

In the last tutorial, you learned *how* objects get created and destroyed. In this one, we are going to look at *memory management* in Cocoa, which pertains to *when* objects get created and destroyed. Cocoa uses a system of memory management called *reference counting*, which is simple enough, but does take some practice. For those that find such manual schemes primitive, you'll be happy to know that [Objective-C 2.0](#), which will be ushered in by Leopard, includes garbage collection, which will eventually make much of what we learn today redundant. In the meantime, though, you will need to understand the reference counting approach, because nearly all code written with Cocoa uses it at this point in time, even if your own code does not.

## Counting References

Reference counting is conceptually simple: Whenever an object needs to make use of another object, it increases a counter in the object by one, and whenever it no longer needs it, it decreases it by one. The counter is called the *retain count* in Cocoa, and resides in the `NSObject` class, meaning virtually all objects in a Cocoa program have a retain count. If an object's retain count drops to zero, meaning no other objects need it anymore, it self-destructs, calling the `dealloc` method.

There are many ways to increase the retain count of an object. The first is simply to call the `alloc` method. Whenever an object is allocated, it automatically has a retain count of one. If you do nothing more with that object, it will remain in existence until the program exits. Other methods for creating objects, such as `copy` and `new`, also leave the object with a retain count of one.

Often, some part of your program will want to make use of an object that it did not create itself. In such cases, you want to ensure that the object in question does not disappear when it is still needed. To achieve this, the object's `retain` method can be called, which increases the retain count by one. This should ensure that the object is not deallocated until it is no longer needed.

When an object is no longer needed, you call either the `release` method, or the `autorelease` method. Both methods decrease the retain count by one, but they differ in the timing of the change. `release` will decrease the retain count immediately, and if it becomes zero, the object will be immediately deallocated.

`autorelease` decreases the retain count at some later time. For now, we won't concern ourselves with exactly when, other than to say the object will stay around long enough for any immediately executing code to complete. When you use `autorelease`, you are saying you are finished with the object, but maybe some other part of the program would like to make use of it for a bit longer. An example of this is when you need to return an object from a method, and you don't need the object anymore. If you call `release`, it will disappear before you can return it; if you call `autorelease`, you have time to return the object, and the calling code has a chance to retain it if it wants to keep it around.

## An Example

Here is a simple example, to clarify the discussion above:

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

int main() {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSMutableData *data1 = [[NSMutableData alloc] init];
    NSLog(@"Retain count of data1 is %u", [data1 retainCount] ); // Here, data1 gets a retain count
of 1
```

```

    [data1 retain];
    NSLog(@"Retain count of data1 is %u", [data1 retainCount] ); // data1 now has a retain count of
2
    NSMutableData *data2 = [data1 copy];
    NSLog(@"Retain count of data1 is %u", [data1 retainCount] ); // data1 remains on 2
    NSLog(@"Retain count of data2 is %u", [data2 retainCount] ); // data2 gets retain count of 1

    [data2 release]; // data2 retain count goes to 0
                    // dealloc is invoked

    [data1 release];
    NSLog(@"Retain count of data1 is %u", [data1 retainCount] ); // data1 retain count goes to 1

    [data1 autorelease];
    NSLog(@"Retain count of data1 is %u", [data1 retainCount] ); // data1 retain count remains on 1
                                                                // but added to autorelease pool

    [pool release]; // data1 retain count goes to 0
                  // dealloc is invoked

    return 0;
}

```

To compile this, copy it into a text file called `retaintest.m`, and enter this command:

```
gcc -ObjC retaintest.m -framework Foundation
```

Run it by entering:

```
./a.out
```

You should get output something like this:

```

2006-12-04 09:43:33.119 a.out[23394] Retain count of data1 is 1
2006-12-04 09:43:33.119 a.out[23394] Retain count of data1 is 2
2006-12-04 09:43:33.119 a.out[23394] Retain count of data1 is 2
2006-12-04 09:43:33.119 a.out[23394] Retain count of data2 is 1
2006-12-04 09:43:33.120 a.out[23394] Retain count of data1 is 1
2006-12-04 09:43:33.120 a.out[23394] Retain count of data1 is 1

```

This example basically runs through the various memory management methods, demonstrating their effects. The class used for this is `NSMutableData`, but this choice was for the most part arbitrary — the same rules apply to other Cocoa classes. Hopefully you can see the rules discussed above in action in this example. Most of the example is straightforward, but the part where `copy` is invoked may have you scratching your head. It is important to realize that `copy` does not change the retain count of the object being copied, but does create a new object — a copy of the original — and this new object gets a retain count of one.

The other aspect of this example that may perplex you is the `NSAutoreleasePool`. This is a class that takes care of releasing objects for which the `autorelease` method is invoked. In particular, just before the autorelease pool gets deallocated, it invokes `release` on any objects that have had `autorelease` called, which may result in them being deallocated, as is the case here for `data1`.

## Reference Counting Rule of Thumb

With so many methods involved in Cocoa's reference counting system, you may be thinking it is awfully complex. It isn't really, if you stick to a simple convention: Whenever you call one of the retain count increasing methods, like `alloc`, `retain`, or `copy`, you should balance that with a call to a retain count decreasing method like `release` or `autorelease`. If you remember this simple rule, you shouldn't run into too much trouble.

## Accessor Methods

Cocoa has a few conventions to help you stick to this rule of thumb. In particular, Cocoa uses *accessor methods* to control access to objects used by a class. Using these methods greatly reduces the risk of memory management bugs.

Accessor methods tend to come in pairs, with a *getter* and a *setter*. The getter is used to retrieve an object, and a setter sets the object. Here is a simple class interface to illustrate:

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

@interface Matrix : NSObject {
    NSData *data;
}

-(id)initWithData:(NSData *)data;

-(void)setData:(NSData *)data;
-(NSData *)data;

@end
```

This could form the basis of a matrix class in a linear algebra framework. The methods `setData:` and `data` are the setter and getter, respectively, of the instance variable `data`. An implementation of this class might look like this:

```
@implementation Matrix

-(id)initWithData:(NSData *)newData {
    if ( self = [super init] ) {
        [self setData:newData];
    }
    return self;
}

-(void)dealloc {
    [self setData:nil];
    [super dealloc];
}

-(void)setData:(NSData *)newData {
    if ( newData != data ) {
        [data release];
        data = [newData retain];
    }
}
```

```

}

-(NSData *)data {
    return data;
}

@end

```

There are actually many different ways to write accessor methods, and the example above represents just one such approach. The getter in this case is very simple, just returning the instance variable. The setter is more complex: it first checks that the new data is not the same as the old data. If it is, the setter does not do anything; if it is not the same, the old data is released, and the new data is retained.

Note also that the initializer calls the `setData:` accessor, rather than just setting the `data` instance variable directly. This is good practice, because it means that nearly all the retaining/releasing takes place in one part of the class: the accessors.

There is one other method that needs to include reference counting method invocations: the `dealloc` method. When a `Matrix` object gets deallocated, it needs to release its data, otherwise there will be a memory leak. `dealloc` can do this either by invoking `release` directly on the `data` instance variable, or by calling the setter with an argument of `nil`, as shown here. (In Objective-C, if you send a message to `nil`, such as will occur in this example, it is simply ignored. This can save you having to test each time if a variable is `nil`, which is a requirement in languages like C and Fortran.)

The naming scheme used for the accessors here is a Cocoa convention, and an important one. The setter gets the name `set`, followed by the instance variable name. The getter has the same name as the instance variable. This is not at all arbitrary, because Cocoa assumes you will use this convention, and many parts of the frameworks actually rely on it. If you choose a different approach, your code will simply not be able to take advantage of much of the Cocoa frameworks. Lesson: stick to the convention.

## You are Autoreleased

Memory management in Objective-C may seem a bit primitive if you are used to languages like Java and Python, but it is considerably more advanced than in languages like C and Fortran, and it does have its advantages. For example, Cocoa programs don't suffer the same performance problems that have plagued Java in the past.

As I indicated at the beginning, things are about to change, and Leopard will usher in real garbage collection. Apple seem to have done a good job of it too, and all indications are that performance will not suffer. Nonetheless, if you plan to do any Cocoa development in the coming years, you do need to understand how the reference counting scheme works, because it will be with us for some time yet. Luckily, it isn't really that difficult, as long as you stick to the rules and conventions established above.

Next time we will move out of memory management, and look at some more advanced aspects of Object-Oriented Programming (OOP), like inheritance and polymorphism, before moving into Xcode and the world of GUIs. Stay tuned.

