will also get plenty of experience with the important design pattern of delegation and working with protocols, frameworks, and object diagrams.

Chapters 6 and 7 focus on the iPhone user interface with the Hypnosister and HypnoTime applications. You will get lots of practice working with views and view controllers as well as implementing scrolling, zooming, paging, and navigating between screens.

Chapter 8 covers the iPhone's accelerometer.You will learn how to obtain, filter, and use the data from the accelerometer to handle motion events, including shakes. You will use accelerometer data to add a new feature to the HypnoTime application.

In Chapter 9, you will create a smaller application named HeavyRotation while learning about **`UIDevice`** notifications and how to implement autorotation in an application.

Chapter 10 introduces the largest application in the book — Homepwner. (By the way, "Homepwner" is not a typo; you can find the definition of "pwn" at `www.urbandictionary.com`.) This application keeps a record of your possessions in case of fire or another catastrophe. Homepwner will take nine chapters total to complete.

In Chapters 10, 11, and 16, you will build experience developing tables on the iPhone. You will learn about table views, their view controllers, and their data sources. You will learn how to display data in a table, how to allow the user to edit the table, and how to improve the interface.

Chapter 12 builds on the navigation experience gained in Chapter 7. You will learn how to use **`UINavigationController`**, and you will give Homepwner a drill-down interface and a navigation bar.

In Chapter 13, you'll learn how to take pictures with the iPhone's camera and how to display and store images in Homepwner. You'll use **`NSDictionary`** and **`UIImagePickerController`**. You'll also learn about **`UIPopoverController`** for the iPad.

Chapter 14 delves into ways to save and load data. In particular, you will archive data in the Homepwner application using the `NSCoding` protocol. The chapter also shows you how to work with multitasking and transistions between application states, such as active, background, and suspended.

Chapter 15 teaches you how to prepare for low-memory warnings and leads you through handling low-memory warnings in Homepwner.

In Chapter 17, you'll take a break from Homepwner and create a drawing application named TouchTracker. You'll learn how to add multi-touch capability and more about touch events. You'll also get experience with the first responder and responder chain concepts and more practice with **`NSDictionary`**. In addition, you'll learn about the Instruments application while debugging performance and memory issues in TouchTracker.

Chapters 18 and 19 introduce layers and the Core Animation framework with a brief return to the HypnoTime application to implement animations. You will learn about implicit and explicit animations and animation objects, like **`CABasicAnimation`** and **`CAKeyFrameAnimation`**.

Chapter 20 will teach you how to play audio and video as you build an application called MediaPlayer. You will learn about playing audio and video on the iPhone, where to keep these resources, streaming limits, and the low-level audio API. You will also enable MediaPlayer to play music while in the background state and learn guidelines and other uses for background execution.

Chapter 21 ventures into the wide world of web services. You will fetch and parse XML data from the iTunes server in an application you create named TopSongs. You'll use **`NSURLConnection`** and **`NSXMLParser`** along the way.

In Chapter 22, you'll return to Homepwner to learn about the iPhone's Address Book functions and the People Picker as you update Homepwner to allow the user to assign people to inherit possessions.

Chapter 23 introduces the concepts and techniques of internationalization and localization. You will learn about **`NSLocale`**, strings tables, and **`NSBundle`** as you localize Homepwner.

Chapter 24 teaches you how to publish a service on the peer-to-peer network Bonjour. You will start a new application named Nayberz that advertises itself on the network.

Chapter 25 explores how to get an application to work with the iPhone's Settings application to create application settings and preferences that the user can customize. You will use **`NSUserDefaults`** and give Nayberz a pane in Settings.

Chapter 26 introduces the SQLite library for storing and fetching data on the iPhone. You get a chance to practice with a small data application named Nayshunz.

Chapter 27 gives you a good grounding in using Core Data to store and access data in an iPhone application. In this chapter, you will build a complex and business-like application named Inventory.

Chapter 28 introduces the iPad and some of its features, like **`UIGestureRecognizer`** and Core Text. You will turn the Whereami application into a universal application, enabling it to run natively on the iPad and the iPhone.

It is important to note something that is not covered in this book: OpenGL ES. We actually wrote a chapter. And then we rewrote it. And rewrote it. And rewrote it. It got longer with every pass. Thus, we've decided to take that chapter and expand it into a separate book.

Style Choices

This book contains a lot of code. We have attempted to make that code and the designs behind it exemplary. We have done our best to follow the idioms of the community, but at times we have wandered from what you might see in Apple's sample code or code you might find in other books. You may not understand these points now, but it is best that we spell them out before you commit to reading this book:

- There is an alternative syntax for calling accessor methods known as *dot-notation*. In this book, we will explain dot-notation, but we will not use it. For us and most beginners, dot-notation tends to obfuscate what is really happening.
- In our subclasses of **`UIViewController`**, we always change the designated initializer to **`init`**. It is our opinion that the creator of the instance should not need to know the name of the NIB file that the view controller uses, or even if it has a NIB file at all.
- We will always create view controllers programmatically. Some programmers will instantiate view controllers inside XIB files. We've found this practice leads to projects that are difficult to comprehend and debug.
- We will nearly always start a project with the simplest template project: the window-based application. The boilerplate code in the other template projects doesn't follow the rules that precede this one, so we think they make a poor basis upon which to build.

We believe that following these rules makes our code easier to understand and easier to maintain. After you have worked through this book (where you *will* do it our way), you should try breaking the rules to see if we're wrong.

Typographical Conventions

To make this book easier to read, certain items appear in certain fonts. Class names, method names, and function names appear in a bold, fixed-width font. Class names start with capital

letters, and method names start with lowercase letters. In this book, method and function names will be formatted the same for simplicity's sake. For example, “In the **`loadView`** method of the **`RexViewController`** class, use the **`NSLog`** function to print the value to the console.”

Variables, constants, and types appear in a fixed-width font but are not bold. So you'll see, “The variable `fido` will be of type `float`. Initialize it to `M_PI`.”

Applications and menu choices appear in the Mac system font. For example, “Open XCode and select New Project... from the File menu.”

All code blocks will be in a fixed-width font. Code that you need to type in is always bold. For example, in the following code, you would type in everything but the first and last lines. (Those lines are already in the code and appear here to let you know where to add the new stuff.)

```
@interface QuizAppDelegate : NSObject <UIApplicationDelegate> {
    int currentQuestionIndex;

    // The model objects
    NSMutableArray *questions;
    NSMutableArray *answers;

    // The view objects
    IBOutlet UILabel *questionField;
    IBOutlet UILabel *answerField;
    UIWindow *window;
}
```

Hardware, Software, and Deployment

To develop iPhone applications, you will need an Intel Mac running Mac OS X Leopard (or above). You will also need to download the iPhone SDK (Software Development Kit). The SDK includes Xcode (Apple's Integrated Development Environment), the iPhone simulator, and other development tools. To download the iPhone SDK, you only need to register as an iPhone Developer, which is free. As a “Registered iPhone Developer,” you will be able to access the iPhone Dev Center (including the Development docs). Go to `http://developer.apple.com/iphone/program/download.html` to register. Make sure you have the USB cable that connects the device to the computer.

You can do a lot with just the simulator, but for more complete and realistic testing, you'll want to install your applications on a real device — an iPhone, iPad, or iPod touch. (Nearly everything in this book will apply to all three devices, but we will usually refer to the “iPhone.” The iPad runs the same OS as the iPhone, and writing iPad applications uses the same techiniques with a few additions discussed in the final chapter. The iPod touch is nearly the same as the iPhone except for the telephone.) To install applications on your iPhone or to distribute them on the App Store, you have to join the “iPhone Developer Program,” which costs $99/year. Go to `http://developer.apple.com/` to join.

Excited yet? Good. Let's get started.

3
Memory Management

Understanding memory management in the Cocoa Touch framework is one of the first major roadblocks for newcomers. Unlike Objective-C on the Mac, Objective-C on the iPhone has no garbage collector. Thus, it is your responsibility to clean up after yourself.

Memory Management Concepts

This book assumes you are coming from a C background, so the words "pointer," "allocate," and "deallocate" shouldn't scare you. If your memory is a little fuzzy, here's a review. The iPhone has a limited amount of random access memory. Random access memory (RAM) is much faster to write to and read from than a hard drive, so when an application is executing, all of the memory it consumes is taken from RAM. When an operating system like iPhone OS launches your application, it reserves a heaping pile of the system's unused RAM for your application. Not-so-coincidentally, the memory your application has to work with is called the *heap*. The heap is your application's playground; it can do whatever it wants to it, and it won't affect the rest of the OS or any other applications.

When your application creates an instance of a class, it goes to the giant heap of memory it was given and takes a little scoop. Since you typically create objects during the course of your application's execution, you start using more and more of the heap. Most objects are not permanent, and when an object is no longer needed, the memory it was consuming should be returned to the heap. This way, it can be reused for another object created later.

There are two major problems in managing memory:

premature deallocation	You must never return memory to the heap until you are sure that no part of the program is still using it.
memory leaks	When a chunk of memory is no longer needed by any part of a program, it must be freed so that the memory can be used again.

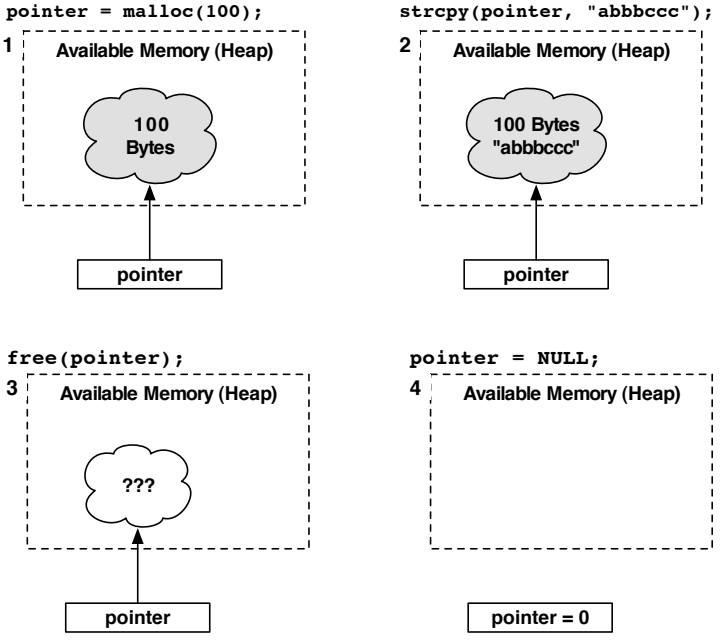
Managing memory in C

In the C programming language, you have to explicitly ask the heap for a certain number of bytes. This is called *allocation*. It is the first stage of the heap life cycle shown in Figure 3.1. To do this, you use a function like **`malloc`**. If you want 100 bytes from that heap, you do something like this:

```
void function(void)
{
    char *buffer = malloc(100);
}
```

You then have 100 bytes with which you can perform some task like writing a string to it and then printing that string (which would require reading from those bytes). The location of the first of those 100 bytes is stored in the pointer `buffer`. You access the 100 bytes by using this pointer.

Figure 3.1 Heap allocation life cycle



When you don't want to use those bytes anymore, you have to give them back to the heap by using the **`free`** function. This is called *deallocation*.

```
void function(void)
{
    char *buffer = malloc(100);
    ... Fill the buffer with text ...
    ... Print to the console ...
    free(buffer);
}
```

By calling **`free`**, those 100 bytes (starting at the address stored in `buffer`) are returned to the heap. If another **`malloc`** function is executed, any of these 100 bytes are fair game to be returned. Those bytes could be divvied up into smaller sections, or they could become part of a larger allocation.

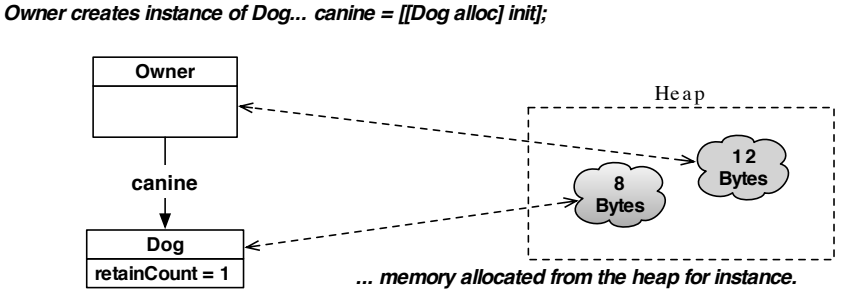
Because you don't know what will happen with those bytes when they are returned to the heap, it isn't safe to access them through the `buffer` pointer anymore.

Managing memory with objects

Even though at the base level an object is bytes allocated from the heap, you never explicitly call **`malloc`** or **`free`** with objects.

Every class knows how many bytes of memory it needs to allocate for an instance. When you create an instance of a class by sending it the **`alloc`** message, the correct number of bytes is allocated from the heap. Like with **`malloc`**, you are returned a pointer to this memory (Figure 3.2). However, when using Objective-C, we think in terms of objects rather than raw memory. While our pointers are still pointing to a spot in memory, we don't need to know the details of that memory; we just know we have an object.

Figure 3.2 Allocating an object



Of course, once you allocate memory from the heap, you need a way to return that memory back to the heap. Every object implements the method **`dealloc`**. When an object receives this message, it returns its memory back to the heap.

So, **`malloc`** is replaced with the class method **`alloc`**, and the function **`free`** is replaced with the instance method **`dealloc`**. However, you never explicitly send a **`dealloc`** message to an object; an object is responsible for sending **`dealloc`** to itself. That begs the question: if an object is in charge of destroying itself, how can it know if other objects are relying on its existence? This is where reference counting comes into play.

Reference Counting

In the Cocoa Touch framework, Apple has adopted *manual reference counting* to manage memory and avoid premature deallocation and memory leaks.

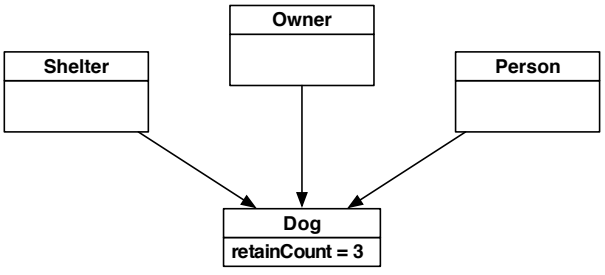
To understand reference counting, imagine a puppy. When the puppy is born, it has an owner. That owner later gets married, and the new spouse also becomes an owner of that dog. The dog is alive because they feed it. Later on, the couple gives the dog away. The new owner of the dog decides he doesn't like the dog and lets it know by kicking it out of the house. Having no owner, the dog runs away and, after a series of unfortunate events, ends up in doggy heaven.

What is the moral of this story? As long as the dog had an owner to care for it, it was fine. When it no longer had an owner, it ran away and ceased to exist. This is how reference counting works. When an object is created, it has an owner. Throughout its existence, it can have different owners, and it can have more than one owner at a time. When it has zero owners, it deallocates itself and goes to instance heaven.

Using retain counts

An object never knows *who* its owners are. It only knows its *retain count* (Figure 3.3).

Figure 3.3 Retain count for a dog



When an object is created — and therefore has one owner — its retain count is set to 1. When an object gains an owner, its retain count is incremented. When an object loses an owner, its retain count is decremented. When that retain count reaches 0, the object sends itself the message **`dealloc`**, which returns all of the memory it occupied to the heap.

Imagine how you would write the code to implement this scheme yourself:

```
- (id)retain
{
    retainCount++;
    return self;
}
- (void)release
{
    retainCount--;
    if (retainCount == 0)
        [self dealloc];
}
```

Simple, right? Now let's consider how retain counts work between objects. If object *A* creates object *B* (through **`alloc`** and **`init`**), *A* must send *B* the message **`release`** at some point in the future. Releasing *B* doesn't necessarily deallocate it; it is left to *B* to decide if it should be deallocated. (If *B* has another owner, it won't destroy itself.)

If some other object *C* wants to keep *B* around, *C* becomes an owner of *B* by sending it the message **`retain`**. What reason does *C* have to keep *B* around? *C* wants to send *B* messages.

Let's imagine you have a grocery list. You created it, so you own it. Later, you give that grocery list to your friend to do the shopping. You don't need to keep the grocery list anymore, so you release it. Your friend is smart, so he retained the list as soon as he was given it. Therefore, the grocery list will still exist whenever he needs it, and your friend is now the sole owner of the list.

Here is your code:

```
- (void)createAndGiveAwayTheGroceryList
{
   // Create a list
   GroceryList *g = [[GroceryList alloc] init];

   // (The retain count of g is 1)

   // Share it with your friend who retains it
   [smartFriend takeGroceryList:g];

   // (The retain count of g is 2)

   // Give up ownership
   [g release];

   // (The retain count of g is 1)
   // But we don't really care here, as this method's
   // responsibility is finished.
}
```

Here is your friend's code:

```
- (void)takeGroceryList:(GroceryList *)x
{
    // Take ownership
    [x retain];

    // Hold onto a pointer to the object
    myList = x
}
```

Retain counts can still go wrong in the two classic ways: leaks and premature deallocation. First, you could give the grocery list to your friend who retains it, but you don't release it. Your friend finishes the shopping and releases the list. You have forgotten where it is, but because you never released it, it still exists. Nobody has this grocery list anymore, but it still exists because its retain count is greater than 0. This is a leak.

Think of the grocery list as an **`NSString`**. You have a pointer to this **`NSString`** in the method where you created it. If you leave the scope of the method without releasing the **`NSString`**, you'll lose the pointer along with the ability to release the **`NSString`** later. Even if every other object releases the **`NSString`**, it will never be deallocated.

Consider the other way this process can go wrong — premature deallocation. You create a grocery list and give it to a friend, who doesn't retain it. When you release it (thinking it was safe with your friend), it is deallocated because you were its only owner. When your friend attempts to use the list, he can't find it because it doesn't exist anymore.

When an object attempts to access another object that no longer exists, your application accesses bad memory, starts to fail, and eventually (although sooner is better than later for debugging) crashes.

If an object retains another object, that other object is guaranteed to exist. So correct use of retain counts avoids premature deallocation. Now let's look more closely at memory leaks.

Avoiding memory leaks with autorelease

You already know that an object is responsible for returning its own bytes to the heap and that an object will do that when it has no owners. What happens when you want to create an object to give away, not to own? You own it by virtue of creating it, but you don't have any use for it.

Let's make this idea more concrete with an example from the RandomPossessions tool you wrote last chapter. In the **Possession** class, you implemented a convenience method called **randomPossession** that returns an instance of **Possession** with random parameters. The owner of this instance is the class **Possession** (because the object was created inside of a **Possession** class method), but **Possession** is only creating it because another object wants it. The pointer to the **Possession** instance is lost when the scope of **randomPossession** runs out, but the object still has a retain count of 1.

Now, in your **main** function, you could release the instance returned to you by this method. But, you didn't allocate the random possession in the **main** function. Therefore, releasing the memory isn't **main**'s responsibility. Since the **alloc** message was sent to the **Possession** class inside **randomPossession**'s implementation, it is **randomPossession**'s responsibility to release the memory. But looking at the following block of code, where could you safely release it?

```
+ (id)randomPossession
{
    ... Create random variables ...
    Possession *newPossession = [[self alloc]
                        initWithPossessionName:randomName
                                valueInDollars:randomValue
                                  serialNumber:randomSerialNumber];
    // If we release newPossession here,
    // the object is deallocated before it is returned.
    return newPossession;
    // If we release newPossession here, this code is never executed.
}
```

How can you avoid this memory leak? You need some way of saying "Don't release this object yet, but I don't want to be an owner of it anymore." Fortunately, you can mark an object for future release by sending it the message **autorelease**. When an object is sent **autorelease**, it is not immediately released; instead, it is added to an instance of the **NSAutoreleasePool**. This **NSAutoreleasePool** keeps track of all the objects that have been autoreleased. Periodically, the autorelease pool is drained; it sends the message **release** to the objects in the pool and then removes them.

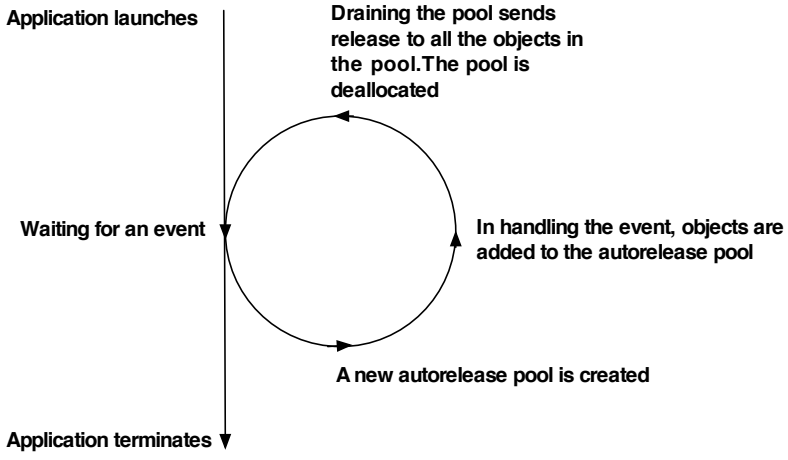
An object marked for autorelease after its creation has two possible destinies: it can either continue its death march to deallocation or another object can retain it. If another object retains it, its retain count is now 2. (It is owned by the retaining object, and it has not yet been sent **release**

by the autorelease pool.) Sometime in the future, that autorelease pool will release it, which will set its retain count back to 1. The return value for **`autorelease`** is the instance that is sent the message, so you can method chain **`autorelease`**.

```
// Because autorelease returns the object being autoreleased, we can do this:
NSObject *x = [[[NSObject alloc] init] autorelease];
```

Sometimes the idea of "the object will be released some time in the future" confuses developers. When an iPhone application is running, there is a run loop that is continually cycling. This run loop checks for events (like a touch or a timer firing) and then processes that event by calling the methods you have written in your classes. Whenever an event occurs, it breaks from that loop and starts executing your code. When your code is finished executing, the application returns to the loop. At the end of the loop, all autoreleased objects are sent the message **`release`**, as shown in Figure 3.4. So, while you are executing a method, which may call other methods, you can safely assume that an autoreleased object will not be released.

Figure 3.4 Autorelease pool draining



Managing memory in accessors and properties

Accessors are methods that get and set instance variables. Getter methods don't require any additional memory management:

```
- (Dog *)pet
{
    return pet;
}
```

Setters, however, need to take care to properly retain new values and release old ones.

```
- (void)setPet:(Dog *)d
{
    [d retain];         // Retain the new value
    [pet release];      // Release the old value
    pet = d;            // Make the pointer point at the new value
}
```

Notice that if `pet` hasn't been set, it is `nil`, and `[pet release]` would have no effect.

It is important to retain the new value before releasing the old one. Why? What if `pet` and `d` are pointers to the same object? What if that object has a retain count of 1? If you release it before you retain it, the retain count goes to 0, and the object is deallocated.

Here is the same thing in another style:

```
- (void)setPet:(Dog *)d
{
    if (pet != d) {
        [d retain];
        [pet release];
        pet = d;
    }
}
```

Once again, properties come to the rescue. If you use properties, all of the memory management code for your accessors is written for you when you synthesize the property. To have the compiler generate an accessor that properly releases and retains for you, you can use the `retain` attribute when declaring your properties in a header file:

```
@property (nonatomic, retain) Dog *pet;
```

Then, in the implementation file, synthesize the method:

```
@synthesize pet;
```

Retain count rules

Let's make a few rules from these ideas:

- If you send the message **`alloc`** to a class, the instance returned has a retain count of 1, and you are responsible for releasing it.
- If you send the message **`copy`** (or **`mutableCopy`**) to an instance, the instance returned has a retain count of 1, and you are responsible for releasing it (just as if you had allocated it).
- Assume that an object created through any other means (like a convenience method) has a retain count of 1 and is marked for autorelease.

- If an object wants to keep another object around (and the keeper didn't allocate it), it must send the wanted object the message **`retain`**.
- If an object no longer wants to keep another object around, it sends that object the message **`release`**.

There is one exception to the rules: in any method that starts with **`new`**, the object returned should be assumed to *not* be autoreleased.

Managing Memory in RandomPossessions

Now that you have the theory and some rules, you can implement better memory management in RandomPossessions. Open the `RandomPossessions.xcodeproj` file that you created in the last chapter. There are four memory management problems to fix in this project.

The first is found in the **`main`** function of `RandomPossessions.m` where you created an instance of **`NSMutableArray`** named `items`. You know two things about this instance: its owner is the **`main`** function and it has a retain count of one. It is then **`main`**'s responsibility to send this instance the message **`release`** when it no longer needs it. The last time you reference `items` in this function is when you print out all of its entries, so you can release it after that:

```
for(int i = 0; i < [items count]; i++) {
    NSLog(@"%@", [items objectAtIndex:i]);
}

[items release];
```

The object pointed to by `items` decrements its retain count when this line of code is executed. In this case, that object is deallocated because **`main`** was the only owner. If another object had retained `items`, it wouldn't have been deallocated.

There is one more detail to take care of. The instance of **`NSMutableArray`** that `items` pointed to is now gone. However, `items` is still storing the address that was the instance's location in memory. It is much safer to set the value of `items` to `nil`. Then any messages mistakenly sent to `items` will have no effect.

```
[items release];
items = nil;
```

The ordering of those two statements is important. Ordering them this way says, "Send the object release, and then clear my pointer to it." What would happen if you swapped the order of these statements? It would be the same thing as saying, "Set my pointer to this object to `nil` and then send the message **`release`** to.... Oh, no! I don't know where that object went!" You would leak the object: it wasn't released before you erased your pointer to it.

The second memory problem occurs when you create an instance of **`NSMutableArray`** and fill it with instances of **`Possession`** returned from the **`randomPossession`** convenience method:

```
    NSMutableArray *items = [[NSMutableArray alloc] init];
    for (int i = 0; i < 10; i++) {
        [items addObject:[Possession randomPossession]];
    }
```

The implementation for **randomPossession** returns an instance of type **Possession** that it created by sending the message **alloc**. This object is owned by this class method and therefore has a retain count of 1.

When you add a **Possession** instance to an **NSMutableArray**, the array becomes an owner of that object, so its retain count is increased to 2. After **randomPossession** finishes executing, however, it loses its pointer to the **Possession** it created. The **Possession** instance still has two owners, but only one still has a pointer to it (`items`). Memory leak!

This is a perfect opportunity to use **autorelease**. The method **randomPossession** should send **autorelease** to an instance it creates and relinquish its ownership of that instance. The object will still exist temporarily and be retained when it is added to the **NSMutableArray**. The instance of **NSMutableArray** will then be the sole owner of this new **Possession**. In effect, you have transferred ownership of the instance from **randomPossession** to `items`. When the array deallocates itself and releases the objects it contains, each object will have a retain count of 0 and will deallocate itself. Memory leak solved.

Now fix the leak in the **randomPossession** method in `Possession.m`.

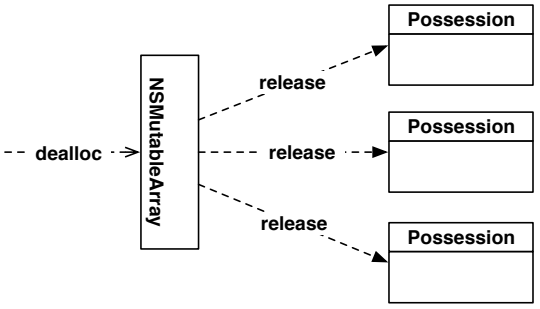
```
+ (id)randomPossession
{
    ... Create random variables ...
    Possession *newPossession = [[self alloc]
                                  initWithPossessionName:randomName
                                          valueInDollars:randomValue
                                            serialNumber:randomSerialNumber];

    return [newPossession autorelease];
}
```

When working with an instance of **NSMutableArray**, three rules apply to object ownership:

- When an object is added to an **NSMutableArray**, that object gets sent the message **retain**; the array becomes an owner of that object and has a pointer to it.
- When an object is removed from an **NSMutableArray**, that object gets sent the message **release**; the array relinquishes ownership of that object and no longer has a pointer to it.
- When an **NSMutableArray** is deallocated, it sends the message **release** to all of its entries as shown in Figure 3.5.

Figure 3.5 Deallocating an NSMutableArray



The third memory problem in RandomPossessions is in the **description** method that **Possession** implements. This method creates and returns an instance of **NSString** that needs to be autoreleased.

```
- (NSString *)description
{
    NSString *descriptionString =
        [[NSString alloc] initWithFormat:@"%@ (%@): Worth $%d, Recorded on %@",
                          possessionName,
                          serialNumber,
                          valueInDollars,
                          dateCreated];
    return [descriptionString autorelease];
}
```

You can make this even simpler by using a convenience method. **NSString** (as well as many other classes in the iPhone SDK) includes convenience methods that return autoreleased objects. Update **description** to use the convenience method **stringWithFormat:** to ensure that the **NSString** instance that **description** creates will be autoreleased.

```
- (NSString *)description
{
    return [NSString stringWithFormat:@"%@ (%@): Worth $%d, Recorded on %@",
                          possessionName,
                          serialNumber,
                          valueInDollars,
                          dateCreated];
}
```

The final memory problem has to do with the instance variables within **Possession** objects.

When the retain count of a **Possession** instance hits zero, it will send itself the message **dealloc**. The **dealloc** method of **Possession** has been implemented by its superclass, **NSObject**, but **NSObject** knows nothing about the instance variables added to **Possession**. So you must override **dealloc** in `Possession.m` to release any instance variables that have been retained.

```
- (void)dealloc
{
    [possessionName release];
    [serialNumber release];
    [dateCreated release];
    [super dealloc];
}
```

Always call the superclass implementation of **dealloc** at the end of the method. When an object is deallocated, it should release all of its own instance variables first. Then, it should go up its class hierarchy and release any instance variables of its superclass. In the end, the implementation of **dealloc** in **NSObject** will return the object's memory to the heap.

Now let's check your understanding of memory management concepts by looking more closely at instance variables and memory management.

Why send **release** to instance variables and not **dealloc**?

One object should never send **dealloc** to another. Always use **release** and let the object check its own retain count and decide whether to send itself **dealloc**.

Why do you need to release these instance variables in the first place? Where are the calls to **alloc**, **retain**, or **copy** that make an instance of **Possession** an owner of these objects?

Let's start with the instance variable `dateCreated`. Because it is allocated in the designated initializer for **Possession**, that instance of **Possession** becomes an owner and needs to release the object pointed to by `dateCreated` according to the first of the retain count rules described on page 52.

To figure out the other two instance variables, `possessionName` and `serialNumber`, you have to go back to their property declarations in `Possession.h`.

```
@property (nonatomic, copy) NSString *possessionName;
@property (nonatomic, copy) NSString *serialNumber;
```

Both of the properties associated with these instance variables have the `copy` attribute. When the message **setPossessionName:** is sent to an instance of **Possession**, the incoming parameter is sent the message **copy**. The instance variable `possessionName` is then set to point at that copied instance. If you wrote the code for **setSerialNumber:** instead of using `@synthesize`, it would look something like this:

```
- (void)setSerialNumber:(NSString *)newSerialNumber
{
    newSerialNumber = [newSerialNumber copy];
    [serialNumber release];
    serialNumber = newSerialNumber;
}
```

The second retain count rule states that, if an object copies something, the object becomes an owner of that thing. Therefore, the owning object needs to release the copied object in its **dealloc**

method. The same would hold true of these instance variables if their property attribute was `retain` (but not if the attribute were `assign`, which is a simple pointer assignment).

Strings come in two flavors: **`NSString`** and **`NSMutableString`**. Because an **`NSString`** can never be changed, there is seldom a need to copy it. Thus, in the case of **`NSString`** (and most other immutable objects), the copy method looks like this:

```
- (id)copyWithZone:(NSZone *)z
{
    [self retain];
    return self;
}
```

This approach prevents unnecessary copying. For example, the code above is basically equivalent to this:

```
- (void)setSerialNumber:(NSString *)newSerialNumber
{
    NSString *newValue;

    // Is it a mutable string?
    if ([newSerialNumber isKindOfClass:[NSMutableString class]])
        // I need to copy it
        newValue = [newSerialNumber copy];
    else
        // It is sufficient to retain it
        newValue = [newSerialNumber retain];

    [serialNumber release];
    serialNumber = newValue;
}
```

Note, however, that this code is not exactly equivalent. Because of some underlying implementation details, both **`NSString`** and **`NSMutableString`** might return `YES` if asked whether they are of type **`NSMutableString`**.

Sometimes in this book, we will show you example code that does not exactly match the implementation in the SDK. We do this to give you a better understanding of the concepts being discussed. We're not lying to you; we're just sparing you some of the details until you're more comfortable with the concepts. Once you're there, you can divine all the details from the documentation. (In fact, we already did this when we gave an example of implementing **`retain`** and **`release`** earlier in this chapter. The implementations of these methods are actually much dirtier.)

Congratulations! You've implemented retain counts and fixed four memory leaks. Your RandomPossessions application now manages its memory like a champ!

Keep this code around because you are going to use it in later chapters.

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z