

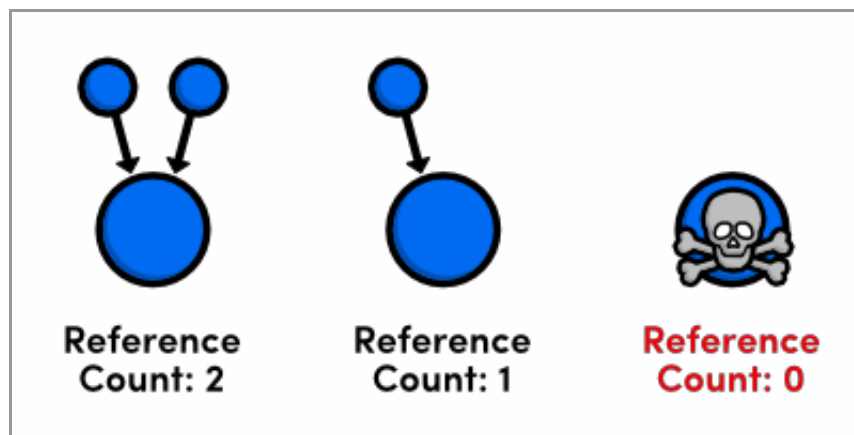
- [RyPress](#)
- [Tutorials](#)
- [Sponsors](#)
- [About](#)
- [Contact](#)

[◀ Back to Ry's Objective-C Tutorial](#)

Memory Management

As discussed in the [Properties](#) module, the goal of any memory management system is to reduce the memory footprint of a program by controlling the lifetime of all its objects. iOS and OS X applications accomplish this through **object ownership**, which makes sure objects exist as long as they have to, but no longer.

This object-ownership scheme is implemented through a **reference-counting system** that internally tracks how many owners each object has. When you claim ownership of an object, you increase its reference count, and when you're done with the object, you decrease its reference count. While its reference count is greater than zero, an object is guaranteed to exist, but as soon as the count reaches zero, the operating system is allowed to destroy it.



Destroying an object with zero references

In the past, developers manually controlled an object's reference count by calling special memory-management methods defined by the [NSObject protocol](#). This is called **Manual Retain Release (MRR)**. However, Xcode 4.2 introduced **Automatic Reference Counting (ARC)**, which automatically inserts all of these method calls for you. Modern applications should always use ARC, since it's more reliable and lets you focus on your app's features instead of its memory management.

This module explains core reference-counting concepts in the context of MRR, then discusses some of the practical considerations of ARC.

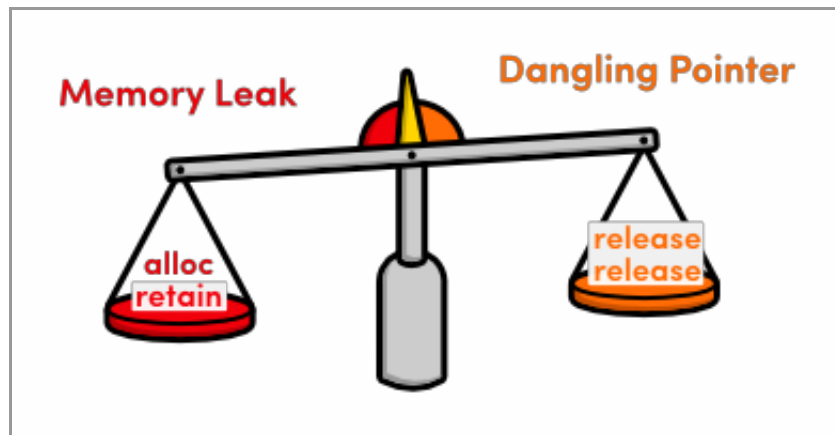
Manual Retain Release

In a Manual Retain Release environment, it's your job to claim and relinquish ownership of every object in your program. You do this by calling special memory-related methods, which are described below.

Method	Behavior
<code>alloc</code>	Create an object and claim ownership of it.
<code>retain</code>	Claim ownership of an existing object.
<code>copy</code>	Copy an object and claim ownership of it.
<code>release</code>	Relinquish ownership of an object and destroy it immediately.
<code>autorelease</code>	Relinquish ownership of an object but defer its destruction.

Manually controlling object ownership might seem like a daunting task, but it's actually very easy. All you have to do is claim ownership of any object you need and remember to relinquish ownership when you're done with it. From a practical standpoint, this means that you have to balance every `alloc`, `retain`, and `copy` call with a `release` or `autorelease` on the same object.

When you forget to balance these calls, one of two things can happen. If you forget to release an object, its underlying memory is never freed, resulting in a **memory leak**. Small leaks won't have a visible effect on your program, but if you eat up enough memory, your program will eventually crash. On the other hand, if you try to release an object too many times, you'll have what's called a **dangling pointer**. When you try to access the dangling pointer, you'll be requesting an invalid memory address, and your program will most likely crash.

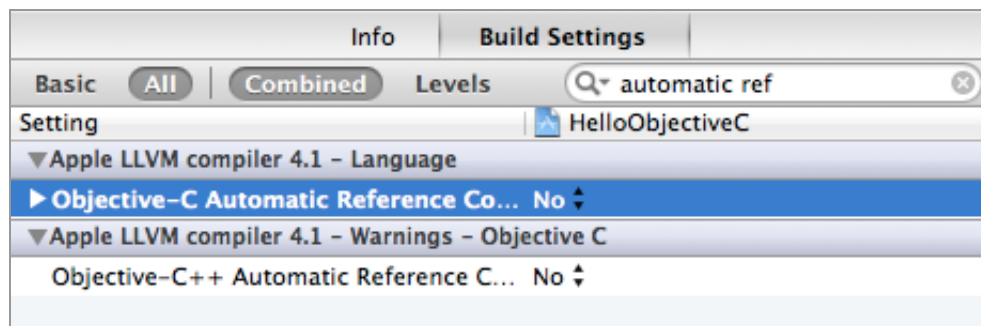


Balancing ownership claims with reliquishes

The rest of this section explains how to avoid memory leaks and dangling pointers through the proper usage of the above methods.

Enabling MRR

Before we can experiment with manual memory management, we need to turn off Automatic Reference Counting. Click the project icon in the *Project Navigator*, make sure the *Build Settings* tab is selected, and start typing automatic reference counting in the search bar. The *Objective-C Automatic Reference Counting* compiler option should appear. Change it from Yes to No.



Turning off Automatic Reference Counting

Remember, we're only doing this for instructional purposes—you should never use Manual Retain Release for new projects.

The alloc Method

We've been using the `alloc` method to create objects throughout this tutorial. But, it's not just allocating memory for the object, it's also setting its reference count to 1.

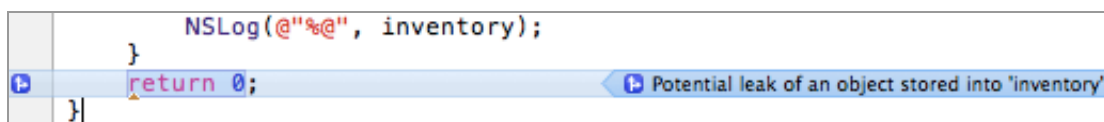
This makes a lot of sense, since we wouldn't be creating the object if we didn't want to keep it around for at least a little while.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSMutableArray *inventory = [[NSMutableArray alloc] init];
        [inventory addObject:@"Honda Civic"];
        NSLog(@"%@", inventory);
    }
    return 0;
}
```

The above code should look familiar. All we're doing is instantiating a [mutable array](#), adding a value, and displaying its contents. From a memory-management perspective, we now own the `inventory` object, which means it's our responsibility to release it somewhere down the road.

But, since we *haven't* released it, our program currently has a memory leak. We can examine this in Xcode by running our project through the static analyzer tool. Navigate to *Product > Analyze* in the menu bar or use the *Shift+Cmd+B* keyboard shortcut. This looks for predictable problems in your code, and it should uncover the following in `main.m`.



A memory leak! Oh no!

This is a small object, so the leak isn't fatal. However, if it happened over and over (e.g., in a long loop or every time the user clicked a button), the program would eventually run out of memory and crash.

The release Method

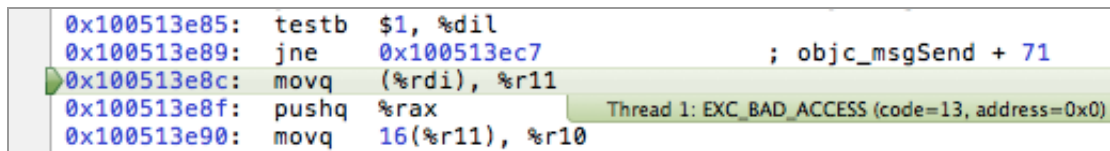
The `release` method relinquishes ownership of an object by decrementing its

reference count. So, we can get rid of our memory leak by adding the following line after the `NSLog()` call in `main.m`.

```
[inventory release];
```

Now that our `alloc` is balanced with a `release`, the static analyzer shouldn't output any problems. Typically, you'll want to relinquish ownership of an object at the end of the same method in which it was created, as we did here.

Releasing an object too soon creates a dangling pointer. For example, try moving the above line to *before* the `NSLog()` call. Since `release` immediately frees the underlying memory, the `inventory` variable in `NSLog()` now points to an invalid address, and your program will crash with a `EXC_BAD_ACCESS` error code when you try to run it:



The screenshot shows a debugger window with assembly code. The code includes instructions like `testb $1, %dil`, `jne 0x100513ec7 ; objc_msgSend + 71`, `movq (%rdi), %r11`, `pushq %rax`, and `movq 16(%r11), %r10`. A green arrow points to the `movq (%rdi), %r11` instruction. To the right, a message box displays the error: "Thread 1: EXC_BAD_ACCESS (code=13, address=0x0)".

Trying to access an invalid memory address

The point is, don't relinquish ownership of an object before you're done using it.

The retain Method

The `retain` method claims ownership of an existing object. It's like telling the operating system, "Hey! I need that object too, so don't get rid of it!" This is a necessary ability when other objects need to make sure their properties refer to a valid instance.

As an example, we'll use `retain` to create a [strong reference](#) to our `inventory` array. Create a new class called `CarStore` and change its header to the following.

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

@interface CarStore : NSObject
```

```
- (NSMutableArray *)inventory;  
- (void)setInventory:(NSMutableArray *)newInventory;  
  
@end
```

This manually declares the accessors for a property called `inventory`. Our first iteration of `CarStore.m` provides a straightforward implementation of the getter and setter, along with an instance variable to record the object:

```
// CarStore.m  
#import "CarStore.h"  
  
@implementation CarStore {  
    NSMutableArray *_inventory;  
}  
  
- (NSMutableArray *)inventory {  
    return _inventory;  
}  
  
- (void)setInventory:(NSMutableArray *)newInventory {  
    _inventory = newInventory;  
}  
  
@end
```

Back in `main.m`, let's assign our `inventory` variable to `CarStore`'s `inventory` property:

```
// main.m  
#import <Foundation/Foundation.h>  
#import "CarStore.h"  
  
int main(int argc, const char * argv[]) {  
    @autoreleasepool {  
        NSMutableArray *inventory = [[NSMutableArray alloc] init];  
        [inventory addObject:@"Honda Civic"];  
  
        CarStore *superstore = [[CarStore alloc] init];
```

```
[superstore setInventory:inventory];
[inventory release];

// Do some other stuff...

// Try to access the property later on (error!)
NSLog(@"%@", [superstore inventory]);
}
return 0;
}
```

The `inventory` property in the last line is a dangling pointer because the object was already released earlier in `main.m`. Right now, the `superstore` object has a [weak reference](#) to the array. To turn it into a strong reference, `CarStore` needs to claim ownership of the array in its `setInventory:` accessor:

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    _inventory = [newInventory retain];
}
```

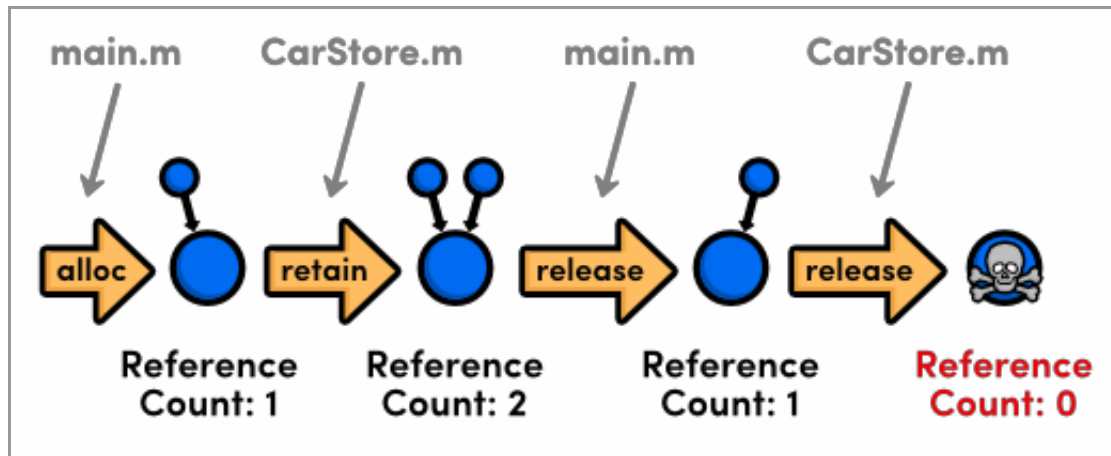
This ensures the `inventory` object won't be released while `superstore` is using it. Notice that the `retain` method returns the object itself, which lets us perform the retain and assignment in a single line.

Unfortunately, this code creates another problem: the `retain` call isn't balanced with a `release`, so we have another memory leak. As soon as we pass another value to `setInventory:`, we can't access the old value, which means we can never free it. To fix this, `setInventory:` needs to call `release` on the old value:

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    if (_inventory == newInventory) {
        return;
    }
    NSMutableArray *oldValue = _inventory;
    _inventory = [newInventory retain];
    [oldValue release];
}
```

}

This is basically what the `retain` and the `strong` property attributes do. Obviously, using `@property` is much more convenient than creating these accessors on our own.



Memory management calls on the inventory object

The above diagram visualizes all of the memory management calls on the `inventory` array that we created in `main.m`, along with their respective locations. As you can see, all of the `alloc`'s and `retain`'s are balanced with a `release` somewhere down the line, ensuring that the underlying memory will eventually be freed up.

The copy Method

An alternative to `retain` is the `copy` method, which creates a brand new instance of the object and increments the reference count on that, leaving the original unaffected. So, if you want to copy the `inventory` array instead of referring to the mutable one, you can change `setInventory:` to the following.

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    if (_inventory == newInventory) {
        return;
    }
    NSMutableArray *oldValue = _inventory;
    _inventory = [newInventory copy];
    [oldValue release];
}
```


You may also recall from [The copy Attribute](#) that this has the added perk of freezing mutable collections at the time of assignment. Some classes provide multiple copy methods (much like multiple `init` methods), and it's safe to assume that any method starting with `copy` has the same behavior.

The autorelease method

Like `release`, the `autorelease` method relinquishes ownership of an object, but instead of destroying the object immediately, it defers the actual freeing of memory until later on in the program. This allows you to release objects when you are “supposed” to, while still keeping them around for others to use.

For example, consider a simple factory method that creates and returns a `CarStore` object:

```
// CarStore.h
+ (CarStore *)carStore;
```

Technically speaking, it's the `carStore` method's responsibility to release the object because the caller has no way of knowing that he owns the returned object. So, its implementation should return an autoreleased object, like so:

```
// CarStore.m
+ (CarStore *)carStore {
    CarStore *newStore = [[CarStore alloc] init];
    return [newStore autorelease];
}
```

This relinquishes ownership of the object immediately after creating it, but keeps it in memory long enough for the caller to interact with it. Specifically, it waits until the end of the nearest `@autoreleasepool{}` block, after which it calls a normal `release` method. This is why there's always an `@autoreleasepool{}` surrounding the entire `main()` function—it makes sure all of the autoreleased objects are destroyed after the program is done executing.

All of those built-in factory methods like `NSString`'s `stringWithFormat:` and `stringWithContentsOfFile:` work the exact same way as our `carStore` method.

Before ARC, this was a convenient convention, since it let you create objects without worrying about calling `release` somewhere down the road.

If you change the `superstore` constructor from `alloc/init` to the following, you won't have to release it at the end of `main()`.

```
// main.m
CarStore *superstore = [CarStore carStore];
```

In fact, you aren't *allowed* to release the `superstore` instance now because you no longer own it—the `carStore` factory method does. It's very important to avoid explicitly releasing autoreleased objects (otherwise, you'll have a dangling pointer and a crashed program).

The dealloc Method

An object's `dealloc` method is the opposite of its `init` method. It's called right before the object is destroyed, giving you a chance to clean up any internal objects. This method is called automatically by the runtime—you should never try to call `dealloc` yourself.

In an MRR environment, the most common thing you need to do in a `dealloc` method is release objects stored in instance variables. Think about what happens to our current `CarStore` when an instance is deallocated: its `_inventory` instance variable, which has been retained by the setter, never has the chance to be released. This is another form of memory leak. To fix this, all we have to do is add a custom `dealloc` to `CarStore.m`:

```
// CarStore.m
- (void)dealloc {
    [_inventory release];
    [super dealloc];
}
```

Note that you should always call the superclass's `dealloc` to make sure that all of the instance variables in parent classes are properly released. As a general rule, you want to keep custom `dealloc` methods as simple as possible, so you shouldn't try to

use them for logic that can be handled elsewhere.

Summary

And that's manual memory management in a nutshell. The key is to balance every `alloc`, `retain`, and `copy` with a `release` or `autorelease`, otherwise you'll encounter either a dangling pointer or a memory leak at some point in your application.

Remember that this section only used Manual Retain Release to understand the internal workings of iOS and OS X memory management. In the real world, much of the above code is actually obsolete, though you might encounter it in older documentation. It's very important to understand that explicitly claiming and relinquishing ownership like this has been completely superseded by Automatic Reference Counting.

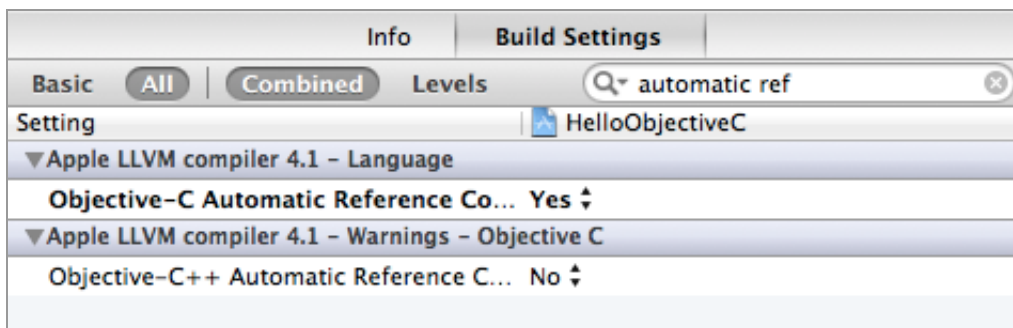
Automatic Reference Counting

Now that you've got your head wrapped around manual memory management, you can forget all about it. Automatic Reference Counting works the exact same way as MRR, but it automatically inserts the appropriate memory-management methods for you. This is a big deal for Objective-C developers, as it lets them focus entirely on *what* their application needs to do rather than *how* it does it.

ARC takes the human error out of memory management with virtually no downside, so the only reason *not* to use it is when you're interfacing with a legacy code base (however, ARC is, for the most part, backward compatible with MRR programs). The rest of this module explains the major changes between MRR and ARC.

Enabling ARC

First, let's go ahead and turn ARC back on in the project's *Build Settings* tab. Change the *Automatic Reference Counting* compiler option to *Yes*. Again, this is the default for all Xcode templates, and it's what you should be using for all of your projects.



Turning on Automatic Reference Counting

No More Memory Methods

ARC works by analyzing your code to figure out what the ideal lifetime of each object should be, then inserts the necessary `retain` and `release` calls automatically. The algorithm requires complete control over the object-ownership in your entire program, which means you're *not allowed* to manually call `retain`, `release`, or `autorelease`.

The only memory-related methods you should ever find in an ARC program are `alloc` and `copy`. You can think of these as plain old constructors and ignore the whole object-ownership thing.

New Property Attributes

ARC introduces new `@property` attributes. You should use the `strong` attribute in place of `retain`, and `weak` in place of `assign`. All of these attributes are discussed in the [Properties](#) module.

The dealloc Method in ARC

Deallocation in ARC is also a little bit different. You don't need to release instance variables as we did in [The dealloc Method](#)—ARC does this for you. In addition, the superclass's `dealloc` is automatically called, so you don't have to do that either.

For the most part, this means you'll never need to include a custom `dealloc` method. One of the few exceptions is when you're using low-level memory allocation functions like `malloc()`. In this case, you'd still have to call `free()` in `dealloc` to avoid a memory leak.

Summary

For the most part, Automatic Reference Counting lets you completely forget about memory management. The idea is to focus on high-level functionality instead of the underlying memory management. The only thing you need to worry about are retain cycles, which was covered in the [Properties](#) module.

If you need a more detailed discussion about the nuances of ARC, please visit the [Transitioning to ARC Release Notes](#).

By now, you should know almost everything you need to know about Objective-C. The only thing we haven't covered are the basic data types provided by both C and the Foundation Framework. The next module introduces all of the standard types, from numbers to strings, arrays, dictionaries, and even dates.

[Continue to *Data Types* ›](#)

- © 2012–2013 RyPress.com
- All Rights Reserved
- Terms of Service