Advanced Memory Management Programming Guide

Performance

2011-09-28

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

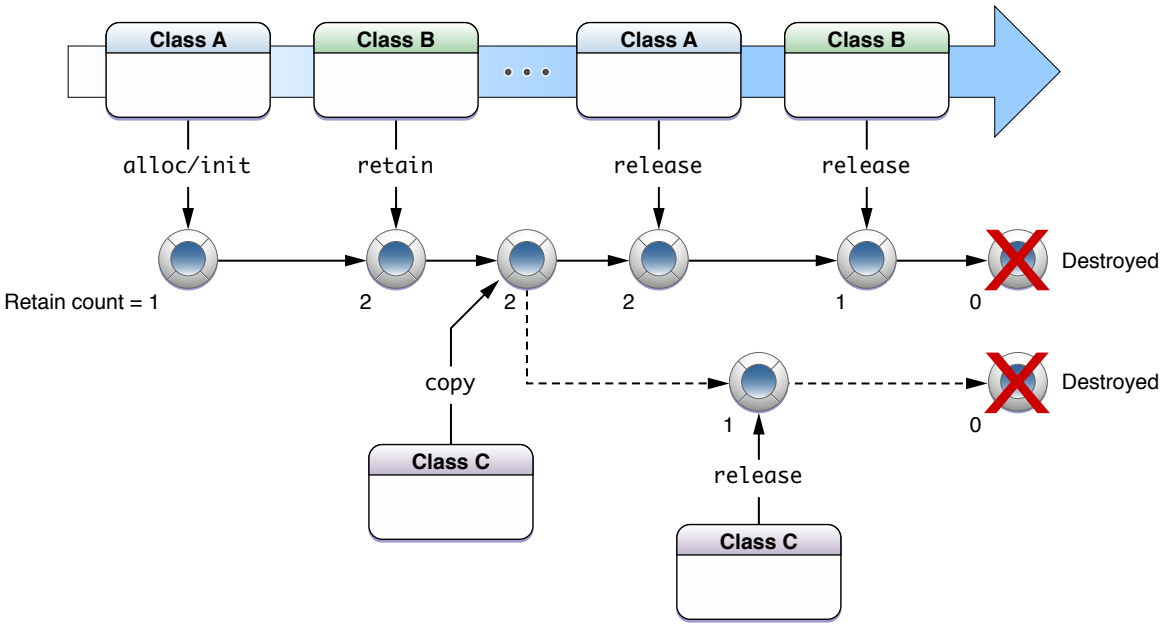
Contents

Figures

About Memory Management

Application memory management is the process of allocating memory during your program's runtime, using it, and freeing it when you are done with it. A well-written program uses as little memory as possible. In Objective-C, it can also be seen as a way of distributing ownership of limited memory resources among many pieces of data and code. When you have finished working through this guide, you will have the knowledge you need to manage your application's memory by explicitly managing the life cycle of objects and freeing them when they are no longer needed.

Although memory management is typically considered at the level of an individual object, your goal is actually to manage object graphs. You want to make sure that you have no more objects in memory than you actually need.



At a Glance

Objective-C provides three methods of application memory management.

1. In the method described in this guide, referred to as "manual retain-release" or **MRR**, you explicitly manage memory by keeping track of objects you own. This is implemented using a model, known as reference counting, that the Foundation class `NSObject` provides in conjunction with the runtime environment.

2. In Automatic Reference Counting, or **ARC**, the system uses the same reference counting system as MRR, but it inserts the appropriate memory management method calls for you at compile-time. You are strongly encouraged to use ARC for new projects. If you use ARC, there is typically no need to understand the underlying implementation described in this document, although it may in some situations be helpful. For more about ARC, see *Transitioning to ARC Release Notes*.

3. In **garbage collection**, the system automatically keeps track of what objects own what other objects. It then automatically frees (or garbage-collects) objects that are no longer referenced. It uses a different mechanism than that employed by MMR and ARC, and is supported only in the runtime environment of Mac OS X, not iOS. For more about garbage collection, see *Garbage Collection Programming Guide*.

If you plan on writing code for iOS, you must use explicit memory management (the subject of this guide). Further, if you plan on writing library routines, plug-ins, or shared code—code that might be loaded into either a garbage-collection or non-garbage-collection process—you want to write your code using the memory-management techniques described throughout this guide. (Make sure that you then test your code in Xcode, with garbage collection disabled and enabled.)

Good Practices Prevent Memory-Related Problems

There are two main kinds or problem that result from incorrect memory management:

- Freeing or overwriting data that is still in use

  This causes memory corruption, and typically results in your application crashing, or worse, corrupted user data.

- Not freeing data that is no longer in use causes memory leaks

  A memory leak is where allocated memory is not freed, even though it is never used again. Leaks cause your application to use ever-increasing amounts of memory, which in turn may result in poor system performance or (in iOS) your application being terminated.

Thinking about memory management from the perspective of reference counting, however, is frequently counterproductive, because you tend to consider memory management in terms of the implementation details rather than in terms of your actual goals. Instead, you should think of memory management from the perspective of object ownership and object graphs.

Cocoa uses a straightforward naming convention to indicate when you own an object returned by a method.

See "Memory Management Policy" (page 11).

Although the basic policy is straightforward, there are some practical steps you can take to make managing memory easier, and to help to ensure your program remains reliable and robust while at the same time minimizing its resource requirements.

See "Practical Memory Management" (page 15).

Autorelease pools provide a mechanism whereby you can send an object a "deferred" `release` message. This is useful in situations where you want to relinquish ownership of an object, but want to avoid the possibility of it being deallocated immediately (such as when you return an object from a method). There are occasions when you might use your own autorelease pools.

See "Using Autorelease Pools" (page 23).

Use Analysis Tools to Debug Memory Problems

To identify problems with your code at compile time, you can use the Clang Static Analyzer that is built into Xcode.

If memory management problems do nevertheless arise, there are other tools and techniques you can use to identify and diagnose the issues.

- Many of the tools and techniques are described in Technical Note TN2239, *iOS Debugging Magic*, in particular the use of `NSZombie` to help find over-released object.
- You can use Instruments to track reference counting events and look for memory leaks. See "Viewing and Analyzing Trace Data".

Memory Management Policy

The basic model used for memory management in a reference-counted environment is provided by a combination of methods defined in the `NSObject` protocol and a standard method naming convention. The `NSObject` class also defines a method, `dealloc`, that is invoked automatically when an object is deallocated. This article describes all the basic rules you need to know to manage memory correctly in a Cocoa program, and provides some examples of correct usage.

Basic Memory Management Rules

The memory management model is based on object ownership. Any object may have one or more owners. As long as an object has at least one owner, it continues to exist. If an object has no owners, the runtime system destroys it automatically. To make sure it is clear when you own an object and when you do not, Cocoa sets the following policy:

- **You own any object you create**

  You create an object using a method whose name begins with "alloc", "new", "copy", or "mutableCopy" (for example, `alloc`, `newObject`, or `mutableCopy`).

- **You can take ownership of an object using retain**

  A received object is normally guaranteed to remain valid within the method it was received in, and that method may also safely return the object to its invoker. You use `retain` in two situations: (1) In the implementation of an accessor method or an `init` method, to take ownership of an object you want to store as a property value; and (2) To prevent an object from being invalidated as a side-effect of some other operation (as explained in "Avoid Causing Deallocation of Objects You're Using" (page 18)).

- **When you no longer need it, you must relinquish ownership of an object you own**

  You relinquish ownership of an object by sending it a `release` message or an `autorelease` message. In Cocoa terminology, relinquishing ownership of an object is therefore typically referred to as "releasing" an object.

- **You must not relinquish ownership of an object you do not own**

  This is just corollary of the previous policy rules, stated explicitly.

A Simple Example

To illustrate the policy, consider the following code fragment:

```
{
    Person *aPerson = [[Person alloc] init];
```

```
    // ...
    NSString *name = person.fullName;
    // ...
    [aPerson release];
}
```

The Person object is created using the `alloc` method, so it is subsequently sent a `release` message when it is no longer needed. The person's name is not retrieved using any of the owning methods, so it is not sent a `release` message. Notice, though, that the example uses `release` rather than `autorelease`.

Use autorelease to Send a Deferred release

You use `autorelease` when you need to send a deferred `release` message—typically when returning an object from a method. For example, you could implement the `fullName` method like this:

```
- (NSString *)fullName {
    NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@",
                                        self.firstName, self.lastName]
autorelease];
    return string;
}
```

You own the string returned by `alloc`. To abide by the memory management rules, you must relinquish ownership of the string before you lose the reference to it. If you use `release`, however, the string will be deallocated before it is returned (and the method would return an invalid object). Using `autorelease`, you signify that you want to relinquish ownership, but you allow the caller of the method to use the returned string before it is deallocated.

You could also implement the `fullName` method like this:

```
- (NSString *)fullName {
    NSString *string = [NSString stringWithFormat:@"%@ %@",
                                self.firstName, self.lastName];
    return string;
}
```

Following the basic rules, you don't own the string returned by `stringWithFormat:`, so you can safely return the string from the method.

By way of contrast, *the following implementation is wrong*:

```
- (NSString *)fullName {
    NSString *string = [[NSString alloc] initWithFormat:@"%@ %@",
                                        self.firstName, self.lastName];
    return string;
}
```

According to the naming convention, there is nothing to denote that the caller of the `fullName` method owns the returned string. The caller therefore has no reason to release the returned string, and it will thus be leaked.

You Don't Own Objects Returned by Reference

Some methods in Cocoa specify that an object is returned by reference (that is, they take an argument of type `ClassName **` or `id *`). A common pattern is to use an `NSError` object that contains information about an error if one occurs, as illustrated by `initWithContentsOfURL:options:error:` (`NSData`) and `initWithContentsOfFile:encoding:error:` (`NSString`).

In these cases, the same rules apply as have already been described. When you invoke any of these methods, you do not create the `NSError` object, so you do not own it. There is therefore no need to release it, as illustrated in this example:

```
NSString *fileName = <#Get a file name#>;
NSError *error = nil;
NSString *string = [[NSString alloc] initWithContentsOfFile:fileName
                        encoding:NSUTF8StringEncoding error:&error];
if (string == nil) {
    // Deal with error...
}
// ...
[string release];
```

Implement dealloc to Relinquish Ownership of Objects

The `NSObject` class defines a method, `dealloc`, that is invoked automatically when an object has no owners and its memory is reclaimed—in Cocoa terminology it is "freed" or "deallocated.". The role of the `dealloc` method is to free the object's own memory, and to dispose of any resources it holds, including ownership of any object instance variables.

The following example illustrates how you might implement a `dealloc` method for a Person class:

```
@interface Person : NSObject {}
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (assign, readonly) NSString *fullName;
@end

@implementation Person
@synthesize firstName=_firstName, lastName=_lastName;
// ...
- (void)dealloc
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end
```

Important: You should never invoke another object's `dealloc` method directly.

You must invoke the superclass's implementation at the *end* of your implementation.

You should not tie management of system resources to object lifetimes; see "Don't Use dealloc to Manage Scarce Resources" (page 19).

When an application terminates, objects may not be sent a `dealloc` message. Because the process's memory is automatically cleared on exit, it is more efficient simply to allow the operating system to clean up resources than to invoke all the memory management methods.

Core Foundation Uses Similar but Different Rules

There are similar memory management rules for Core Foundation objects (see *Memory Management Programming Guide for Core Foundation*). The naming conventions for Cocoa and Core Foundation, however, are different. In particular, Core Foundation's Create Rule (see "The Create Rule" in *Memory Management Programming Guide for Core Foundation*) does not apply to methods that return Objective-C objects. For example, in the following code fragment, you are *not* responsible for relinquishing ownership of `myInstance`:

```
MyClass *myInstance = [MyClass createInstance];
```

Practical Memory Management

Although the fundamental concepts described in "Memory Management Policy" (page 11) are straightforward, there are some practical steps you can take to make managing memory easier, and to help to ensure your program remains reliable and robust while at the same time minimizing its resource requirements.

Use Accessor Methods to Make Memory Management Easier

If your class has a property that is an object, you must make sure that any object that is set as the value is not deallocated while you're using it. You must therefore claim ownership of the object when it is set. You must also make sure you then relinquish ownership of any currently-held value.

Sometimes it might seem tedious or pedantic, but if you use accessor methods consistently, the chances of having problems with memory management decrease considerably. If you are using `retain` and `release` on instance variables throughout your code, you are almost certainly doing the wrong thing.

Consider a Counter object whose count you want to set.

```
@interface Counter : NSObject {
    NSNumber *_count;
}
@property (nonatomic, retain) NSNumber *count;
@end;
```

The property declares two accessor methods. Typically, you should ask the compiler to synthesize the methods; however, it's instructive to see how they might be implemented.

In the "get" accessor, you just return the instance variable, so there is no need for `retain` or `release`:

```
- (NSNumber *)count {
    return _count;
}
```

In the "set" method, if everyone else is playing by the same rules you have to assume the new count may be disposed of at any time so you have to take ownership of the object—by sending it a `retain` message—to ensure it won't be. You must also relinquish ownership of the old count object here by sending it a `release` message. (Sending a message to `nil` is allowed in Objective-C, so the implementation will still work if `_count` hasn't yet been set.) You must send this after `[newCount retain]` in case the two are the same object—you don't want to inadvertently cause it to be deallocated.

```
- (void)setCount:(NSNumber *)newCount {
    [newCount retain];
    [_count release];
    // Make the new assignment.
    _count = newCount;
}
```

Use Accessor Methods to Set Property Values

Suppose you want to implement a method to reset the counter. You have a couple of choices. The first implementation creates the `NSNumber` instance with `alloc`, so you balance that with a `release`.

```
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [self setCount:zero];
    [zero release];
}
```

The second uses a convenience constructor to create a new `NSNumber` object. There is therefore no need for `retain` or `release` messages

```
- (void)reset {
    NSNumber *zero = [NSNumber numberWithInteger:0];
    [self setCount:zero];
}
```

Note that both use the set accessor method.

The following will almost certainly work correctly for simple cases, but as tempting as it may be to eschew accessor methods, doing so will almost certainly lead to a mistake at some stage (for example, when you forget to retain or release, or if the memory management semantics for the instance variable change).

```
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [_count release];
    _count = zero;
}
```

Note also that if you are using key-value observing, then changing the variable in this way is not KVO compliant.

Don't Use Accessor Methods in Initializer Methods and dealloc

The only places you shouldn't use accessor methods to set an instance variable are in initializer methods and `dealloc`. To initialize a counter object with a number object representing zero, you might implement an `init` method as follows:

```
- init {
    self = [super init];
    if (self) {
        _count = [[NSNumber alloc] initWithInteger:0];
    }
    return self;
}
```

To allow a counter to be initialized with a count other than zero, you might implement an `initWithCount:` method as follows:

```
- initWithCount:(NSNumber *)startingCount {
    self = [super init];
    if (self) {
        _count = [startingCount copy];
    }
```

```
    return self;
}
```

Since the Counter class has an object instance variable, you must also implement a `dealloc` method. It should relinquish ownership of any instance variables by sending them a `release` message, and ultimately it should invoke super's implementation:

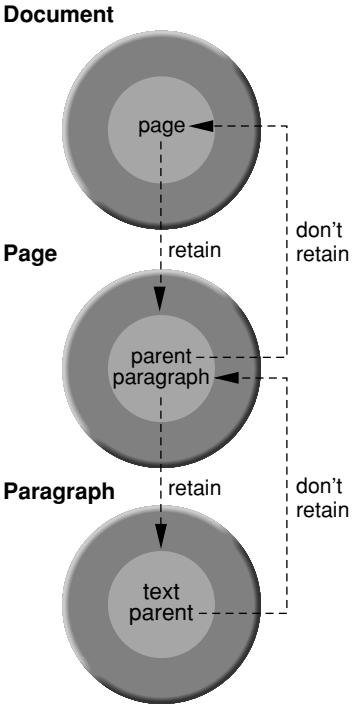
```
- (void)dealloc {
    [_count release];
    [super dealloc];
}
```

Use Weak References to Avoid Retain Cycles

Retaining an object creates a **strong reference** to that object. An object cannot be deallocated until all of its strong references are released. A problem, known as a **retain cycle**, can therefore arise if two objects may have cyclical references—that is, they have a strong reference to each other (either directly, or through a chain of other objects each with a strong reference to the next leading back to the first).

The object relationships shown in Figure 1 (page 17) illustrate a potential retain cycle. The Document object has a Page object for each page in the document. Each Page object has a property that keeps track of which document it is in. If the Document object has a strong reference to the Page object and the Page object has a strong reference to the Document object, neither object can ever be deallocated. The Document's reference count cannot become zero until the Page object is released, and the Page object won't be released until the Document object is deallocated.

Figure 1 An illustration of cyclical references



The solution to the problem of retain cycles is to use weak references. A **weak reference** is a non-owning relationship where the source object does not retain the object to which it has a reference.

To keep the object graph intact, however, there must be strong references somewhere (if there were only weak references, then the pages and paragraphs might not have any owners and so would be deallocated). Cocoa establishes a convention, therefore, that a "parent" object should maintain strong references to its "children," and that the children should have weak references to their parents.

So, in Figure 1 (page 17) the document object has a strong reference to (retains) its page objects, but the page object has a weak reference to (does not retain) the document object.

Examples of weak references in Cocoa include, but are not restricted to, table data sources, outline view items, notification observers, and miscellaneous targets and delegates.

You need to be careful about sending messages to objects for which you hold only a weak reference. If you send a message to an object after it has been deallocated, your application will crash. You must have well-defined conditions for when the object is valid. In most cases, the weak-referenced object is aware of the other object's weak reference to it, as is the case for circular references, and is responsible for notifying the other object when it deallocates. For example, when you register an object with a notification center, the notification center stores a weak reference to the object and sends messages to it when the appropriate notifications are posted. When the object is deallocated, you need to unregister it with the notification center to prevent the notification center from sending any further messages to the object, which no longer exists. Likewise, when a delegate object is deallocated, you need to remove the delegate link by sending a `setDelegate:` message with a `nil` argument to the other object. These messages are normally sent from the object's `dealloc` method.

Avoid Causing Deallocation of Objects You're Using

Cocoa's ownership policy specifies that received objects should typically remain valid throughout the scope of the calling method. It should also be possible to return a received object from the current scope without fear of it being released. It should not matter to your application that the getter method of an object returns a cached instance variable or a computed value. What matters is that the object remains valid for the time you need it.

There are occasional exceptions to this rule, primarily falling into one of two categories.

1. When an object is removed from one of the fundamental collection classes.

   ```
   heisenObject = [array objectAtIndex:n];
   [array removeObjectAtIndex:n];
   // heisenObject could now be invalid.
   ```

   When an object is removed from one of the fundamental collection classes, it is sent a `release` (rather than `autorelease`) message. If the collection was the only owner of the removed object, the removed object (`heisenObject` in the example) is then immediately deallocated.

2. When a "parent object" is deallocated.

   ```
   id parent = <#create a parent object#>;
   // ...
   heisenObject = [parent child] ;
   [parent release]; // Or, for example: self.parent = nil;
   // heisenObject could now be invalid.
   ```

In some situations you retrieve an object from another object, and then directly or indirectly release the parent object. If releasing the parent causes it to be deallocated, and the parent was the only owner of the child, then the child (`heisenObject` in the example) will be deallocated at the same time (assuming that it is sent a `release` rather than an `autorelease` message in the parent's `dealloc` method).

To protect against these situations, you retain `heisenObject` upon receiving it and you release it when you have finished with it. For example:

```
heisenObject = [[array objectAtIndex:n] retain];
[array removeObjectAtIndex:n];
// Use heisenObject...
[heisenObject release];
```

Don't Use dealloc to Manage Scarce Resources

You should typically not manage scarce resources such as file descriptors, network connections, and buffers or caches in a `dealloc` method. In particular, you should not design classes so that `dealloc` will be invoked when you think it will be invoked. Invocation of `dealloc` might be delayed or sidestepped, either because of a bug or because of application tear-down.

Instead, if you have a class whose instances manage scarce resources, you should design your application such that you know when you no longer need the resources and can then tell the instance to "clean up" at that point. You would typically then release the instance, and `dealloc` would follow, but you will not suffer additional problems if it does not.

Problems may arise if you try to piggy-back resource management on top of `dealloc`. For example:

1. Order dependencies on object graph tear-down.

   The object graph tear-down mechanism is inherently non-ordered. Although you might typically expect—and get—a particular order, you are introducing fragility. If an object falls in an autorelease pool unexpectedly, the tear-down order may change, which may lead to unexpected results.

2. Non-reclamation of scarce resources.

   Memory leaks are bugs that should be fixed, but they are generally not immediately fatal. If scarce resources are not released when you expect them to be released, however, you may run into more serious problems. If your application runs out of file descriptors, for example, the user may not be able to save data.

3. Cleanup logic being executed on the wrong thread.

   If an object falls into an autorelease pool at an unexpected time, it will be deallocated on whatever thread's pool it happens to be in. This can easily be fatal for resources that should only be touched from one thread.

Collections Own the Objects They Contain

When you add an object to a collection (such as an array, dictionary, or set), the collection takes ownership of it. The collection will relinquish ownership when the object is removed from the collection or when the collection is itself released. Thus, for example, if you want to create an array of numbers you might do either of the following:

```
NSMutableArray *array = <#Get a mutable array#>;
NSUInteger i;
// ...
for (i = 0; i < 10; i++) {
    NSNumber *convenienceNumber = [NSNumber numberWithInteger:i];
    [array addObject:convenienceNumber];
}
```

In this case, you didn't invoke `alloc`, so there's no need to call `release`. There is no need to retain the new numbers (`convenienceNumber`), since the array will do so.

```
NSMutableArray *array = <#Get a mutable array#>;
NSUInteger i;
// ...
for (i = 0; i < 10; i++) {
    NSNumber *allocedNumber = [[NSNumber alloc] initWithInteger: i];
    [array addObject:allocedNumber];
    [allocedNumber release];
}
```

In this case, you *do* need to send `allocedNumber` a `release` message within the scope of the `for` loop to balance the `alloc`. Since the array retained the number when it was added by `addObject:`, it will not be deallocated while it's in the array.

To understand this, put yourself in the position of the person who implemented the collection class. You want to make sure that no objects you're given to look after disappear out from under you, so you send them a `retain` message as they're passed in. If they're removed, you have to send a balancing `release` message, and any remaining objects should be sent a `release` message during your own `dealloc` method.

Ownership Policy Is Implemented Using Retain Counts

The ownership policy is implemented through reference counting—typically called "retain count" after the `retain` method. Each object has a retain count.

- When you create an object, it has a retain count of 1.
- When you send an object a `retain` message, its retain count is incremented by 1.
- When you send an object a `release` message, its retain count is decremented by 1.

  When you send an object a `autorelease` message, its retain count is decremented by 1 at some stage in the future.
- If an object's retain count is reduced to zero, it is deallocated.

Important: Typically there should be no reason to explicitly ask an object what its retain count is (see `retainCount`). The result is often misleading, as you may be unaware of what framework objects have retained an object in which you are interested. In debugging memory management issues, you should be concerned only with ensuring that your code adheres to the ownership rules.

Using Autorelease Pools

Autorelease pools provide a mechanism whereby you can send an object a "deferred" `release` message. This is useful in situations where you want to relinquish ownership of an object, but want to avoid the possibility of it being deallocated immediately (such as when you return an object from a method). Typically, you don't need to create your own autorelease pools, but there are some situations in which either you must or it is beneficial to do so.

About Autorelease Pools

An autorelease pool is an instance of `NSAutoreleasePool` that "contains" other objects that have received an `autorelease` message. When the autorelease pool is deallocated, it sends a `release` message to each of those objects. An object can be put into an autorelease pool several times; an object receives a `release` message for each time it was put into the pool (sent an `autorelease` message).

Autorelease pools are arranged in a stack, although they are commonly referred to as being "nested." If you create a new autorelease pool, it is added to the top of the stack. When pools are deallocated, they are removed from the stack. When an object is sent an `autorelease` message, it is added to the current topmost pool for the current thread.

Cocoa always expects there to be an autorelease pool available. If a pool is not available, autoreleased objects do not get released and your application leaks memory. If you send an `autorelease` message when a pool is not available, Cocoa logs a suitable error message. The AppKit and UIKit frameworks automatically create a pool at the beginning of each event-loop iteration, such as a mouse down event or a tap, and drain it at the end. Therefore you typically do not have to create a pool, or even see the code that is used to create one. There are, however, three occasions when you might use your own autorelease pools:

- If you are writing a program that is not based on a UI framework, such as a command-line tool.
- If you write a loop that creates many temporary objects.

  You may create an autorelease pool inside the loop to dispose of those objects before the next iteration. Using an autorelease pool in the loop helps to reduce the maximum memory footprint of the application.
- If you spawn a secondary thread.

  You must create your own autorelease pool as soon as the thread begins executing; otherwise, your application will leak objects. (See "Autorelease Pools and Threads" (page 25) for details.)

You create an `NSAutoreleasePool` object with the usual `alloc` and `init` messages, and you dispose of it with `drain`. An exception is raised if you send `autorelease` or `retain` to an autorelease pool. To understand the difference between `release` or `drain`, see "Garbage Collection" (page 26). An autorelease pool should always be drained in the same context (such as the invocation of a method or function, or the body of a loop) in which it was created.

Autorelease pools are used "inline." *There should typically be no reason why you should make an autorelease pool an instance variable of an object.*

Use Local Autorelease Pools to Reduce Peak Memory Footprint

Many programs create temporary objects that are autoreleased. These objects add to the program's memory footprint until the pool is drained. You can use local autorelease pools to help reduce peak memory footprint. When your pool is drained, the temporary objects are released, which typically results in their deallocation thereby reducing the program's memory footprint.

The following example shows how you might use a local autorelease pool in a for loop.

```
NSArray *urls = <# An array of file URLs #>;
for (NSURL *url in urls) {

    NSAutoreleasePool *loopPool = [[NSAutoreleasePool alloc] init];

    NSError *error = nil;
    NSString *fileContents = [[[NSString alloc] initWithContentsOfURL:url
                                          encoding:NSUTF8StringEncoding
error:&error] autorelease];

    /* Process the string, creating and autoreleasing more objects. */

    [loopPool drain];
}
```

The `for` loop processes one file at a time. An `NSAutoreleasePool` object is created at the beginning of this loop and drained at the end. Therefore, any object sent an `autorelease` message inside the loop (such as `fileContents`) is added to `loopPool`, and when `loopPool` is drained at the end of the loop those objects are released.

After an autorelease pool is deallocated, you should regard any object that was autoreleased while that pool was active as "disposed of." Do not send a message to that object or return it to the invoker of your method. If you must use a temporary object beyond an autorelease context, you can do so by sending a `retain` message to the object within the context and then send it `autorelease` after the pool has been drained, as illustrated in this example:

```
- (id)findMatchingObject:(id)anObject {

    id match = nil;
    while (match == nil) {
        NSAutoreleasePool *subPool = [[NSAutoreleasePool alloc] init];

        /* Do a search that creates a lot of temporary objects. */
        match = [self expensiveSearchForObject:anObject];

        if (match != nil) {
            [match retain]; /* Keep match around. */
        }
        [subPool drain];
    }

    return [match autorelease];   /* Let match go and return it. */
```

```
}
```

By sending `retain` to `match` while `subpool` is in effect and by sending `autorelease` to it after `subpool` has been drained, `match` is effectively moved from `subpool` to the pool that was previously active. This extends the lifetime of `match` and allows it to receive messages outside the loop and be returned to the invoker of `findMatchingObject:`.

Autorelease Pools and Threads

Each thread in a Cocoa application maintains its own stack of `NSAutoreleasePool` objects. When a thread terminates, it automatically releases all of the autorelease pools associated with itself. If you are writing a Foundation-only application or if you detach a thread, you need to create your own autorelease pool.

If your application or thread is long-lived and potentially generates a lot of autoreleased objects, you should periodically destroy and create autorelease pools (like the Kit does on the main thread); otherwise, autoreleased objects accumulate and your memory footprint grows. If your detached thread does not make Cocoa calls, you do not need to create an autorelease pool.

Note: If you create secondary threads using the POSIX thread APIs instead of `NSThread`, you cannot use Cocoa—including `NSAutoreleasePool`—unless Cocoa is in multithreading mode. Cocoa enters multithreading mode only after detaching its first `NSThread` object. To use Cocoa on secondary POSIX threads, your application must first detach at least one `NSThread` object, which can immediately exit. You can test whether Cocoa is in multithreading mode with the `NSThread` class method `isMultiThreaded`.

Scope of Autorelease Pools and Implications of Nested Autorelease Pools

It is common to speak of autorelease pools as being nested, but you can also think of nested autorelease pools as being on a stack, with the "innermost" autorelease pool being on top of the stack. Each thread in a program maintains a stack of autorelease pools. When you create an autorelease pool, it is pushed onto the top of the current thread's autorelease pool stack. When an object is sent an `autorelease` message—or when it is passed as the argument to the `addObject:` method—it is always put in the autorelease pool at the top of the stack.

The scope of an autorelease pool is therefore defined by its position in the stack. The topmost pool is the pool to which autoreleased objects are added. If another pool is created, the current topmost pool goes out of scope until the new pool is drained (at which point the original pool once again becomes the topmost pool). It goes out of scope permanently when it is itself drained.

If you drain an autorelease pool that is not the top of the stack, all (unreleased) autorelease pools above it on the stack are drained (and all their objects sent appropriate `release` messages). If you neglect to send `drain` to an autorelease pool when you are finished with it (something not recommended), the pool is drained when one of the autorelease pools in which it nests is drained.

This behavior has implications for exceptional conditions. If an exception occurs, and the thread suddenly transfers out of the current context, the pool associated with that context is drained. However, if that pool is not the top pool on the thread's stack, all the pools above the drained pool are also drained (releasing all

their objects in the process). The top autorelease pool on the thread's stack then becomes the pool previously underneath the drained pool associated with the exceptional condition. Because of this behavior, exception handlers do not need to release objects that were sent `autorelease`. Neither is it necessary or even desirable for an exception handler to send `release` to its autorelease pool, unless the handler is re-raising the exception.

Garbage Collection

Although the garbage collection system (described in *Garbage Collection Programming Guide*) does not use autorelease pools, they can be useful in providing hints to the collector if you are developing a hybrid framework (that is, one that may be used in garbage-collected and reference-counted environments). The points where autorelease pools are released are typically also points at which it might be useful to hint to the garbage collector that collection is likely to be warranted.

In a garbage collected environment, `release` is a no-op (a do-nothing instruction). `NSAutoreleasePool` therefore provides a `drain` method that in a reference-counted environment behaves the same as calling `release`, but which in a garbage collected environment triggers garbage collection (if the memory allocated since the last collection is greater than the current threshold). Typically, therefore, you should use `drain` rather than `release` to dispose of an autorelease pool.

Document Revision History

This table describes the changes to *Advanced Memory Management Programming Guide*.

Date	Notes
2011-09-28	Updated to reflect new status as a consequence of the introduction of ARC.
2011-03-24	Major revision for clarity and conciseness.
2010-12-21	Clarified the naming rule for mutable copy.
2010-06-24	Minor rewording to memory management fundamental rule, to emphasize simplicity. Minor additions to Practical Memory Management.
2010-02-24	Updated the description of handling memory warnings for iOS 3.0; partially rewrote "Object Ownership and Disposal."
2009-10-21	Augmented section on accessor methods in Practical Memory Management.
2009-08-18	Added links to related concepts.
2009-07-23	Updated guidance for declaring outlets on Mac OS X.
2009-05-06	Corrected typographical errors.
2009-03-04	Corrected typographical errors.
2009-02-04	Updated "Nib Objects" article.
2008-11-19	Added section on use of autorelease pools in a garbage collected environment.
2008-10-15	Corrected missing image.
2008-02-08	Corrected a broken link to the "Carbon-Cocoa Integration Guide."
2007-12-11	Corrected typographical errors.
2007-10-31	Updated for Mac OS X v10.5. Corrected minor typographical errors.
2007-06-06	Corrected minor typographical errors.
2007-05-03	Corrected typographical errors.
2007-01-08	Added article on memory management of nib files.
2006-06-28	Added a note about dealloc and application termination.
2006-05-23	Reorganized articles in this document to improve flow; updated "Object Ownership and Disposal."

Date	Notes
2006-03-08	Clarified discussion of object ownership and dealloc. Moved discussion of accessor methods to a separate article.
2006-01-10	Corrected typographical errors. Updated title from "Memory Management."
2004-08-31	Changed Related Topics links and updated topic introduction.
2003-06-06	Expanded description of what gets released when an autorelease pool is released to include both explicitly and implicitly autoreleased objects in "Using Autorelease Pools" (page 23).
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.