David Chisnall

SECOND EDITION

Updated for ARC, OS X 10.7, and iOS 5

Objective-C

PHRASEBOOK

Objective-C

PHRASEBOOK

SECOND EDITION

David Chisnall

Addison-Wesley

DEVELOPER'S LIBRARY

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

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 is on file.

ISBN-13: 978-0-321-81375-6

ISBN-10: 0-321-81375-8

Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, Indiana.

First printing October 2011

Editor-in-Chief
Mark Taub

Acquisitions Editor
Mark Taber

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Bart Reed

Proofreader
Charlotte Kughen

Publishing Coordinator
Vanessa Evans

Cover Designer
Gary Adair

Senior Compositor
Gloria Schurick

Table of Contents

This page intentionally left blank

About the Author

David Chisnall is a freelance writer and consultant. While studying for his PhD, he co-founded the Étoilé project, which aims to produce an open-source desktop environment on top of GNUstep, an open-source implementation of the OpenStep and Cocoa APIs. He is an active contributor to GNUstep and is the original author and maintainer of the GNUstep Objective-C 2 runtime library and the associated compiler support in the Clang compiler.

After completing his PhD, David hid in academia for a while, studying the history of programming languages. He finally escaped when he realized that there were places off campus with an equally good view of the sea and without the requirement to complete quite so much paperwork. He occasionally returns to collaborate on projects involving modeling the semantics of dynamic languages.

David has a great deal of familiarity with Objective-C, having worked both on projects using the language and on implementing the language itself. He has also worked on implementing other languages, including dialects of Smalltalk and JavaScript, on top of an Objective-C runtime, allowing mixing code between all of these languages without bridging.

When not writing or programming, David enjoys dancing Argentine Tango and Cuban Salsa, playing badminton and ultimate frisbee, and cooking.

Acknowledgments

When writing a book about Objective-C, the first person I should thank is Nicolas Roard. I got my first Mac at around the same time I started my PhD and planned to use it to write Java code, not wanting to learn a proprietary language. When I started my PhD, I found myself working with Nicolas, who was an active GNUstep contributor. He convinced me that Objective-C and Cocoa were not just for Macs and that they were both worth learning. He was completely right: Objective-C is a wonderfully elegant language, and the accompanying frameworks make development incredibly easy.

The next person to thank is Fred Kiefer. Fred is the maintainer of the GNUstep implementation of the AppKit framework. He did an incredibly thorough (read: pedantic) technical review of this book, finding several places where things were not explained as well as they could have been. If you enjoy reading this book, then Fred deserves a lot of the credit.

Finally, I need to thank everyone else who was involved in bringing this book from my text editor to your hands, especially Mark Taber who originally proposed the idea to me.

We Want to Hear from You

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.

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

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

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

E-mail:	mark.taber@pearson.com
Mail:	Mark Taber
	Associate Publisher
	Addison Wesley Publishing
	800 East 96th Street
	Indianapolis, IN 46240 USA

Reader Services

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

Introduction

Blaise Pascal once wrote, “I didn’t have time to write a short letter, so I wrote a long one instead.” This phrasebook is the shortest book I’ve written, and trying to fit everything that I wanted to say into a volume this short was a challenge.

When Mark Taber originally suggested that I write an Objective-C Phrasebook, I was not sure what it would look like. A phrasebook for a natural language is a list of short idioms that can be used by people who find themselves in need of a quick sentence or two. A phrasebook for a programming language should fulfil a similar rôle.

This book is not a language reference. Apple provides a competent reference for the Objective-C language on the `http://developer.apple.com` site. This is not a detailed tutorial; unlike my other Objective-C book, *Cocoa Programming Developer’s Handbook*, you won’t find complete programs as code examples. Instead, you’ll find very short examples of Objective-C idioms, which hopefully you can employ in a wide range of places.

One of the most frustrating things in life is finding that code examples in a book don’t actually work. There are two sorts of code listings in this book. Code on a white background is intended to illustrate a simple point. This code may depend on some implied context and should not be taken as working, usable examples.

The majority of the code you will find in this book is on a gray background. At the bottom of each of these examples, you will find the name of the file that the listing was taken from. You can download these from the book's page on InformIT's website: `http://www.informit.com/title/0321743628`

When I wrote the first edition of this book, I wrote and tested all of the examples on OS X. After sending the draft manuscript off for editing, I tested them on GNUstep and was pleasantly surprised that almost all of them worked. By the time the book was published, they all worked. The second edition covers a number of features that are only supported by Apple on Mac OS X 10.7 or iOS 5. All of these examples also work with GNUstep. As before, I have written all of the examples on OS X, without making any concessions for GNUstep other than testing the examples there.

A Note About Typesetting

This book was written in Vim, using semantic markup. From here, three different versions are generated. Two are created using pdflatex. If you are reading either the printed or PDF version, then you can see one of these. The only difference between the two is that the print version contains crop marks to allow the printer to trim the pages.

The third version is XHTML, intended for the ePub edition. This is created using the EtoileText framework, which first parses the LaTeX-style markup to a tree structure, then performs some transformations for handling cross-references and indexing, and finally generates XHTML. The code for doing this is all written in Objective-C.

If you have access to both, you may notice that the code listings look slightly nicer in the ePub edition. This is because EtoileText uses SourceCodeKit, another Étoilé framework, for syntax highlighting. This uses part of Clang, a modern Objective-C compiler, to mark up the code listings. This means that ranges of the code are annotated with exactly the same semantic types that the compiler sees. For example, it can distinguish between a function call and a macro instantiation.

You can find all of the code for doing this in the Étoilé subversion repository: `http://svn.gna.org/viewcvs/etoile/trunk/Etoile/`

3

Memory Management

If you come from a C or C++ background, you're probably used to tracking ownership of objects and manually allocating and destroying them. If you're coming from a language such as Java, you're probably accustomed to having the garbage collector take care of all of this for you.

Objective-C does not, at the language level, provide anything for allocating or deallocating objects. This is left up to C code. You commonly allocate objects by sending their class a `+alloc` message. This then calls something like `malloc()` to allocate the space for the object. Sending a `-dealloc` message to the instance will then clean up its instance variables and delete it.

The Foundation framework adds reference counting to this simple manual memory management. This makes life much easier, once you understand how it works. Newer compilers provide some assistance for you, eliminating the need to write the reference counting code yourself.

Retaining and Releasing

```
NSArray *anArray = [NSArray array];
anArray = [[NSArray alloc] init];
[anArray release];
```

From: retainRelease.m

Every object that inherits from `NSObject` has a reference count associated with it. When this reference count reaches 0, it is destroyed. An object created with `+alloc` or any of the related methods, such as `+new` or `+allocWithZone:`, begins life with a reference count of one.

To control the reference count of an object, you send it `-retain` and `-release` messages. As their names would imply, you should use these messages when you want to retain a reference to an object, or when you want to release an existing reference. The `-retain` message increments the object's reference count, and the `-release` message decrements it.

You can also send a `-retainCount` message to an object to determine its current reference count. It's tempting to use this for optimization and invoke some special cases when you are sure there is only one reference to an object. This is a very bad idea. As the name implies, this method tells you the number of retained references to the object, not the number of references. It is common not to bother sending a `-retain` message to objects when you create a pointer to them on the stack. This means that an object may be referenced in two or more places, even though its retain count is only one.

Pointers in Objective-C are divided into two categories: *owning references* and *non-owning references*. An owning reference is one that contributes towards the retain count of an object. When you call a method like `+new` or `-retain`, you get an owning reference to a new object. Most other methods return a non-owning reference. Instance variables and global variables are typically owning pointers, so you should assign owning references to them. You also need to ensure that you delete the existing owning reference (by sending a `-release` message) when performing an assignment.

Temporary variables are typically non-owning references. Automatic reference counting and garbage collection, which we'll look at later in this chapter, both introduce special kinds of non-owning references.

Assigning to Instance Variables

```
- (void)setStringValue: (NSString*)aString
{
  id tmp = [aString retain];
  [string release];
  string = tmp;
}
```

From: ivar.m

There are a few things that you have to be careful about when using reference counting in this way. Consider the following simple set method:

```
- (void)setStringValue: (NSString*)aString
{
  [string release];
  string = [aString retain];
}
```

From: ivar.m

This looks sensible. You release the reference to the old value, then retain the new value and assign it. Most of the time, this will work, but in a few cases it won't, and that can be confusing to debug.

What happens if the value of `aString` and `string` are the same? In this case, you are sending the same object a `-release` message then a `-retain` message. If some other code holds references to this object, it will still work, but if not then the first message will cause the object to be destroyed and the second will be sent to a dangling pointer.

A more correct implementation of this method would retain the new object first, as shown at the start of this section. Note that you should assign the result of the `-retain` message because some objects will return another object when you retain them. This is very rare, but it does happen on occasion.

Finally, this method is not thread-safe. If you want a thread-safe set method, you need to retain the new value, perform an atomic exchange operation on the result and the instance variable, and then release the old value. In general, however, it is almost impossible to reason about code that supports this kind of fine-grained concurrency, and the amount of cache churn it causes will offset any performance gains from parallelism, so it's a terrible idea. If you really need it, it's better to use declared properties to synthesize the accessor than try to write it yourself.

Automatic Reference Counting

With the release of iOS 5 and Mac OS X 10.7, Apple introduced *Automatic Reference Counting (ARC)*. Conceptually, you can think of this as having the compiler automatically figure out when `-retain` and `-release` should be called, and calling them for you. The implementation is somewhat more complicated.

Note: Most of the examples in this book use manual reference counting. This is intentional. Even though ARC is recommended for new development, there is a lot of legacy code around that does not use it. Even if you are using ARC, you should still understand the retain and release semantics that are implicitly added for you. It's very easy for a programmer who is comfortable with manual reference counting to switch to using ARC, just as it's useful to understand your CPU's instruction set even if you never write any assembly code.

Rather than inserting message sends directly into the code, the compiler front end inserts calls to functions like `objc_retain()` and `objc_release()`. The optimizer then tries to combine or eliminate these calls. In the simple case, these functions do the equivalent of a message send. In some common cases, they are significantly more efficient.

In simple use, you can forget about memory management when using ARC. If you are using a recent version of XCode, then ARC is the default. If you are compiling on the command line or with some other build system, add -fobjc-arc. Now just forget about retaining and releasing objects.

It would be nice if life were that simple. Unfortunately, ARC does have some limitations. More specifically, it formalizes the fuzzy boundary between C memory and Objective-C objects. ARC divides pointers into three categories. Strong pointers follow the same retain/release semantics that we've looked at already. Weak pointers, which we'll look at later, are non-owning references that are automatically zeroed when the object is destroyed. Unsafe unretained pointers are pointers that are ignored by ARC: You are responsible for tracking the lifetime of objects stored in them.

By default, all object pointers in instance variables or on the stack are strong. Object pointers in structures have no default. They must be explicitly marked with the `__unsafe_unretained` ownership qualifier. Even though this is the only permitted ownership for these pointers, it must be explicitly stated to provide a reminder to people reading the code that ARC will ignore it.

Returning Objects via Pointer Arguments

```
__weak id weak;
int writeBack(id *aValue)
{
  *aValue = [NSObject new];
  weak = *aValue;
  return 0;
}

int main(void)
{
  @autoreleasepool
  {
    id object;
    writeBack(&object);
    NSLog(@"Object: %@", object);
    object = nil;
    NSLog(@"Object: %@", weak);
  }
  NSLog(@"Object: %@", weak);
  return 0;
}
```

From: writeback.m

In ARC mode, pointer-to-pointer arguments are somewhat complicated. These are typically used for two things, either passing arrays or returning objects. If you are passing an array down the stack, then you should make sure that you declare it as **const**. This tells the compiler that the callee will not perform any assignments to it, so ARC can pass the array without any complex interaction.

If you are returning an object via this mechanism, ARC produces some fairly complex code. In the example at the start of this section, the call to `writeBack()` generates code that is roughly equivalent to this:

```
id tmp = [object retain];
writeBack(&tmp);
[tmp retain];
[object release];
object = tmp;
```

In `writeBack()`, the new object will be autoreleased before storing it in the temporary value. This means that, at the end of this, `object` contains an owning reference to the new object.

If you declared `object` as `__autoreleasing id`, the generated code is a lot simpler. This will simply autorelease the value initially stored in `object`, which will have no effect because the initial value for all object pointers—even with automatic storage—is `nil`, and will pass the pointer directly and expect the callee to store a non-owning (autoreleased) pointer in it, if it modifies it.

When you run this example, you'll see that the weak reference is only zeroed when the autorelease pool is destroyed. The `writeBack()` function stores an autoreleased object into the passed pointer in all cases, and never releases the passed value. The caller is always responsible for ensuring that the value passed in is a non-owning reference, which it does either by ensuring that it's a pointer to an autoreleased object or a copy of a pointer to a retained object.

If you instead mark the parameter **`out`** (only allowed on method parameters, not C function parameters) then the callee guarantees that it will not read the value, so the compiler will skip the step where it makes a copy of the pointer in `object` before the call.

If you need to pass multiple objects up the stack, then you should probably return an `NSArray` instance. There are some alternatives, but they are sufficiently complex that it's simply not worth the effort: they're easy to get subtly wrong and spend ages debugging, and if you do manage to get it right, you'll probably find it's slower than using an `NSArray`.

If you are passing multiple values down the stack, then you should declare an array type with an explicit ownership qualifier for the parameter, not a pointer type. For example, `__unsafe_unretained id[]` instead of `id*`. This ensures that the writeback mechanism is not used.

Avoiding Retain Cycles

```
- (void)setDelegate: (id)aDelegate
{
  delegate = aDelegate;
}
```

From: ivar.m

The problem with pure reference counting is that it doesn't detect cycles. If you have two objects that retain references to each other, then neither will ever be freed.

In general, this is not a major problem. Objective-C data structures tend to be acyclic, but there are some common cases where cycles are inevitable. The most common is the *delegation pattern*. In this pattern, an object typically implements some mechanisms and delegates policy to another object. Most of AppKit works in this way.

The delegate needs a reference to the object, and the object needs a reference to its delegate. This immediately creates a cycle. The common idiom that addresses this problem is that objects do not retain their delegates. If you pass an object as an argument to a `-setDelegate:` method, you need to make sure that some other object holds a reference to it, or it will be deleted prematurely.

With ARC, you have two choices: You can either mark the instance variable as **__weak** or as `__unsafe_unretained`. The former is safer, because it ensures that there is no chance of a dangling pointer: When the delegate is destroyed, the instance variable will be set to `nil`.

There are two disadvantages of using a weak pointer. The first is portability. Weak pointers work on iOS 5, Mac OS X 10.7, and with GNUstep, but they don't work on older versions of iOS or Mac OS X. The other disadvantage is performance. Every access to a weak pointer goes via a helper function. This checks that the object was not in the process of deallocation and retains it if it is still valid, or returns `nil` if it is not.

In contrast, unsafe unretained pointers are just plain pointers. They are cheap to access and work on any deployment target. Their disadvantage is that you are responsible for ensuring that you don't try to access them after the pointee has been deallocated.

A good compromise is to use weak pointers when debugging and unsafe unretained pointers for deployment. Add debug assert statements checking that the pointer is not `nil` before you send it any messages, and you'll end up with a helpful error, rather than a crash, if you have bugs.

Migrating to ARC

If you are starting a new project in XCode, ARC will be the default and there is little reason not to use it. If you are working on an existing code base, then you are probably using manual reference counting. You can save some long-term development effort by moving to ARC, but it is a little bit more involved than simply flipping a compiler switch.

Clang provides a migration tool that will attempt to rewrite Objective-C code to use ARC. This can be invoked from the command line via the -ccc-arcmt-check and -ccc-arcmt-modify arguments. The first reports any things in the code that cannot be automatically translated. The second actually performs the rewriting—modifying the original file—if there are no errors.

For simple Objective-C code, the migration tool will just work. The most obvious thing that it does is remove all `-retain`, `-release`, and `autorelease` message sends. It will also remove the explicit `[super dealloc]` from a `-dealloc` method: That call is now inserted automatically by the compiler. ARC automatically releases all instance variables, so you will only need to implement `-dealloc` at all if you are freeing `malloc()`'d memory or similar.

If you implement custom reference counting methods, then you will need to delete them. A common reason for this is to prevent accidental

---

Note: ARC actually creates a `-.cxx_destruct` method to handle freeing instance variables. This method was originally created for calling C++ destructors automatically when an object was destroyed. The visible difference of this with ARC is that Objective-C instance variables are now deallocated after `-dealloc` in the root class has finished, not before. In most cases, this should make no difference.

---

deallocation of singletons. This is less important with ARC, because programmer error is less likely to cause an object to be prematurely deleted.

The biggest problems come if you are trying to store Objective-C pointers in C structures. The simplest solution is just not to do that: Use Objective-C objects with public instance variables instead. This lets the compiler handle memory management for the object, as well as the fields.

The only time when using structures referring to Objective-C objects is considered safe is when you are passing them down the stack. The object pointers can then be `__unsafe_unretained` qualified without any problems, as long as they remain valid in the caller.

You will notice that you no longer have any `assign` properties after using the migration tool. These will be rewritten as `unsafe_unretained` or `weak`, depending on whether the deployment target that you've selected supports weak references. You may want to explicitly change some to `unsafe_unretained` if they are used for breaking simple cycles and you find weak references to be a performance problem.

The migration tool will try to insert `__bridge` casts where required, but these are worth checking. These casts are used to move objects in and out of ARC-managed code. In non-ARC Objective-C, you are free to do things like `(void*)someObject`, because object pointers are just C pointers that you can send messages to. In ARC mode, this cast would be ambiguous because the compiler doesn't know what the ownership semantics of `void*` are supposed to be, so it is rejected.

The migration tool will rewrite this as `(__bridge void*)someObject`, but that may not be what you want. We'll look at these casts in more detail in the "Interoperating with C" section.

Autorelease Pools

```
id returnObject(void)
{
  return [[NSObject new] autorelease];
}

int main(void)
{
  @autorelease {
    id object = returnObject();
    [object retain];
  }
  // Object becomes invalid here.
  [object release];
  return 0;
}
```

From: autorelease.m

Aside from cycles, the biggest problem with reference counting is that there are short periods when no object really owns a particular reference. In C, deciding whether the caller or callee is responsible for allocating memory is a problem.

In something like the `sprintf()` function, the caller allocates the space. Unfortunately, the caller doesn't actually know how much space is needed, so the `snprintf()` variant was added to let the callee know how much space is available. This can still cause problems, so the `asprintf()` version was added to let the callee allocate the space.

If the callee is allocating the space, who is responsible for freeing it? The caller, presumably, but because the caller didn't create it, anything that checks for balanced `malloc()` and `free()` calls will fail to spot the leak.

In Objective-C, this problem is even more common. Lots of methods may return temporary objects. If you're returning a temporary object, it needs to be freed, but if you're returning a pointer to an instance variable, it doesn't. You could retain such a pointer first, but then you need to remember to release every single object that is returned from a method. This quickly gets tiresome.

The solution to this problem is the *autorelease pool*. When you send an object an `-autorelease` message, it is added to the currently active `NSAutoreleasePool` instance. When this instance is destroyed, every object added to it is sent a `-release` message.

The `-autorelease` message is a deferred `-release` message. You send it to an object when you no longer need a reference to it but something else might.

If you are using `NSRunLoop`, an autorelease pool will be created at the start of every run loop iteration and destroyed at the end. This means that no temporary objects will be destroyed until the end of the current iteration. If you are doing something that creates a lot of temporary objects, you may wish to create a new autorelease pool, like so:

```
id pool = [NSAutoreleasePool new];
[anObject doSomethingThatCreatesObjects];
[pool drain];
```

Note that you send an autorelease pool a `-drain` message rather than a `release` message when you destroy it. That is because the Objective-C runtime will ignore `-release` messages when you are in garbage collected mode. The `-drain` message in this mode provides a hint to the collector, but does not destroy the pool, when you are in garbage collected mode.

In OS X 10.7, Apple made autorelease pools part of the language. Programs that explicitly reference the `NSAutoreleasePool` class are considered invalid in ARC mode and the compiler will reject them. The replacement is the `@autoreleasepool` construct. This defines a region where an autorelease pool is valid. In non-ARC mode, this will insert exactly the same code as the above snippet. In ARC mode, it inserts calls to the `objc_autoreleasePoolPush()` and `objc_autoreleasePoolPop()` functions, which do something similar.

Using Autoreleased Constructors

```
+ (id)object
{
  return [[[self alloc] init] autorelease];
}
```

From: namedConstructor.m

I said in the last section that objects created with `+alloc` have a retain count of one. In fact, all objects are created with a retain count of one, but objects created with a named constructor, such as `+stringWithFormat:` or `+array`, are also autoreleased.

If you create an object with one of these mechanisms, you must send it a `-retain` message if you want to keep it. If you don't, it will be collected later when the autorelease pool is destroyed.

This is a convention that is important to observe in your own classes. If someone creates an instance of one of your classes with a named constructor, he will expect not to have to release it. A typical named constructor would look something like the one at the start of this section.

Note that, because this is a class method, the **`self`** object will be the class. By sending the `+alloc` message to **`self`** rather than to the class name, this method can work with subclasses automatically.

With ARC, these conventions are formalized in *method families*. Methods that begin `alloc`, `new`, `copy`, or `mutableCopy` return an *owning reference*, a reference that must be released if not stored. Other methods return a non-owning reference, one that has either been autoreleased or is stored somewhere else with a guarantee that it will not be released.

Autoreleasing Objects in Accessors

```
- (NSString*)stringValue
{
  return [[string retain] autorelease];
}
```

From: ivar.m

Another common issue with reference counting, as implemented in Foundation, is that you commonly don't retain objects that you only reference on the stack. Imagine that you have some code like this:

```
NSString *oldString = [anObject stringValue];
[anObject setStringValue: newString];
```

If the `-setStringValue:` method is implemented as I suggested earlier, this code will crash because the object referenced by `oldString` will be deleted when you set the new string value. This is a problem. There are two possible solutions, both involving autorelease pools. One is to autorelease the old value when you set the new one. The other is the definition of the `-stringValue` method from the start of this section.

This ensures that the string will not be accidentally destroyed as a result of anything that the object does. Another common idiom is to substitute a `-copy` message for `-retain`. This is useful if the instance variable might be mutable. If it's immutable, `-copy` will be equivalent to `-retain`. If it's mutable, the caller will get an object that won't change as a result of other messages sent to the object.

Supporting Automatic Garbage Collection

```
$ gcc -c -framework Cocoa -fobjc-gc-only
    collected.m
$ gcc -c -framework Cocoa -fobjc-gc collected.m
```

Starting with OS X 10.5, Apple introduced automatic garbage collection to Objective-C. This can make life easier for programmers, but in most cases it comes with a performance penalty. Apple's collector uses a lot of memory to track live references and therefore is not available on the iPhone. It is also not supported with older versions of OS X and has only limited support with GNUstep, so you should avoid using garbage collection if you want to write portable code.

If you compile your code in garbage collected mode, all `-retain`, `-release`, and `-autorelease` messages will be ignored. The compiler will automatically insert calls to functions in the runtime for every assign operation to memory on the heap.

Code must be compiled with garbage collection support to use the garbage collector. This will insert calls to a set of functions that, on the Mac runtime, are declared in objc-auto.h on any assignment to memory on the heap.

These functions make sure that the garbage collector is aware of the write. These are required because the collector is concurrent. It will run in a background thread and will delete objects when it can no longer find references to them. The collector must be notified of updated pointers, or it might accidentally delete an object that you have just created a reference to.

You have two options when compiling for garbage collection. If you compile with the -fobjc-gc-only flag your code will only support garbage collection. If you compile with the -fobjc-gc flag, the code will support both reference counting and automatic garbage collection. This is useful when you are compiling a framework. You must still remember to add `-retain` and `-release` calls in the correct

```
OBJC_EXPORT BOOL objc_atomicCompareAndSwapGlobal(id predicate, id replacement, volatile id *objectLocation)
    __OSX_AVAILABLE_STARTING(__MAC_10_6, __IPHONE_NA) OBJC_ARC_UNAVAILABLE;
OBJC_EXPORT BOOL objc_atomicCompareAndSwapGlobalBarrier(id predicate, id replacement, volatile id *objectLocation)
    __OSX_AVAILABLE_STARTING(__MAC_10_6, __IPHONE_NA) OBJC_ARC_UNAVAILABLE;
// atomic update of an instance variable
OBJC_EXPORT BOOL objc_atomicCompareAndSwapInstanceVariable(id predicate, id replacement, volatile id *objectLocation)
    __OSX_AVAILABLE_STARTING(__MAC_10_6, __IPHONE_NA) OBJC_ARC_UNAVAILABLE;
OBJC_EXPORT BOOL objc_atomicCompareAndSwapInstanceVariableBarrier(id predicate, id replacement, volatile id *objectLocation)
```

From: objc-auto.h

places, but users of your framework can then use it with or without collection.

Interoperating with C

In garbage collected mode, not all memory is scanned. Anything allocated by `malloc()` is invisible to the garbage collector. If you pass an object pointer as a **`void*`** parameter to a C function, which then stores it in `malloc()`'d memory, it becomes invisible to the collector and may be freed even though there are still references to it.

In ARC mode, the compiler will handle pointers that are on the stack or in instance variables for you automatically, but it won't track pointers in structures or anything else explicitly declared as `__unsafe_unretained`.

Normally, you would send a `-retain` message to an object before storing it on the heap, but that does nothing in garbage collected mode and is forbidden in ARC mode. Instead, you have to use the `CFRetain()` function. This will increment the object's reference count, irrespective of whether the collector is running. The collector will only free objects when their retain count is zero and it cannot find any references to them in traced memory.

When you have finished with a reference that is outside of the collector's scope, you need to call `CFRelease()`.

ARC provides a richer memory model for this kind of operation. Explicit casts from object to non-object pointer types are no longer allowed. They must be replaced by *bridged casts*. Consider the following bit of code in non-ARC mode:

```
void *aPointer = (void*)someObject;
```

In ARC mode, this would create an untracked pointer from a tracked pointer, which is not something that you want to do without thinking. You have three basic options. The first is most commonly used for on-stack variables, or variables pointing to objects that are guaranteed to be referenced elsewhere:

```
void *aPointer = (__bridge void*)someObject;
```

This performs the cast with no transfer of ownership. If all other references to `someObject` are dropped, then `aPointer` becomes a dangling pointer. If the `void*` pointer is going on the heap somewhere and should keep an owning reference to the object, then you should use a retained bridging cast:

```
void *aPointer = (__bridge_retained void*)
    someObject;
```

This will move a single owning reference out of ARC's control. This is roughly equivalent to sending a `-retain` message to `someObject` before it is stored. If you write `(__bridge_retained void*)someObject` with no assignment, then this tells the compiler to retain the object. Doing this is considered very bad style. You should use the inverse operation when casting back to an object pointer:

```
id anotherObjectPointer = (__bridge_transfer
    id)aPointer;
aPointer = NULL;
```

This transfers an owning reference into ARC's control. ARC is now responsible for releasing the object, so it is important to remember to zero the C pointer. If you are not taking ownership of the pointer, then you should use a simple `__bridge` cast.

Understanding Object Destruction

```
@interface Example : NSObject
{
  void *cPointer;
  id objectPointer;
}
@end
@implementation Example
- (void)finalize
{
  if (NULL != cPointer) { free(cPointer); }
  [super finalize];
}
- (void)dealloc
{
  if (NULL != cPointer) { free(cPointer); }
#if !__has_feature(objc_arc)
  [objectPointer release];
  [super dealloc];
#endif
}
@end
```

From: dealloc.m

There are three methods that are invoked during the destruction of an object, depending on the mode. One will run every time, but cannot be written by you. The `-.cxx_destruct` method is always called by the Objective-C runtime and handles the destruction of any fields that the compiler is responsible for cleaning up. This includes C++ objects in Objective-C++ mode and Objective-C object pointers in ARC mode.

The other two methods are `-finalize` and `-dealloc`. In garbage collected mode, you do not need to do anything to relinquish references to Objective-C objects, but you do still need to clean up any resources that are not managed by the garbage collector. This includes closing file handles, freeing memory allocated with `malloc()`, and so on. If your class has instance variables that need this kind of manual cleanup, then you should declare a `-finalize` method.

Note: The garbage collector will typically call `-finalize` methods in a special cleanup thread. This means that `-finalize` methods must be thread safe. If they refer to global resources, then they must ensure that doing so does not introduce race conditions.

If you are not using garbage collection, then you should do cleanup in a `-dealloc` method. The contents of this method depend on whether or not you are using ARC. Traditionally, `-dealloc` methods were full of `-release` message sends to every instance variable that the class declared. With ARC, this is not required. ARC will relinquish any owning references to objects in

-.cxx_destruct, so you only need to clean up non-object instance variables in -dealloc.

In both GC and manual retain/release mode, you should forward the message to the superclass, calling [super dealloc] or [super finalize] as appropriate. In ARC mode, explicitly calling -dealloc is not permitted. Instead, ARC will insert a [super dealloc] call at the end of your -dealloc method in any non-root class.

Note that the example at the start of this section is overly complicated. In real code, you are unlikely to need to support both ARC and manual retain/release modes. Both can interoperate in the same program. The only reason to support non-ARC mode is if you wish to support legacy compilers.

Using Weak References

```
__weak id weak;

int main(void)
{
  id obj = [NSObject new];
  weak = obj;
  obj = nil;
  objc_collect(OBJC_FULL_COLLECTION);
  fprintf(stderr, "Weak reference: %p\n", weak);
  return 0;
}
```

From: weak.m

One of the nicest things about Apple's garbage collection implementation is the existence of *zeroing weak references*. Pointers that are not retained are often referred to as "weak" in Objective-C documentation that predates the garbage collector. These are references that are allowed to persist beyond the lifetime of the object. Unfortunately, there is no automatic way of telling whether they are still valid.

Weak references were found to be so useful with the garbage collector that they are now also supported by ARC, with slightly different semantics. The ARC implementation provided for backwards compatibility does not support weak references, so they can only be used with ARC on Apple platforms if you restrict yourself to OS X 10.7 or iOS 5 and later.

Note: Weak references in a reference counted environment, such as those referencing delegates, are commonly used to eliminate retain cycles. This is not needed in a tracing environment, so you can use strong references for pointers to delegates and anywhere else you might have a retain cycle.

If you declare an object pointer **`__weak`**, you get a zeroing weak reference. This is not counted by the garbage collector when determining if an object is still live and does not increment the object's reference count when assigned to in ARC mode. If all of the references to an object are weak, it can be destroyed. Afterwards, reading the weak references will return `nil`.

Weak references are most commonly used in connection with things such as notifications. You will keep a weak reference to an object and keep sending it messages as long as it is referenced elsewhere, then you can have it cleaned up automatically later.

Cocoa now comes with some collections that let you store weak references. Older versions of the Foundation framework provide `NSMapTable` and `NSHashTable` as opaque C types, with a set of C functions to use them. These interfaces are still available, but with 10.5, Apple made these two C types into classes.

The `NSMapTable` type is a general form of `NSDictionary` that can be used to store any pointer-sized types as both values and keys. With garbage collection, you can use this class to store mappings to and from strong or weak object pointers as well. This is useful for things such as `NSNotificationCenter`, so that objects can be collected while they are still registered to receive notifications and can be automatically removed from the notification center when this happens.

The example at the start of this section shows an important difference between ARC and GC modes. If you compile and run this example with garbage collection enabled, you will probably see it print an object address. This is because the garbage collector will still see stale temporaries while scanning the stack.

In contrast, this will always print 0 in ARC mode. ARC, unlike GC, is entirely deterministic. Assigning `nil` to the strong pointer decrements the object's reference count and triggers deallocation. The weak pointer is zeroed before deallocation begins and so is guaranteed to be zero by the time the `fprintf()` is reached.

Allocating Scanned Memory

```
id *buffer =
  NSAllocateCollectable(
      10 * sizeof(id),
      NSScannedOption);
```

From: gc.m

If you allocate memory with `malloc()`, it is invisible to the garbage collector. This is a problem if you want, for example, something like a C array containing objects. We've already looked at one solution to this. You can call `CFRetain()` on the object you are about to store and `CFRelease()` on the old value, and then swap them over.

This is not ideal, although it will work. The other option is to allocate a region of memory from the garbage collector. The `NSAllocateCollectable()` function is similar to `malloc()`, but with two important differences.

The first is that the memory that it returns is garbage collected. There is no corresponding `NSFreeCollectable()` function. When the last pointer to the buffer disappears, the buffer will be collected.

Note: The Apple collector does not support interior references, so you must make sure you keep a pointer to the start of the region. Pointers to somewhere in the middle of the buffer will not prevent it from being freed.

The second difference is that the second parameter to this function defines the kind of memory that you want. If you are going to be using the buffer for storing C types, you can just pass zero here. If you pass `NSScannedOption`, the returned buffer will be scanned as a possible location of object pointers, as well as pointers to other memory regions returned by `NSAllocateCollectable()`.

Index

I

J

K

L