Mobile Development

Dr. Dobb's Journal

January 2013

Debugging Memory in iOS

Next

ALSO INSIDE

UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

January 2013

FEATURES

EDITORIAL

More on DrDobbs.com

Writing Your First Windows 8 App: The Lay of the Land

A lot of the basic features of Windows apps have changed with the release of Win 8. Here is a step-by-step explanation of what's new and how to get started writing Win8-compliant apps.

http://www.drdobbs.com/240143752

Easy, Real-Time Big Data Analysis Using Storm

Conceptually straightforward and easy to work with, Storm makes handling big data analysis a breeze.

http://www.drdobbs.com/240143874

The Scourge of Error Handling

Return values and exceptions are both inadequate tools for handling errors. But we're stuck with them for the foreseeable future — just as we have been for decades.

http://www.drdobbs.com/240143878

Conway's Game of Life In Parallel

The iconic game is perfectly suited to parallel execution. This series of posts looks at several approaches — from OpenMP to message passing — that improve the game's performance.

http://www.drdobbs.com/240143844

Scaling Up And Out

Most attention today is focused on adding nodes or cloud instances to scale out systems, but scaling systems vertically is important as well.

http://www.drdobbs.com/240142249

Mailbag

Error handling and methodologies

The Scourge of Error Handling

With respect to editor Andrew Binstock's editorial on the problems of handling errors (www.drdobbs.com/240143878), we received these interesting observations.

"Andrew Binstock's recent article 'The Scourge of Error Handling,' briefly mentioned Haskell's `Maybe` monad. I thought I'd write about some successes in error handling I've seen from using this monad (called `Option` in Scala) and the related `Either` monad.

At the risk of infuriating the monad police, I think of these types as returning data structures: The `Option` monad can be thought of as a list that contains either zero or one element, the `Either` monad can be thought of as the left and right branches of a tree. If an `Option` is an empty list, that's `None`, if it has an element, that's `Some`. That's broadly applicable: if a function turns a value of type `T` into a value of type `U`, it's probably logical that when applied to an empty `List<T>` it returns an empty `List<U>`.

The challenge with error handling is usually not so much with a single function call but when functions are chained: when `foo` relies on the output of `bar`, which in turn relies on the output of `bat`. With value-oriented returns that return error codes, the developer generally needs to make their code branch on a problem lest it fall prey to 'Garbage In, Garbage Out.' But if the code is written to expect a `List<T>` and return a `List<U>`, getting passed an empty list of type `T` is no big deal, since the code can just return an empty list `U` without sweating the logic. 'Garbage Can In, Garbage Can Out,' is nice and hygienic: If `bat` returns a `None`, `foo(bar(bat))` also returns a `None` — a more robust result at runtime than a `NullPointerException`. However, this robustness may come at the price of obscurity; getting a `None` back from a complex calculation prevents your application from crashing, but a `NullPointerException` with a stack trace may well be easier to debug (it would be straightforward to add a stack trace to `None` implementations, if implementers could tolerate the taint of pragmatism).

Even more flexible than `Option` is `Either`, which returns subtypes `Left` or `Right`. By convention, `Right` is returned when a function succeeds, `Left` when an error or exception occurs `Just` as with `Option`, this can be thought of in terms of data-structures or containers: the `Left` is a Garbage Can and the `Right` is a box from Amazon containing good stuff. Again, it's easy for a function passed a `Left` to recognize that it's being called with bad data and avoid furthering the problem.

One benefit of the `Either` monad is that it's easy to accumulate the results of a large number of function calls and then divvy them up between successes and failures. For instance, in a project that evaluated research proposals, we applied a large number of validations to every element in the proposal, and eventually split up problem observations from valid ones with code as simple as `(badObservations, goodObservations) = observations.map(errorChecks).split`. We could then present the `badObservations` to the scientists with an explanation of what problem, exactly, we were diagnosing. This was vastly more helpful than just showing them the first problem and vastly nicer in the codebase, since our diagnosis code didn't need to concern itself with the problem of accumulating results.

Although these techniques work very well at the application level, as a general mechanism, they have the significant runtime disadvantage of not short-circuiting evaluation. If `bat` has a problem in `foo(bar(bat()))`, `foo` and `bar` are still called (to no effect). That model is presumably a non-starter for system-level and embedded programming in very resource-constrained environments.

Another disadvantage is that these techniques invoke functional programming's dreaded "M" word (and, naturally, I couldn't resist introducing yet-another metaphor to explain monads). I know many programmers are turned off by the jargon of functional programming, which is unfortunate, as the techniques described can be implemented in every popular programming language from JavaScript and Python to C++."

—Larry O'Brien

[Larry was the editor in chief of *Computer Language* and *Software Development*, both sister publications to *Dr. Dobb's*. —Ed.]

"The solution is to decently-factor your code. Every function should do exactly one thing. It's an artform, and there is no right answer, but many wrong ones. If you have an unmanageable linear sequence of these

```
if (... == fail) { ... }
```

snippets, then you must find a way to group them as well. Factoring remains the only tool available to produce clear code. Without proper factoring, no code can be readable no matter what syntactic or stylistic baggage you tack on. With proper factoring, no code is unreadable. All syntactic sugar since the dawn of HLASM has been dedicated to making it easier to factor in one way or another.

It is much more nebulous than simply stating 'I'm going to re-invent `try-catch`, except this time the compiler is going to do a good job of it.' On the other hand, it is the only thing that has ever worked."

— Greg Alexander

Andrew Binstock responds: "Refactoring, or factoring if you prefer, is a crucial step in keeping code clean and orderly. It also reduces the clutter of error checking, but it doesn't get rid of the problem of error checking or handling exceptions."

"I agree that error handling has been neglected. Maybe one day a famous person will break down the use cases and mechanically derive a better way of handling errors. I suspect the lack of progress is due to the extremely unimaginative times we live in."

— Kyle Lahnakoski

The Fruitlessness of No Methodology

These letters responded to our editorial anent the difficulties of a no-methodology approach (www.drdobbs.com/240144237):

"Dear Editor,

Watts Humphrey's personal software process (PSP) solves the problem of cowboy programming that your column discussed. However, the existence of a solution does not mean that everybody will use it. The problem with poor software is not the lack of technical solutions, but the fact that there are only weak methods for transmitting these solutions to the population that needs to know them. There is also resistance from the group who need the solution."

— Capers Jones

"I'm afraid your prejudices are showing. Not all 'mavericks' spend all their time inside the debugger. Actually, I think reliance on the debugger is the sign of bad programmers of all stripes, and you get bad programmers working in — and even championing — every kind of methodology. I do know that, as a hiring manager, frequently I seek

out soloists because I've found that they are often capable of achieving in short order things that a team of ten can't achieve no matter how much time they're given. Often, solo players produce outstanding results (high-reliability, comprehensible code) precisely because they don't have to spend hours trying to explain their ideas to the entire team — including the stupidest member.

Your prejudices show even more when you write 'The presence of a methodology will invariably generate a clearer understanding of the deliverables, a well-defined set of responsibilities that can be assigned to team members, defined build and test processes, a review process, and a shipment calendar.'

It will? Invariably? Not in my experience! I've seen all kinds of processes fail abysmally (agile methodologies slightly more often than most), for a whole raft of reasons. I do know that teams who claim that 'agile is the best way,' without any understanding of why or where it's appropriate, are the people most likely to destroy the team structure they've worked so hard to construct.

— Jules May

Andrew Binstock responds: "You're right. 'invariably' is too strong a word due to the case you bring up: poor methodology implementations. However, any methodology done more or less right will deliver those results when compared with the use no methodology at all."

Have a correction or a thoughtful opinion on Dr. Dobb's content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

You Are No Longer A Testing Expert

The testing and optimizing techniques of the past don't apply to mobile.

By Dave Berg

The rise of mobile technology and applications has marked the beginning of the end for many so-called application testing "experts." Mobile has the potential to become an extinction event that threatens to render an entire generation of testing professionals obsolete, unless they choose to evolve. Unfortunately, many are still in denial about the need to adapt. This denial is fueling the sorry state of mobile application testing we see today. Perhaps this note can serve as a wake up call for change and lead to a flourish of experts engineering well-performing mobile apps.

The Same But Different

Mobile continues to redefine the ways in which we develop, test, and deploy applications. The skills gained through years of traditional (defined here as hard-lined, PC-based) app development and testing no longer apply. While they might seem the same, their complexity is as different as a 2013 Toyota Prius is from a 1960s VW Bug. They both have four wheels and an engine, but unless you have the mechanical skill set and knowledge to work on an on-board computer or nickel-metal hydride batteries, there is no way I'm letting you touch my hybrid.

All About the Network

Which do you usually complain about more when it comes to performance: your traditional Internet connection or your mobile data connection? I'll put my money on your mobile connection. The reason is the network. From city to city, and even sometimes block to block, connection and performance is unreliable and constantly shifting. The simple physics of mobile technology result in greater latency variation than a hard-lined connection, making slower, unreliable speeds inherent in a mobile connection.

This was not something you had to account for when testing apps across traditional network conditions. A hard-lined connection, even if it was at a crawling 28.8Mbps, could be counted on to remain constant. You knew what network conditions your app would encounter and this made performance testing (relatively) easy.

Assuming a constant level of network performance is a risky misstep with mobile that can leave major issues undetected. Testing apps in a traditional manner without accounting for such mobile network issues as increased and variable latency, jitter, and packet loss means your testing will yield unreliable results and put your app in a position for failure.

There Is Nothing More You Can Do

In traditional app development and testing, there was almost always something "else" you could do to improve performance. If you were perfecting an enterprise application, you more than likely had control

over the hardware that would run the application. You could install more memory or faster processors on that hardware to make up for deficiencies in the app.

With the rise of BYOD (bring your own device) and the fragmentation of the mobile device space, there is very little you can do to the hardware to mask application performance issues. An iPhone is an iPhone until the next one is released. With mobile apps, you must tweak the app to fit the hardware.

While it used to take years between versions of Microsoft Windows, new Android and iOS versions are being released annually (sometimes even more often). Each new OS version presents a new set of issues to design, develop, and test for to ensure apps perform up to par on the latest and greatest systems.

With mobile apps, too, you don't have the ability to manipulate the network as you could when working with a traditional system. Your toolbox of fixes is now obsolete. You cannot add extra bandwidth to the Verizon or AT&T mobile networks. Similarly, WAN accelerators or other network and application optimization solutions are already built into the mobile networks, so that option for increasing performance is off the table.

If issues are discovered in mobile app testing, you need a new set of tools. Reducing the chattiness of the application, shrinking or eliminating files or images, and JavaScript placement are now some of the "go-to" remedies for the mobile app tester. The focus has to be on improving the app through better design, development, and testing. Ways of manipulating the environment in which hard-lined applications operated just don't cut it anymore.

Time Is Not On Your Side

It takes time to become an expert at anything. How long have you been developing and testing Internet and PC-based apps? Contrast that to how long you have been dealing with mobile applications. With that difference in time and experience, it is hard to consider anyone an "expert" in mobile app testing.

But here's the Catch-22. While serious emphasis on mobile apps might still be just a few years old, you do not have the luxury of spending years building up a mobile skill set. You need those skills now. The speed of mobile adoption and its dramatic effect on business, in terms of productivity, revenue, and competitive advantage, is unlike anything we have seen. The public, and therefore your employer, is demanding well-performing mobile apps now.

We are reaching a momentous tipping point where the majority of Internet activities will be executed from a mobile device. While I'm not about to sound the death knell for traditional Internet connections, the future, and its jobs, will belong to mobile.

Embrace the change. Embrace the challenge of mastering an emerging, yet deeply rooted, technology. By embracing and mastering the differences of mobile application testing, you are positioning yourself to be in demand for many years to come.

— Dave Berg is the senior director of product management at Shunra Software, a Philadelphia-based company specializing in network virtualization.

Comment

Debugging Memory In iOS Apps

Dangling pointers, double frees, memory leaks and hogs — when it comes to writing apps for iOS, you've got to avoid these common pitfalls. Here's how!

By José R. C. Cruz

Software products slated for the iOS market have to be frugal in their use of memory. iOS devices such as the iPhone and iPad have limited physical memory, much less than their flash storage. Using Xcode can help design frugal code and subject source files to static analysis. You can also use its Instruments tool to track down memory problems at runtime. In this article, I discuss some common memory problems and how they can affect a typical iOS app. I show you how to detect these problems with the aforementioned tools and some ways of fixing them.

You will need a working knowledge of ANSI-C, Objective-C, and Xcode. The sample project needs version 3.x (or newer) of the Xcode development suite.

Types of Memory Problems

Most memory problems are one of four types. The first type is the dangling pointer. This is an object or data pointer that still refers to a deallocated memory block. The block may still have valid data, but the data can "disappear" at any point in time. Attempts to access a dangling pointer may lead to a segmentation fault (`EXC_BAD_ACCESS` or `SIGSEGV`). A dangling pointer should not be confused with a `NULL`, which is defined as `((void *) 0)`.

Consider the snippet in Listing One. Here, class `FooProblem` has a single property (`objcString`) and a single action method. The action method, `demoDanglingPointer:`, initializes `objcString` to an empty `NSString` (line 18). Then, in a separate code block, it creates an instance of `NSMutableString` (line 20). It sends the instance a release message (line 22), but also assigns that same instance to `objcString` (line 25). Once the code block exits, deallocation occurs and `objcString` is left holding a dangling pointer.

Listing One

```
01 // -- FooProblem.h
02 @interface FooProblem : NSObject
03 {
04     // -- properties:demo
05     NSString *objcString;
06 }
07
08 // -- methods:demo:actions
09 - (IBAction) demoDanglingPointer:(id)aSrc;
10 @end
11
```

```
12 // -- FooProblem.m
13 @implementation FooProblem
14 - (IBAction) demoDanglingPointer:(id)aSrc
15 {
16     NSString *tBar;
17
18     objcString = [NSString string];
19       {
20                 tBar   =   [[NSMutableString   alloc]
initWithString:@"foobar"];
21       //... do something else
22       [tBar release];
23       }
24    //..do something else
25    objcString = tBar;
26 }
27 @end
```

Now consider now the snippet in Listing Two. This variant of `FooProblem` declares a C `struct` named `FooLink` (lines 1-6). In its `demoDanglingPointer:` action, it declares the local variable `tTest` and assigns the latter the output from method `danglingPosix()` (line 26). The `danglingPosix()` method creates a local instance of `FooLink (tBar)` inside a code block (lines 38-41). It assigns values to the struct fields and sets the local `tFoo` to `tBar` (line 42). But just before the code block exits, `danglingPosix()` disposes of `tBar` with a call to `free()` (line 44) and returns the pointer held by `tFoo` (line 48). The result is that `local tTest` now has a dangling pointer.

Listing Two

```
01 typedef struct Foo
02 {
03    char *fFoo;
04    unsigned int fBar;
05    struct Foo *fNext;
06 } FooLink;
07
```

```
08 // -- FooProblem.h
09 @interface FooProblem : NSObject
10 {
11    // -- properties
12    //...
13 }
14
15 // -- methods:demo:actions
16 - (IBAction) demoDanglingPointer:(id)aSrc;
17 - (FooLink *)danglingPOSIX;
18 @end
19
20 // -- FooProblem.m
21 @implementation FooProblem
22 - (IBAction) demoDanglingPointer:(id)aSrc
23 {
24    FooLink *tTest;
25
26    tTest = [self danglingPOSIX];
27    // ...do something else
28 }
29
30 - (FooLink *)danglingPOSIX
31 {
32    FooLink *tFoo;
33
34    // initialise the output result
35    tFoo = nil;
36
37    {
38          FooLink *tBar;
39          tBar = malloc(sizeof(FooLink));
40          tBar->fFoo = "foobar";
41          tBar->fBar = 1234;
42          tFoo = tBar;
43          //... do something else
44          free(tBar);
45    }
46
47    // return the string result
48    return (tFoo);
49 }
50 @end
```

The next type of memory problem is the double free. This occurs when a code routine tries to dispose of an object or structure that's already been disposed of. Disposal need not happen in succession, so long as it affects the same pointer. A double free also leads to a segmentation fault, followed by a crash.

Listing Three is a classic example of a double free. The action method `demoDoubleFree:` creates an instance of `FooLink` and populates its fields (lines 3-5). It sends the instance to the method `doubleFreePOSIX:`, which updates the two fields (lines 14-15). But then `doubleFreePOSIX:` disposes of the `FooLink` instance with a call to `free()` (line 18). When it returns control to `demoDoubleFree:`, `demoDoubleFree:` also disposes of the same `FooLink` structure using `free()` (line 8).

Listing Three

```
01 - (IBAction) demoDoubleFree:(id)aSrc
02 {
03    tFoo = malloc(sizeof(FooLink));
04    tFoo->fFoo = "Foobar";
05    tFoo->fBar = 12345;
06
07    [self doubleFreePOSIX:tFoo];
08    free(tFoo);
09 }
10
11 - (void)doubleFreePOSIX:(FooLink *)aFoo
12 {
13    //...do something else
14    aFoo->fFoo = "BarFoo";
15    aFoo->fBar = aFoo->fBar + 123;
16
17    //... do something else
18    free(aFoo);
19 }
```

Listing Four shows another double free example. The action method `demoDoubleFree:` creates and adds an `NSNumber` instance to the mutable array property `objcArray` (lines 23-24). Then it invokes the method `doubleFreeObjC`. This method parses the array property and uses its entries to create an `NSString` object (line 39-40). Later, it sends a release message to each entry (line 44). If the entry is the `NSNumber` object, a double free error occurs. This is because the `NSNumber` object was marked for autorelease. An explicit release interferes with the autorelease pool's attempt to dispose of the object.

Listing Four

```
01 // -- FooProblem.h
02 @interface FooProblem : NSObject
03 {
04    // -- properties
05    NSMutableArray *objcArray;
06 }
07
08 // -- methods:demo:actions
09 - (IBAction) demoDoubleFree:(id)aSrc;
10
11 // -- methods:demo:utilities
12 - (void)doubleFreeObjC;
13 @end
14
15 // -- FooProblem.m
16 @implementation FooProblem
17 // ...truncated for length
18
19 - (IBAction) demoDoubleFree:(id)aSrc
20 {
21    NSNumber *tNum;
22
23    tNum = [NSNumber numberWithLong:rand()];
24    [objcArray addObject:tNum];
25
26    //...do something else
27
28    [self doubleFreeObjC];
29 }
30
```

```
31 - (void)doubleFreeObjC
32 {
33    NSString *tText;
34    id tObj;
35
36    tText = [NSString string];
37    for (tObj in objcArray)
38    {
39        tText = [tText
40          stringByAppendingFormat:@"/@", tObj];
41
42          //...do something else
43
44          [tObj release];
45    }
46 }
47 @end
```

A memory leak is another typical memory problem. It occurs where a routine fails to dispose of its objects or structures. The failure may be due to an error or exception, or it may be due to poor code design. A continuous memory leak can lead to a low-memory condition. At best, it could cause iOS to terminate the offending app. At worst, it could force users to perform a hard reset.

Listing Five is one example of a memory leak. This variant of `FooProblem` creates an empty `NSMutableArray` instance and assigns it to the property `objcArray` (lines 17-18). Its `dealloc` method, however, does not send a release message to `objcArray`. The result is, of course, a memory leak, even if the `array` object remains empty.

Listing Five

```
01 // -- FooProblem.h
02 @interface FooProblem : NSObject
03 {
04    // -- properties
05    NSMutableArray *objcArray;
06 }
07 @end
08
09
10 // -- FooProblem.m
11 @implementation FooProblem
12 - (id)init
13 {
14    if (self = [super init])
15    {
16          // prepare the following properties
17          objcArray = [[NSMutableArray alloc]
18        initWithCapacity:1];
19
20          return (self);
21    }
22    else
23          // the parent has failed to initialise
24          return (nil);
25 }
26
```

```
27 //...other methods go here
28
29 - (void)dealloc
30 {
31    //...do something else
32
33    // pass the message to the parent
34      [super dealloc];
35 }
36 @end
```

Listing Six presents another example of a memory leak. Here, the action method `demoMemoryLeak:` creates an instance of `FooLink` and stores it into the property `posxLeak` (lines 16-18). At first glance, the method looks correct. But if you kept invoking `demoMemoryLeak:` without disposing of the previous `FooLink` instance, you will end up with a leak.

Listing Six

```
01 // -- FooProblem.h
02 @interface FooProblem : NSObject
03 {
04    // -- properties
05    FooLink *posxLeak;
06 }
07 - (IBAction) demoMemoryLeak:(id)aSrc;
08 @end
09
10
11 // -- FooProblem.m
12 @implementation FooProblem
13 - (IBAction) demoMemoryLeak:(id)aSrc
14 {
15    // initialise the following private property
16    posxLeak = malloc(sizeof(FooLink));
17    posxLeak->fFoo = "foobar";
18    posxLeak->fBar = rand();
19 }
20 @end
```

The final common problem is the memory hog. This is where an object or structure uses up at least a quarter of the available memory. It holds on to its allocated memory until it is properly disposed of. An example of a memory hog is shown in Listing Seven. The action method `demoMemoryHog:` uses the instance method `initWithContentsOfFile:` to create an `NSMutableArray` object with data from the file `foobar.data` (lines 6-11). If the source file is small, perhaps less than a half a megabyte, the array object will also be manageably small. But what if the file is more than a megabyte in size? What if the read data results in a multi-dimensional array? Then you get an array object that can take up half or more of the available memory.

Listing Seven

```
01 - (IBAction) demoMemoryHog:(id)aSrc
02 {
03    NSString *tPth;
04
05    // initialise the following property
06    tPth = [NSString
07          stringWithString:@"~/Documents/foobar.data"];
08    tPth = [tPth stringByExpandingTildeInPath];
09
10    objcArray = [[NSMutableArray alloc]
11          initWithContentsOfFile:tPth];
12 }
```

Listing Eight is another example of a memory hog. This variant of `demoMemoryHog:` starts by creating a `FooLink` instance and populates its two fields (lines 7-9). Then, it proceeds to create a linked-list of 1024 nodes (lines 12-24). Once done, the resulting data structure stays resident, taking up precious memory unless all of the nodes are disposed properly.

Listing Eight

```
01 - (IBAction) demoMemoryHog:(id)aSrc
02 {
03    FooLink *tNode, *tTest, *tLink;
04    NSInteger tIdx, tMax;
05
06    // create the first link note
07    tNode = malloc(sizeof(FooLink));
08    tNode->fFoo = "foobar";
09    tNode->fBar = 1;
10
11    // create the rest of the link
12    tTest = tNode;
13    tMax = 1024;
14    for (tIdx = 0; tIdx < tMax; tIdx++)
15    {
16          // prepare the link node
17          tLink = malloc(sizeof(FooLink));
18          tLink->fFoo = "barfoo";
19          tLink->fBar = tIdx + 2;
20
21          // update the linked list
22          tNode->fNext = tLink;
23          tNode = tLink;
24    }
25
26    //...do other things
27 }
```

There is one more memory problem that bears mentioning — heap corruption. Heap corruption happens when code in the app's heap region is altered unexpectedly. The effects are usually subtle, ranging from incorrect behavior to random crashes. There are many causes of heap corruption. It may be sourced to stack collisions or buffer overruns. Or it may be caused by any of the memory problems discussed earlier. As such, a definitive example of heap corruption is hard to present, and is omitted from the rest of this discussion.

Debugging by Static Analysis

Static analysis can identify potential memory problems in a project. You can then fix these problems while avoiding the overhead of runtime debugging. With Xcode, static analysis is done with the open-source tool clang, which can handle both ANSI-C and ObjC source files. It works with the new LLVM compiler, available on version 3.2 or later of the Xcode IDE.

To start the analysis, choose Build and Analyze from Xcode's Project menu. Clang then highlights the suspect statement, and describes the problem. Figure 1 shows a sample result of one analysis. Function `danglingObjC` creates an `NSString` instance (`tOrig`) with the method `initWithString:`. It copies the object and assigns the copy to the local `tCopy`. Then it sends `tOrig` a release message and returns `tCopy` as its result.

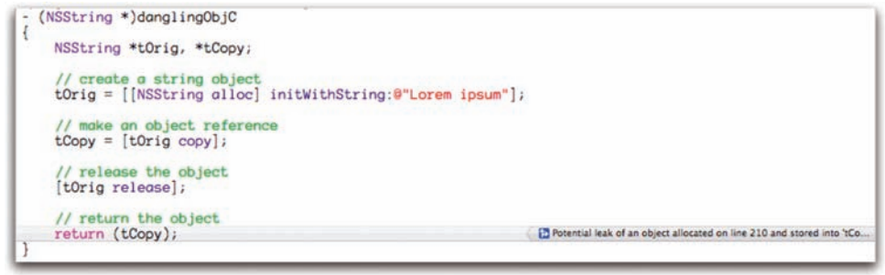


Figure 1.

Here, Clang highlighted the last statement in the function. It warned that the `tCopy` result is a potential memory leak. This is because the copy message creates a separate instance of `NSString`. The release message affects only the `tOrig` object, not `tCopy`. One way to correct this error is to mark `tCopy` for autorelease as it is returned:

```
return ([tCopy autorelease]);
```

Another way is to have the caller method dispose of the returned string object:

```
tText = [self danglingObjC];
//...do something with the string object
[tText release];
```

Figure 2 shows another sample result. The action method `demoDanglingPointer:` starts by assigning an empty `NSString` instance to property `objcString`. Then, in a separate block, it creates another `NSString` instance (`tBar`), one with a value of `"foobar"`. It sends a release message to `tBar`, but also uses `tBar` to create and assign a new `NSString` object to `objcString`.

```
- (IBAction) demoDanglingPointer:(id)aSrc
{
    // -- ObjC:dangling-pointer
    NSString *tBar;

    objcString = [NSString string];
    {
        tBar = [[NSMutableString alloc] initWithString:@"foobar"];
        [tBar release];
    }
    objcString = [NSString stringWithString:tBar];   Reference-counted object is used after it is released
}
```

Figure 2.

Here, too, Clang highlighted the last statement in `demoDanglingPointer:`. It warned that the `NSString` object assigned to `objcString` is potentially invalid. The memory region with its value `"foobar"` may be disposed of at any time, essentially making it a dangling pointer. One way to correct this problem is to move the code for creating `tBar` outside the code block. Another way is to create `tBar` with the factory method `stringWithString:`, and do away with the explicit release message:

```
tBar = [NSMutableString stringWithString:@"foobar"];
```

This marks the `string` object for autorelease. As long as `objcString` stays valid, its reference to the object remains strong, preventing the latter's accidental disposal.

The Instruments Tool

Not all memory problems are easily caught through static analysis. Some appear only at runtime, and sometimes only after running a method or function several times. To detect these problems, you need to use the Instruments tool.

This tool uses the open-source framework DTrace to attach itself to an iOS app process and to acquire trace data. It graphs the trace data and lists the objects that make up the data. And it shows the stack frame that lead to each object.

Figure 3 shows a typical trace session. The session window divides itself into four panes; each pane can be hidden or shown. The Instruments pane consists of modules that gather and process the trace data. The Track pane displays the trace data from each module. The Details pane lists the objects and structures detected by the session. It can filter the list to a specific few, and it can drill-down on a specific object. The Extended Details pane reveals the stack frame for a given object. It also identifies the source file or library that made the object.

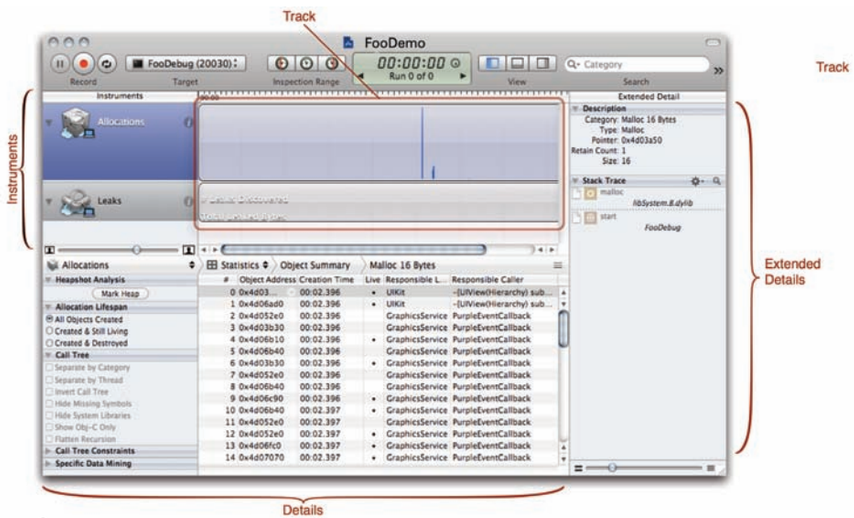


Figure 3.

Multiple sessions are supported. As a rule, though, only one session should be active to avoid confusion and to reduce impact on app performance. Trace data can be studied during or after acquisition, or it can be saved to the disk for later analysis.

A session can have one or more modules, all targeting the same process. The Instruments tool comes with at least a dozen modules for monitoring an iOS app process. It also has facilities for creating a custom module. But there are just two modules that are suited to memory debugging.

First is the Allocations module (Figure 4). This module records every type of object or structure created and disposed of by the app process. It keeps track of each object's retention and reference count. It can even recast deallocated objects as zombies (for flagging any attempts to use a deallocated object).

The second is the Leaks module (Figure 5). This module looks for objects or structures that are potential leaks. It displays each leak as a bright-

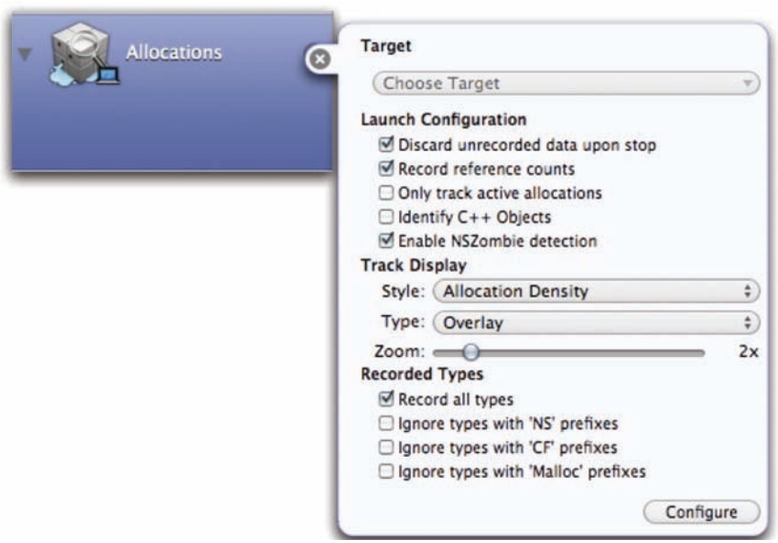


Figure 4.

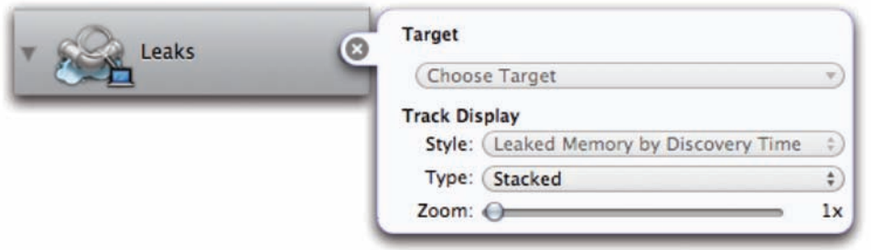


Figure 5.

colored spike overlaid against a running total of leaks. It can be used alone, but it is often used in conjunction with the Allocations module.

Debugging with Instruments

To prepare the iOS app for a debug session, first make sure that the app builds without error and installs into the IOS simulator. Make sure it has displayed its main window and it can respond to user clicks. If the app crashed or froze at launch time, the memory problem is likely in its `init` code.

Next, comment out or remove all calls to `NSLog()`. `NSLog()` delivers its log messages as `NSString` objects. These objects can cause confusion, especially when the problem is due to an `NSString`. Isolating `NSLog()` calls helps reduce the number of string objects to be monitored.

Finally, use only the version of Instruments that came bundled with the active Xcode IDE. For Xcode 3.2.6, it is Instruments 2.7. For Xcode 4.2, it is Instruments 4.2. A different version of Instruments may not work properly with a given IDE session. At best, Instruments will crash when it tries to acquire process data.

Figure 6 shows the session setup used in this article. On it are the Allocations and Leaks module described earlier. The Allocations module is prepped to track all objects and structures, record reference counts, and employ zombies. The Leaks module is prepped to look for leaks at one-

second intervals. The setup also includes the Sampler module. This module keeps track of processes on the iOS simulator. With it, you can verify if a memory problem is caused by the target process and not some other errant process.

To connect to the target app, click on the Target pop-up button and choose the process name from the submenu Attach to Process. In our sample setup, the target app is `FooDebug` with a process ID of `24912`. To start a trace, choose Mini Instruments from the Instruments' View menu. This collapses the session window into a basic translucent record window (Figure 7). Click the record button on that window to start acquiring process data, and run the desired feature or task on the app. Click the record button again to end acquisition, and the close button to view the results. If your display is large enough, you could dispense with the Mini Instruments mode and have both session window and iOS simulator side by side.

Figure 8 shows how an ObjC memory leak appears in the session window.

The routine in question is the action method `demoMemoryLeak`: (Listing Nine) executed twice.

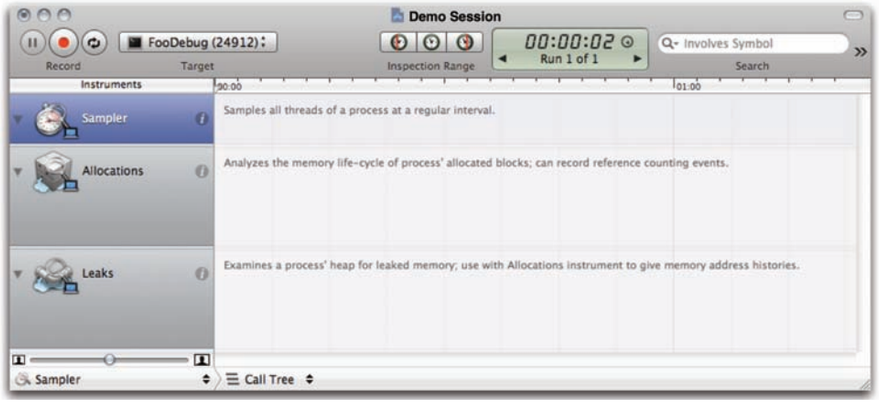


Figure 6.

Figure 7.

Listing Nine

```
01 - (IBAction) demoMemoryLeak:(id)aSrc
02 {
03    NSMutableString *tLeak;
04    NSNumber *tNum;
05
06    // prepare a new string object
07    tNum = [NSNumber numberWithLong:rand()];
08    tLeak = [[NSString alloc]
09          initWithFormat:@"%d", tNum];
10
```

```
11    // update the following private property
12    objcString = tLeak;
13 }
```

Each memory leak appears as a red spike in the Track pane next to the Leaks module. Selecting a spike using an Option-drag reveals the leak as an `NSCFString` object. Clicking the right-arrow icon next to the object's pointer address (`0x4e2a8b0`) starts a drill-down. It shows that the leak was caused by the instance method `initWithString:`. One way to correct this leak is to use the factory method `stringWithString:` to create the `string` object. Another way is to replace line 12 in Listing Nine with these statements:

```
[objcString release];
objcString = tLeak;
```

Figure 9 shows another memory leak. This one was caused by the `demoMemoryLeak:` method described back in Listing Six. Here, too, each red spike marks each leak that appeared. Selecting a spike reveals the leak: a `malloc`'d 16-byte data structure with a pointer address of `0x4b56020`. It also reveals the name of the action method. A drill-down on the structure supplies more details about the leak. To fix this leak, try inserting this statement before line 16 of Listing Six:

```
free(posxLeak);
```

Figure 10 shows the session tracking a double free problem. The target routine in this one is the `demoDoubleFree:` action method described in Listing Four. Three spikes show the allocations made just before the app crashes with an `EXC_BAD_ACCESS`. Selecting these spikes bring up the single `CFNumber` object (`0x4d11970`) that `demoDoubleFree:` created and added to `objcArray`. A drill-down then reveals the object's retention history.

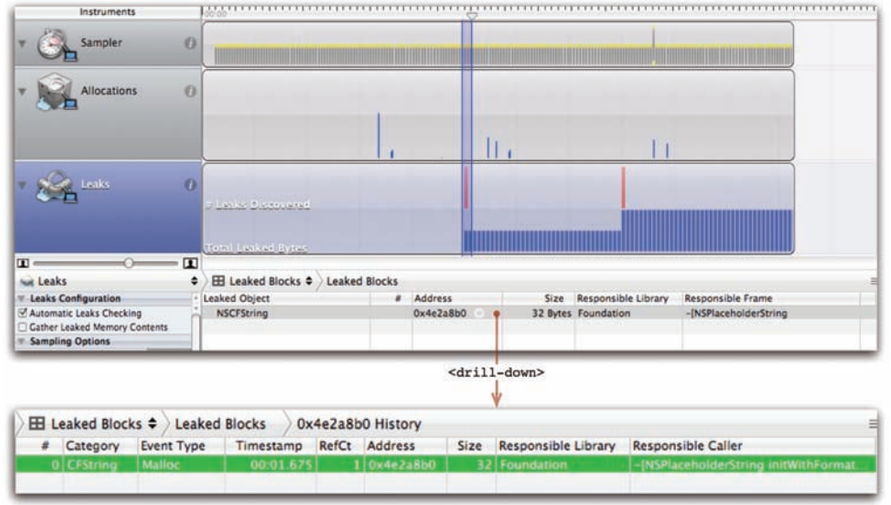


Figure 8.

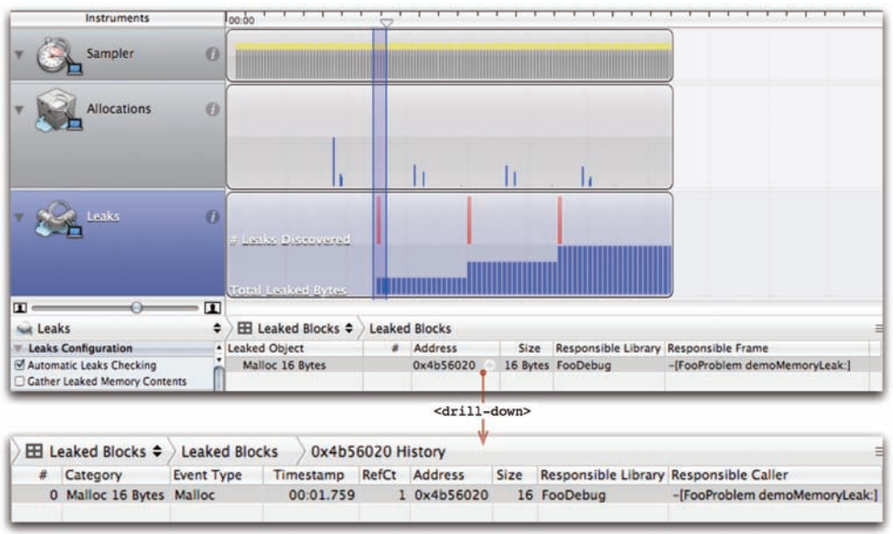


Figure 9.

IN THIS ISSUE

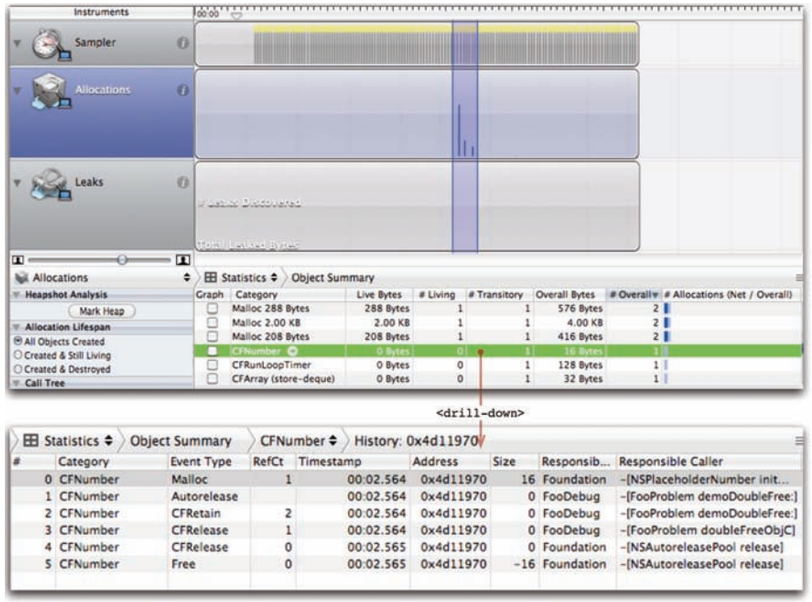


Figure 10.

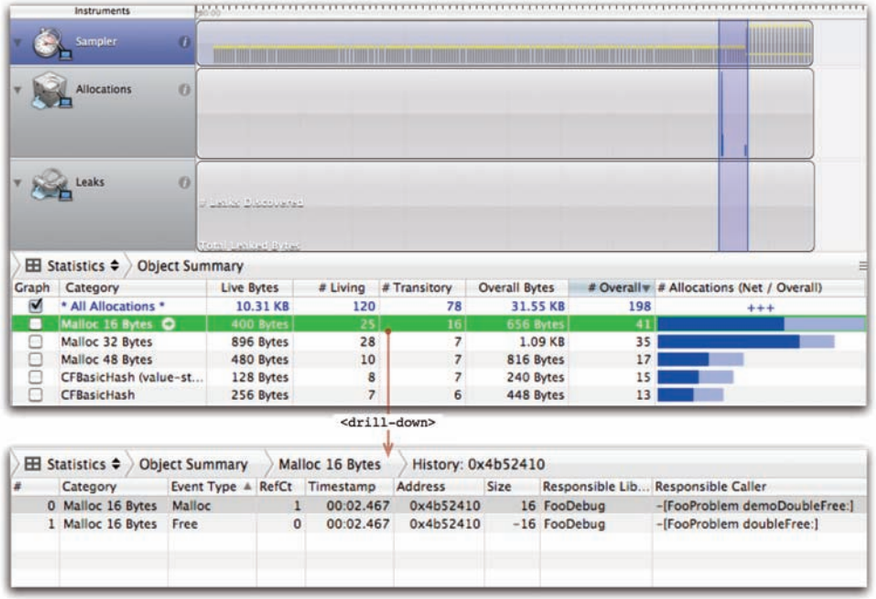


Figure 11.

`CFNumber` is the Core Foundation analogue of `NSNumber`. Upon its creation, the object starts with the expected retain count of 1. It is also marked for autorelease due to the use of a factory method. When the object is added to the array, its retention count goes up by 1. The `doubleFreeObjC:` method extracts the object from `objcArray`, and retain count goes down by 1.

But then `doubleFreeObjC:` sends the `CFNumber` object a release message. When `doubleFreeObjC:` exits, the autorelease pool reacts by sending its own release message to the same object. The retain count falls below zero and `EXC_BAD_ACCESS` occurs. One way to fix this problem is to remove the explicit release message (line 44) from `doubleFreeObjC:`.

Figure 11 tracks another double free, this time caused by the `demoDoubleFree:` action method from Listing Three. The last two spikes on the Track pane for the Allocations module mark the point before the app crashes with an `EXC_BAD_ACCESS`. Selecting these two spikes reveals the object types created before the crash. The type you want to examine is `Malloc-16`.

A drill-down on `Malloc-16` lists the pointers — one of which is `0x4b52410`, the pointer to the `tFoo` data structure. A drill-down on `0x4b52410` displays its allocation history. It reveals `tFoo` getting a reference count of 1 when first allocated in `demoDoubleFree:`. But `doubleFree:` disposes of `tFoo` with a call to `free()`, decreasing the count to 0. So when `demoDoubleFree:` tries to dispose of `tFoo` with `free()` as well, it will raise the aforementioned exception. Correcting this problem requires removing one of the calls to `free()`.

Figure 12 shows a memory hog in progress. The routine is the `demoMemoryHog:` action method (Listing Ten) creating an array of `NSNumber` objects. The Track pane showed a large swatch of allocations. Selecting

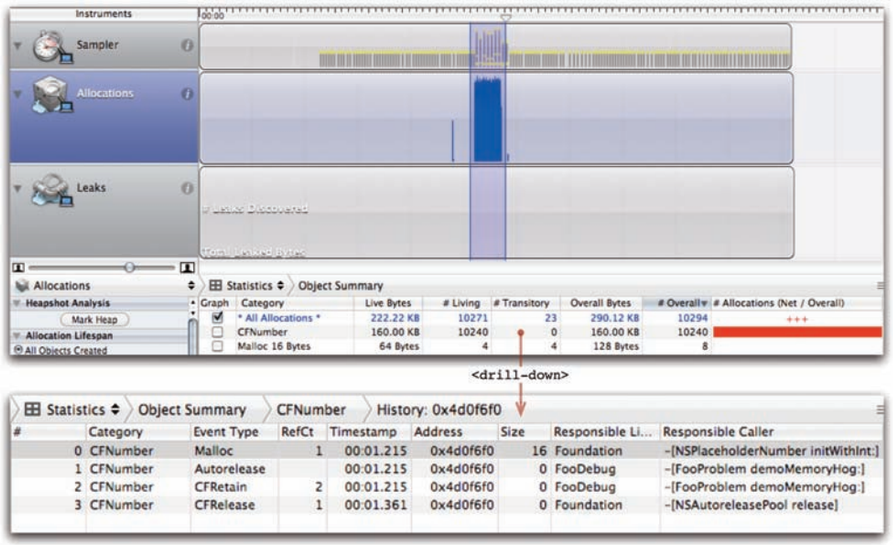


Figure 12.

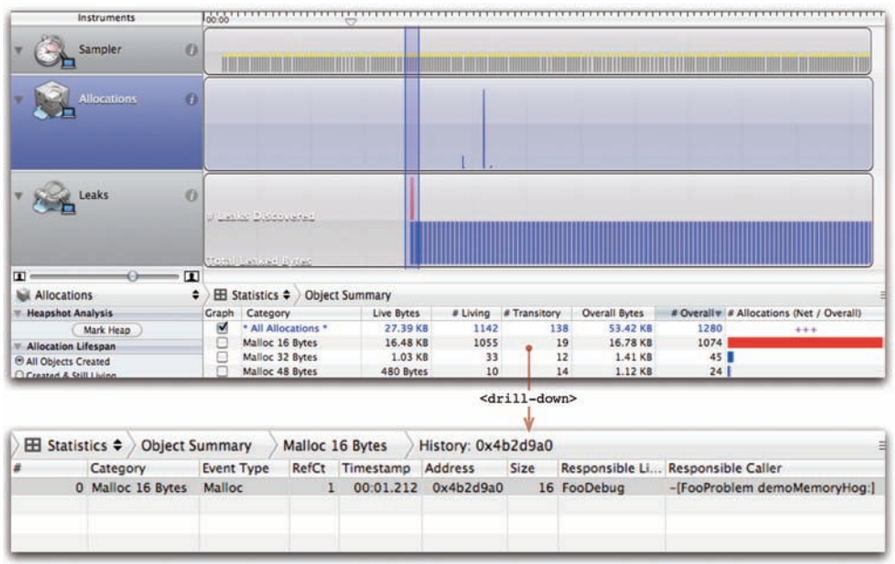


Figure 13.

the swatch displays the `CFNumber` objects, all 10,240 instances, taking up a total of 160 KB of memory. The graph on the Allocations (Net/Overall) column for `CFNumber` was a solid red.

Listing Ten

```
01 - (IBAction) demoMemoryHog:(id)aSrc
02 {
03     NSNumber *tNum;
04     NSInteger tIdx, tMax, tInt;
05
06     // set the maximum array size
07     tMax = 1024 * 10;
08     for (tIdx = 0; tIdx < tMax; tIdx++)
09     {
10          // create a test entry
11          tInt = rand();
12          tNum = [NSNumber numberWithInt:tInt];
13
14          // update the test array
15          [objcArray addObject:tNum];
16     }
17 }
```

A drill-down lists the addresses of each object (not shown); another drill-down reveals the allocation history for one `CFNumber` object (`0x4d0f6f0`). This memory hog is not serious, because the `CFNumber` objects are all marked for autorelease. But unless you try to restrict array growth, you will end up with a low-memory condition.

Figure 13 is another memory hog in progress. The routine is the `demoMemoryHog:` method from Listing Eight. The memory hog appears as a red spike on the Track pane next to the Leaks module. Selecting the spike reveals the 1055 instances of `Malloc-16` type still resident in

memory. Again, the graph on the Allocations (Net/Overall) column was a solid red for that type.

A drill-down on `Malloc-16` lists the pointer addresses of the instances and the routine that made them. And a drill-down on a pointer (shown as `0x4b2d9a0`) provides details for one instance. Memory consumption is small, a mere 16.48 KB. But if the linked-list grew unimpeded, it could lead to a low-memory condition.

Conclusion

In this article, I showed four common memory problems that can affect an iOS app, how to identify some these problems with Xcode's static analysis feature, and how to prepare an Instruments session and attach it to an iOS app process, as well as how to gather process data and examine that data for memory problems. Ensuring your code is frugal with memory usage always pays off when it comes to developing apps for iOS.

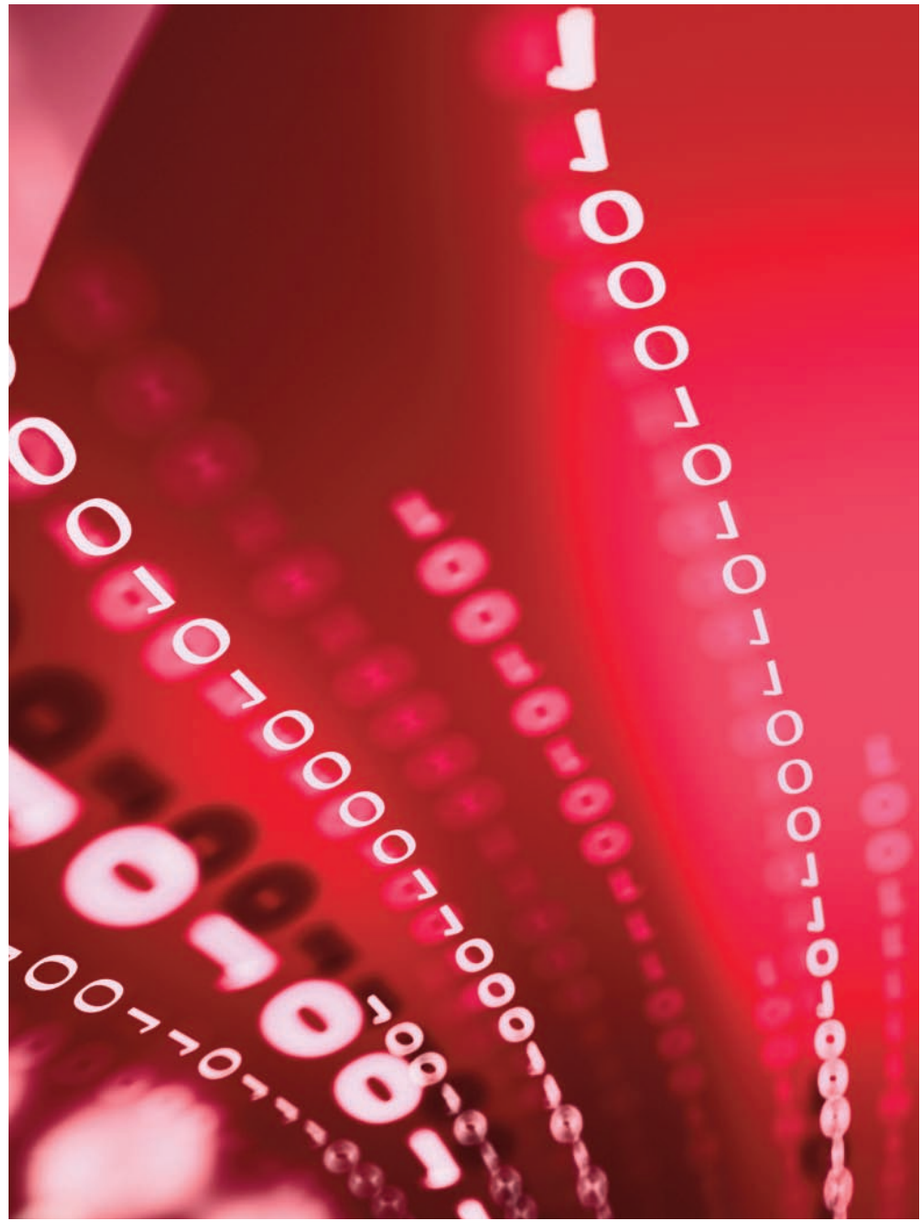
References

Memory Usage Performance Guidelines [PDF]. Mac Developer Library. http://is.gd/YKegFi

Instruments User Guide. [PDF] Mac Developer Library. http://is.gd/fdZnXk

— José Cruz is a freelance engineering writer based in British Columbia. He frequently contributes articles to MacTech, REALStudio Developer, and Dr. Dobb's.

Comment



Handling Touch Input on iOS6

Learn how to use the essential components of all touch-based interaction.

By Erica Sadun

The touch represents the heart of iOS interaction; it provides the core way that users communicate their intent to an application. Touches are not limited to button presses and keyboard interaction. You can design and build applications that work directly with users' gestures in meaningful ways. This article introduces direct manipulation interfaces that go far beyond prebuilt controls. I show how to create views that users can drag around the screen. I also discuss how to distinguish and interpret gestures, which are a high-level touch abstraction, and gesture recognizer classes, which automatically detect common interaction styles like taps, swipes, and drags.

Cocoa Touch implements direct manipulation in the simplest way possible: It sends touch events to the view you're working with. As an iOS developer, you tell the view how to respond. Before jumping into gestures and gesture recognizers, you should gain a solid foundation in this underlying touch technology. It provides the essential components of all touch-based interaction.

Each touch conveys information: where the touch took place (both the current and previous location), what phase of the touch was used (essentially mouse down, mouse moved, and mouse up in the desktop application world, corresponding to finger or touch down, moved, and up in the direct-manipulation world), a tap count (for example, single-tap/double-tap), and when the touch took place (through a time stamp).

iOS uses a responder chain to decide which objects should process touches. As the name suggests, responders are objects that respond to events and they act as a chain of possible managers for those events. When the user touches the screen, the application looks for an object to handle this interaction. The touch is passed along, from view to view, until some object takes charge and responds to that event.

At the most basic level, touches and their information are stored in `UITouch` objects, which are passed as groups in `UIEvent` objects. Each object represents a single touch event, containing single or multiple touches. This depends both on how you've set up your application to respond (that is, if you've enabled multi-touch interaction), and how the user touches the screen (that is, the physical number of touch points).

Your application receives touches in view or view-controller classes; both implement touch handlers via inheritance from the `UIResponder` class. You decide where to process and respond to touches. Trying to implement low-level gesture control in nonresponder classes has tripped up many new iOS developers.

Handling touches in views may seem counterintuitive. You probably expect to separate the way an interface looks (its view) from the way it responds to touches (its controller). Further, using views for direct touch interaction may seem to contradict Model-View-Controller design orthogonality, but it can be necessary and help promote encapsulation.

Consider the case of working with multiple touch-responsive subviews, such as game pieces on a chess board. Building interaction behavior directly into view classes allows you to send meaningful, semantically rich feedback to your main application while hiding implementation minutia. For example, you can inform your model that a pawn has moved to Queen's Bishop 5 at the end of an interaction sequence, rather than transmit a meaningless series of vector changes. By hiding the way the game pieces move in response to touches, your model code can focus on game semantics instead of view position updates.

Drawing presents another reason to work in the `UIView` class. When your application handles any kind of drawing operation in response to user touches, you need to implement touch handlers in views. Unlike views, view controllers don't implement the all-important `drawRect:` method needed for providing custom presentations.

Working at the view-controller level also has its perks. Instead of pulling out primary handling behavior into a secondary class implementation, adding touch management directly to the view controller allows you to interpret standard gestures, such as tap-and-hold or swipes, where those gestures have meaning. This better centralizes your code and helps tie controller interactions directly to your application model.

In the following sections and recipes, I discuss how touches work, how you can incorporate them into your apps, and how you connect what a user sees with how that user interacts with the screen.

Phases

Touches have lifecycles. Each touch can pass through any of five phases that represent the progress of the touch within an interface. These phases are as follows:

1. `UITouchPhaseBegan` — Starts when the user touches the screen.
2. `UITouchPhaseMoved` — Means a touch has moved on the screen.
3. `UITouchPhaseStationary` — Indicates that a touch remains on the screen surface, but that there has not been any movement since the previous event.
4. `UITouchPhaseEnded` — Gets triggered when the touch is pulled away from the screen.
5. `UITouchPhaseCancelled` — Occurs when the iOS system stops tracking a particular touch. This usually occurs due to a system interruption, such as when the application is no longer active or the view is removed from the window.

Taken as a whole, these five phases form the interaction language for a touch event. They describe all the possible ways that a touch can progress or fail to progress within an interface, and provide the basis for control for that interface. It's up to you as the developer to interpret those phases and provide reactions to them. You do that by implementing a series of responder methods.

Touches and Responder Methods

All subclasses of the `UIResponder` class, including `UIView` and `UIViewController`, respond to touches. Each class decides whether

and how to respond. When choosing to do so, they implement customized behavior when a user touches one or more fingers down in a view or window.

Predefined callback methods handle the start, movement, and release of touches from the screen. Corresponding to the phases you've already seen, the methods involved are as follows. Notice that `UITouchPhaseStationary` does not generate a callback.

- `touchesBegan:withEvent:` — Gets called at the starting phase of the event, as the user starts touching the screen.
- `touchesMoved:withEvent:` — Handles the movement of the fingers over time.
- `touchesEnded:withEvent:` — Concludes the touch process, where the finger or fingers are released. It provides an opportune time to clean up any work that was handled during the movement sequence.
- `touchesCancelled:WithEvent:` — Called when Cocoa Touch must respond to a system interruption of the ongoing touch event.

Each of these is a `UIResponder` method, often implemented in a `UIView` or `UIViewController` subclass. All views inherit basic nonfunctional versions of the methods. When you want to add touch behavior to your application, you override these methods and add a custom version that provides the responses your application needs.

Your classes can implement all or just some of these methods. For real-world deployment, you will always want to add a touches-cancelled event to handle the case of a user dragging his or her finger offscreen or the case of an incoming phone call, both of which cancel an ongoing touch sequence. As a rule, you can generally redirect a canceled touch to your `touchesEnded:withEvent:` method. This allows your code to complete the touch sequence, even if the user's finger has not left the screen. Apple recommends overriding all four methods as a best practice when working with touches.

Note that views have an exclusive touch mode that prevents touches from being delivered to other views in the same window. When enabled, this property blocks other views from receiving touch events. The primary view handles all touch events exclusively.

Touching Views

When dealing with many onscreen views, iOS automatically decides which view the user touched and passes any touch events to the proper view for you. This helps you write concrete direct-manipulation interfaces where users touch, drag, and interact with onscreen objects.

Just because a touch is physically on top of a view doesn't mean that a view has to respond. Each view can use a "hit test" to choose whether to handle a touch or to let that touch fall through to views beneath it. As you see in the recipes that follow, you can use clever response strategies to decide when your view should respond, particularly when you're using irregular art with partial transparency.

With touch events, the first view that passes the hit test opts to handle or deny the touch. If it passes, the touch continues to the view's superview and then works its way up the responder chain until it is handled or until it reaches the window that owns the views. If the window does not process it, the touch moves to the application instance, where it is either processed or discarded.

iOS supports both single- and multi-touch interfaces. Single-touch GUIs handle just one touch at any time. This relieves you of any responsibility to determine which touch you were tracking. The one touch you receive is the only one you need to work with. You look at its data, respond to it, and wait for the next event.

When working with multi-touch — that is, when you respond to multiple onscreen touches at once — you receive an entire set of touches. It is up to you to order and respond to that set. You can, however, track each touch separately and see how it changes over time, providing a richer set of possible user interaction. Recipes for both single-touch and multi-touch interaction follow later in this article.

Gesture Recognizers

With gesture recognizers, Apple added a powerful way to detect specific gestures in your interface. Gesture recognizers simplify touch design. They encapsulate touch methods, so you don't have to implement them yourself, and provide a target-action feedback mechanism that hides implementation details. They also standardize how certain movements are categorized, as drags or swipes, and so forth.

With gesture-recognizer classes, you can trigger callbacks when iOS perceives that the user has tapped, pinched, rotated, swiped, panned, or used a long press. Although their software development kit (SDK) implementations remain imperfect, these detection capabilities simplify development of touch-based interfaces. You can code your own for improved reliability, but most developers will find that the recognizers, as shipped, are robust enough for many application needs. You'll find several recognizer-based recipes in this article. Because recognizers all basically work in the same fashion, you can easily extend these recipes to your specific gesture recognition requirements.

Here is a rundown of the kinds of gestures that are built in to recent versions of the iOS SDK:

- Taps correspond to single or multiple finger taps onscreen. Users can tap with one or more fingers; you specify how many fingers you require as a gesture recognizer property and how many taps you want to detect. You can create a tap recognizer that works with single finger taps, or more nuanced recognizers that look, for example, for two-fingered triple-taps.
- Swipes are short, single- or multi-touch gestures that move in a single cardinal direction: up, down, left, or right. They cannot move too far off course from that primary direction. You set the direction you want your recognizer to work with. The recognizer returns the detected direction as a property.
- Pinches are when a user moves two fingers together. To pinch or unpinch, a user must move two fingers together or apart in a single movement. The recognizer returns a scale factor indicating the degree of pinching.
- Rotations are when a user moves two fingers at once either in a clockwise or counterclockwise direction, producing an angular rotation as the main returned property.
- Pans occur when users drag their fingers across the screen. The recognizer determines the change in translation produced by that drag.
- Long presses are when the user touches the screen and holds his or her finger (or fingers) there for a specified period of time. You can specify how many fingers must be used before the recognizer triggers.

Adding a Simple Direct Manipulation Interface

Your design focus moves from the `UIViewController` to the `UIView` when you work with direct manipulation. The view, or more precisely the `UIResponder`, forms the heart of direct manipulation development. You create touch-based interfaces by customizing methods that derive from the `UIResponder` class.

Recipe 1 centers on touches in action. This example creates a child of `UIImageView`, called `DragView`, and adds touch responsiveness to the class. Being an image view, it's important to enable user interaction (that is, set `setUserInteractionEnabled` to `YES`). This property affects all the view's children as well as the view itself. User interaction is generally enabled for most views, but `UIImageView` is the one exception that stumps most beginners; Apple apparently didn't think people would generally manipulate them.

Recipe 1 works by updating a view's center to match the movement of an onscreen touch. When a user first touches any `DragView`, the object stores the start location as an offset from the view's origin. As the user drags, the view moves along with the finger — always maintaining the same origin offset so that the movement feels natural. Movement occurs by updating the object's center. Recipe 1 calculates `x` and `y` offsets and adjusts the view center by those offsets after each touch movement.

Upon being touched, the view pops to the front. That's due to a call in the `touchesBegan:withEvent:` method. The code tells the superview that owns the `DragView` to bring that view to the front. This allows the active element to always appear foremost in the interface.

This recipe does not implement touches-ended or touches-cancelled methods. Its interests lie only in the movement of onscreen objects. When the user stops interacting with the screen, the class has no further work to do.

Recipe 1: Creating a draggable view.

```
@interface DragView : UIImageView
{
    CGPoint startLocation;
}
@end

@implementation DragView
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
        self.userInteractionEnabled = YES;
    return self;
}

- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate and store offset, and pop view into front if needed
    startLocation = [[touches anyObject] locationInView:self];
    [self.superview bringSubviewToFront:self];
}

- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate offset
    CGPoint pt = [[touches anyObject] locationInView:self];
    float dx = pt.x - startLocation.x;
    float dy = pt.y - startLocation.y;
    CGPoint newcenter = CGPointMake(
        self.center.x + dx,
        self.center.y + dy);

    // Set new location
    self.center = newcenter;
}
@end
```

The full sample project for Recipe 1 and the five other recipes in this article are available online at http://is.gd/TXr1yb (navigate to the first folder to find them.)

Adding Pan Gesture Recognizers

With gesture recognizers, you can achieve the same kind of interaction shown in Recipe 1 without working quite so directly with touch handlers. Pan gesture recognizers detect dragging gestures. They allow you to assign a callback that triggers whenever iOS senses panning.

Recipe 2 mimics Recipe 1's behavior by adding a recognizer to the view when it is first initialized. As iOS detects the user dragging on a `DragView` instance, the `handlePan:` callback updates the view's center to match the distance dragged.

This code uses what might seem like an odd way of calculating distance. It stores the original view location in an instance variable (`previousLocation`), then calculates the offset from that point each time the view updates with a pan detection callback. This allows you to use affine transforms or apply the `setTranslation:inView:` method; you normally do not move view centers, as done here. This recipe creates a `dx/dy` offset pair and applies that offset to the view's center, changing the view's actual frame.

Unlike simple offsets, affine transforms allow you to meaningfully work with rotation, scaling, and translation all at once. To support transforms, gesture recognizers provide their coordinate changes in absolute terms rather than relative ones. Instead of issuing iterative offset vectors, the `UIPanGestureRecognizer` returns a single vector representing a translation in terms of some view's coordinate system, typically the coordinate system of the manipulated view's superview. This vector translation lends itself to simple affine transform calculations and can be mathematically combined with other changes to produce a unified transform representing all changes applied simultaneously.

Here's what the `handlePan:` method looks like using straight transforms and no stored state:

```
- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    if (uigr.state == UIGestureRecognizerStateEnded)
    {
        CGPoint newCenter = CGPointMake(
            self.center.x + self.transform.tx,
            self.center.y + self.transform.ty);
        self.center = newCenter;

        CGAffineTransform theTransform = self.transform;
        theTransform.tx = 0.0f;
        theTransform.ty = 0.0f;
        self.transform = theTransform;

        return;
    }

    CGPoint translation = [uigr translationInView:self.superview];
    CGAffineTransform theTransform = self.transform;
    theTransform.tx = translation.x;
    theTransform.ty = translation.y;
    self.transform = theTransform;
}
```

Notice how the recognizer checks for the end of interaction, then updates the view's position and resets the transform's translation. This adaptation requires no local storage and would eliminate the need for a `touchesBegan:withEvent:` method. Without these modifications, Recipe 2 has to store previous state.

Recipe 2: Using a pan gesture recognizer to drag views.

```
@interface DragView : UIImageView
{
    CGPoint previousLocation;
}
@end

@implementation DragView
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *panRecognizer =
            [[UIPanGestureRecognizer alloc]
               initWithTarget:self action:@selector(handlePan:)];
        self.gestureRecognizers = @[panRecognizer];
    }
    return self;
}

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // Remember original location
    previousLocation = self.center;
}

- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    self.center = CGPointMake(previousLocation.x + translation.x,
        previousLocation.y + translation.y);
}
@end
```

Using Multiple Gesture Recognizers Simultaneously

Recipe 3 builds off the ideas presented in Recipe 2, but with several differences. First, it introduces multiple recognizers that work in parallel. To achieve this, the code uses three separate recognizers — rotation, pinch, and pan — and adds them all to the `DragView`'s `gestureRecognizers` property. It assigns the `DragView` as the delegate for each recognizer. This allows the `DragView` to implement the `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` delegate method, enabling these recognizers to work simultaneously. Until this method is added to return `YES` as its value, only one recognizer will take charge at a time. Using parallel recognizers allows you to, for example, both zoom and rotate in response to a user's pinch gesture.

Note that `UITouch` objects store an array of gesture recognizers. The items in this array represent each recognizer that receives the touch object in question. When a view is created without gesture recognizers, its responder methods will be passed touches with empty recognizer arrays.

Recipe 3 extends the view's state to include scale and rotation instance variables. These items keep track of previous transformation values and permit the code to build compound affine transforms. These compound transforms, which are established in Recipe 3's `updateTransformWithOffset:` method, combine translation, rotation, and scaling into a single result. Unlike the previous recipe, this recipe uses transforms uniformly to apply changes to its objects, which is the standard practice for recognizers.

Finally, this recipe introduces a hybrid approach to gesture recognition. Instead of adding a `UITapGestureRecognizer` to the view's recognizer array, Recipe 3 demonstrates how you can add the kind of

basic touch method used in Recipe 1 to catch a triple-tap. In this example, a triple-tap resets the view back to the identity transform. This undoes any manipulation previously applied to the view and reverts it to its original position, orientation, and size. As you can see, the touches began, moved, ended, and cancelled methods work seamlessly alongside the gesture recognizer callbacks, which is the point of including this extra detail in this recipe. Adding a tap recognizer would have worked just as well.

This recipe demonstrates the conciseness of using gesture recognizers to interact with touches.

Recipe 3: Recognizing gestures in parallel.

```
@interface DragView : UIImageView <UIGestureRecognizerDelegate>
{
    CGFloat tx; // x translation
    CGFloat ty; // y translation
    CGFloat scale; // zoom scale
    CGFloat theta; // rotation angle
}
@end

@implementation DragView
- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // initialize translation offsets
    tx = self.transform.tx;
    ty = self.transform.ty;
}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    if (touch.tapCount == 3)
    {
        // Reset geometry upon triple-tap
        self.transform = CGAffineTransformIdentity;
        tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;
    }
}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    [self touchesEnded:touches withEvent:event];
}

- (void) updateTransformWithOffset: (CGPoint) translation
{
    // Create a blended transform representing translation,
    // rotation, and scaling
    self.transform = CGAffineTransformMakeTranslation(
        translation.x + tx, translation.y + ty);
    self.transform = CGAffineTransformRotate(self.transform, theta);
    self.transform = CGAffineTransformScale(self.transform, scale, scale);
}

- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    [self updateTransformWithOffset:translation];
}

- (void) handleRotation: (UIRotationGestureRecognizer *) uigr
{
    theta = uigr.rotation;
    [self updateTransformWithOffset:CGPointZero];
}

- (void) handlePinch: (UIPinchGestureRecognizer *) uigr
{
    scale = uigr.scale;
    [self updateTransformWithOffset:CGPointZero];
}

- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
```

```
        (UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}

- (id) initWithImage:(UIImage *)image
{
    // Initialize and set as touchable
    if (!(self = [super initWithImage:image])) return nil;

    self.userInteractionEnabled = YES;

    // Reset geometry to identities
    self.transform = CGAffineTransformIdentity;
    tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;

    // Add gesture recognizer suite
    UIRotationGestureRecognizer *rot = [[UIRotationGestureRecognizer alloc]
        initWithTarget:self action:@selector(handleRotation:)];
    UIPinchGestureRecognizer *pinch = [[UIPinchGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePinch:)];
    UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePan:)];
    self.gestureRecognizers = @[rot, pinch, pan];
    for (UIGestureRecognizer *recognizer in self.gestureRecognizers)
        recognizer.delegate = self;

    return self;
}
@end
```

Resolving Gesture Conflicts

Gesture conflicts may arise when you need to recognize several types of gestures at the same time. For example, what happens when you need to recognize both single- and double-taps? Should the single-tap recognizer fire at the first tap, even when the user intends to enter a double-tap? Or should you wait and respond only after it's clear that the user isn't about to add a second tap? The iOS SDK allows you to take these conflicts into account in your code.

Your classes can specify that one gesture must fail in order for another to succeed. Accomplish this by calling `requireGestureRecognizerToFail:`. This is a gesture method that takes one argument, another gesture recognizer. This call creates a dependency between the object receiving this message and another gesture object. What it means is this: For the first gesture to trigger, the second gesture must fail. If the second gesture is recognized, the first gesture will not be.

In real life, this typically means that the recognizer adds a delay until it can be sure that the dependent recognizer has failed. It waits until the second gesture is no longer possible. Only then does the first recognizer complete. If you recognize both single- and double-taps, the application waits a little longer after the first tap. If no second tap happens, the single-tap fires. Otherwise, the double-tap fires, but not both.

Your GUI responses will slow down to accommodate this change; your single-tap responses become slightly laggy. That's because there's no way to tell if a second tap is coming until time elapses. You should never use both kinds of recognizers where instant responsiveness is critical to your user experience. Try, instead, to design around situations where that tap means "do something now" and avoid requiring both gestures for those modes.

Don't forget that you can add, remove, and disable gesture recognizers on the fly. A single-tap may take your interface to a place where it then makes sense to further distinguish between single- and double-taps. When leaving that mode, you could disable or remove the double-tap recognizer to regain better single-tap recognition. Tweaks like this limit interface slowdowns where needed.

Constraining Movement

One problem with the simple approach of the earlier recipes is that it's entirely possible to drag a view offscreen to the point where the

user cannot see or easily recover it. Those recipes use unconstrained movement. There is no check to test whether the object remains in view and is touchable. Recipe 4 fixes this problem by constraining a view's movement to within its parent.

It achieves this by limiting movement in each direction, splitting its checks into separate x and y constraints. This two-check approach allows the view to continue to move even when one direction has passed its maximum. If the view has hit the rightmost edge of its parent, for example, it can still move up and down.

Figure 1 shows a sample interface. The subviews (flowers) are constrained into the black rectangle in the center of the interface and cannot be dragged off-view. Recipe 4's code is general and can adapt to parent bounds and child views of any size.

Recipe 4: Bounded movement.

```
- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    CGPoint newcenter = CGPointMake(
        previousLocation.x + translation.x,
        previousLocation.y + translation.y);

    // Restrict movement into parent bounds
    float halfx = CGRectGetMidX(self.bounds);
    newcenter.x = MAX(halfx, newcenter.x);
    newcenter.x = MIN(self.superview.bounds.size.width - halfx,
        newcenter.x);

    float halfy = CGRectGetMidY(self.bounds);
    newcenter.y = MAX(halfy, newcenter.y);
    newcenter.y = MIN(self.superview.bounds.size.height - halfy,
        newcenter.y);

    // Set new location
    self.center = newcenter;
}
```

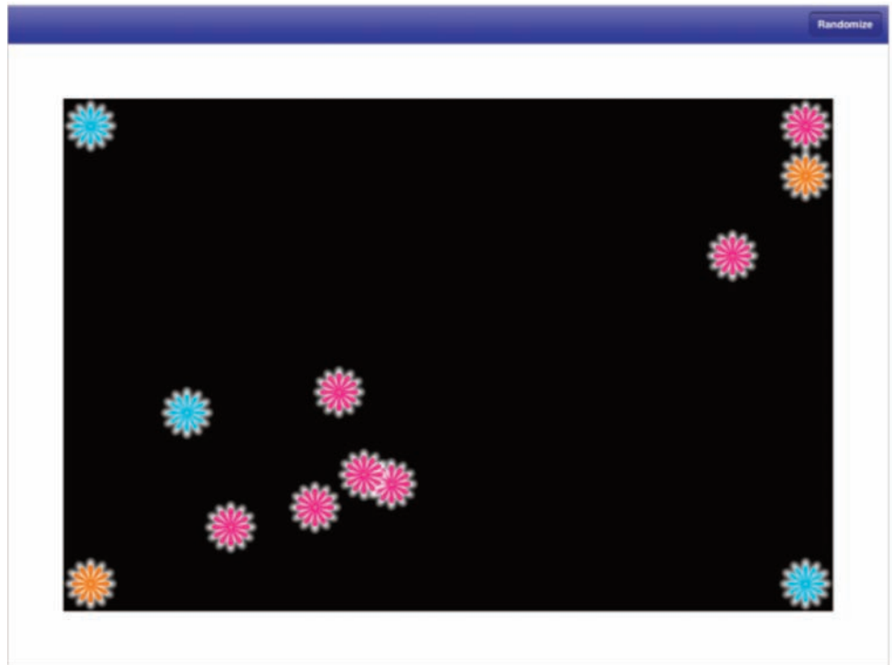


Figure 1: The movement of these flowers is bounded into the black rectangle.

Testing Touches

Most onscreen view elements for direct manipulation interfaces are not rectangular. This complicates touch detection because parts of the actual view rectangle may not correspond to actual touch points. Figure 2 shows the problem in action. The screenshot on the right shows the interface with its touch-based subviews. The shot on the left shows the actual view bounds for each subview. The light gray areas around each onscreen circle fall within the bounds, but touches to those areas should not "hit" the view in question.

iOS senses user taps throughout the entire view frame. This includes the undrawn area, such as the corners of the frame outside the actual

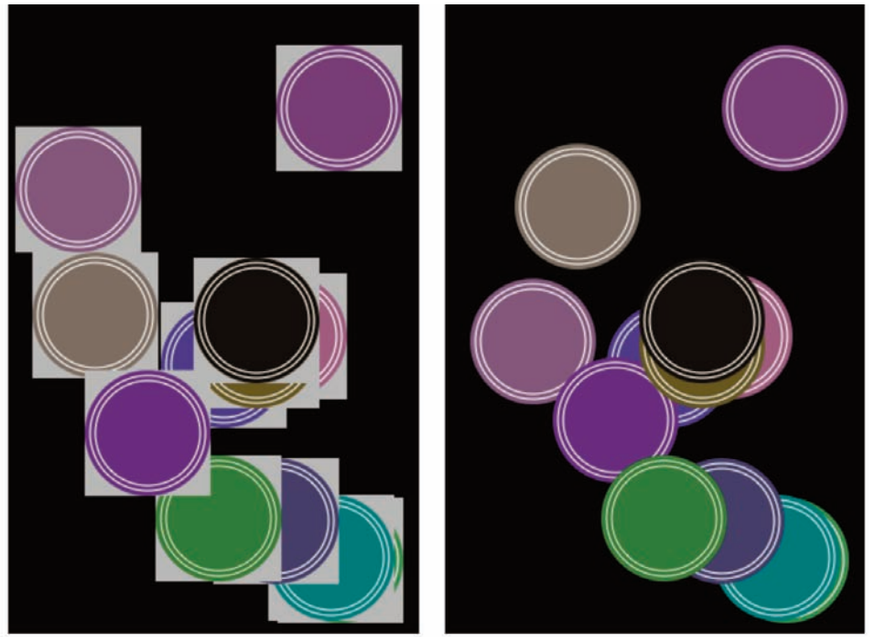


Figure 2: The application should ignore touches to the gray areas that surround each circle (left). The actual interface (right) uses a clear background (zero alpha values) to hide the parts of the view that are not used.

circles of Figure 2, just as much as the primary presentation. That means that unless you add some sort of hit test, users may attempt to tap through to a view that's "obscured" by the clear portion of the `UIView` frame. Visualize your actual view bounds by setting its background color, for example:

```
dragger.backgroundColor = [UIColor lightGrayColor];
```

This adds the backsplashes shown in Figure 2 (left) without affecting the actual onscreen art. In this case, the art consists of a centered circle with a transparent background. Unless you add some sort of test, all taps to any portion of this frame are captured by the view in question. Enabling background colors offers a convenient debugging aid to visualize the true extent of each view; don't forget to comment out the background color assignment in production code. Alternatively, you can set a view layer's border width or style.

Recipe 5 adds a simple hit test to the views, determining whether touches fall within the circle. This test overrides the standard `UIView`'s `pointInside:withEvent:` method. This method returns either `YES` (the point falls inside the view) or `NO` (it does not). The test here uses basic geometry, checking whether the touch lies within the circle's radius. You can provide any test that works with your onscreen views. (As you will see in Recipe 6, that test can be expanded for much finer control.)

Be aware that the math for touch detection on Retina Display devices remains the same as that for older units. The extra onboard pixels do not affect your gesture-handling math. Your view's coordinate system remains floating point with subpixel accuracy. The number of pixels the device uses to draw to the screen does not affect `UIView` bounds and `UITouch` coordinates; it simply provides a way to provide higher detail graphics within that coordinate system.

Do not confuse the point-inside test, which checks whether a point falls inside a view, with the similar-sounding `hitTest:withEvent:`. The hit test returns the topmost view (closest to the user/screen) in a view hierarchy that contains a specific point. It works by calling `pointInside:withEvent:` on each view. If the point inside method returns `YES`, the search continues down that hierarchy.

Recipe 5: Providing a circular hit test.

```
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    CGPoint pt;
    float HALFSIDE = SIDELENGTH / 2.0f;

    // normalize with centered origin
    pt.x = (point.x - HALFSIDE) / HALFSIDE;
    pt.y = (point.y - HALFSIDE) / HALFSIDE;

    // x^2 + y^2 = radius^2
    float xsquared = pt.x * pt.x;
    float ysquared = pt.y * pt.y;

    // If the radius <= 1, the point is within the clipped circle
    if ((xsquared + ysquared) <= 1.0) return YES;
    return NO;
}
```

Testing Against a Bitmap

Unfortunately, most views don't fall into the simple geometries that make the hit test from Recipe 5 so straightforward. The flowers shown in Figure 1, for example, offer irregular boundaries and varied transparencies. For complicated art, it helps to test touches against a bitmap. Bitmaps provide byte-by-byte information about the contents of an image-based view, allowing you to test whether a touch hits a solid portion of the image or should pass through to any views below.

Recipe 6 extracts an image bitmap from a `UIImageView`. It assumes that the image used provides a pixel-by-pixel representation of the view in question. When you distort that view (normally by resizing a frame or applying a transform), update the math accordingly. `CGPoints` can be transformed via `CGPointApplyAffineTransform()` to handle scaling and rotation changes. Keeping the art at a 1:1 proportion to the actual view pixels simplifies lookup and avoids any messy math. You can recover the pixel in question, test its alpha level, and determine whether the touch has hit a solid portion of the view.

This example uses a cutoff of 85. That corresponds to a minimum alpha level of 33% (that is, 85/255). This custom `pointInside:` method considers any pixel with an alpha level below 33% to be transparent. This is arbitrary. Use any level (or other test, for that matter) that works with the demands of your actual GUI. Unless you need pixel-perfect touch detection, you can probably scale down the bitmap so that it uses less memory and adjust the detection math accordingly.

Recipe 6: Testing touches against bitmap alpha levels.

```
// Return the offset for the alpha pixel at (x,y) for RGBA
// 4-bytes-per-pixel bitmap data
static NSUInteger alphaOffset(NSUInteger x, NSUInteger y, NSUInteger w)
    {return y * w * 4 + x * 4;}

// Return the bitmap from a provided image
NSData *getBitmapFromImage(UIImage *image)
{
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
        fprintf(stderr, "Error allocating color space\n");
        return NULL;
    }

    CGSize size = image.size;
    unsigned char *bitmapData = calloc(size.width * size.height * 4, 1);
    if (bitmapData == NULL)
    {
        fprintf (stderr, "Error: Memory not allocated!");
        CGColorSpaceRelease(colorSpace);
        return NULL;
    }
```

```
    CGContextRef context = CGBitmapContextCreate (bitmapData,
        size.width, size.height, 8, size.width * 4, colorSpace,
        kCGImageAlphaPremultipliedFirst);
    CGColorSpaceRelease(colorSpace );
    if (context == NULL)
    {
        fprintf (stderr, "Error: Context not created!");
        free (bitmapData);
        return NULL;
    }

    CGRect rect = CGRectMake(0.0f, 0.0f, size.width, size.height);
    CGContextDrawImage(context, rect, image.CGImage);
    unsigned char *data = CGBitmapContextGetData(context);
    CGContextRelease(context);

     NSData *bytes = [NSData dataWithBytes:data length:size.width
        * size.height * 4];
    free(bitmapData);

    return bytes;
}

// Store the bitmap data into an NSData instance variable
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        data = getBitmapFromImage(anImage);
    }
    return self;
}

// Does the point hit the view?
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    if (!CGRectContainsPoint(self.bounds, point)) return NO;
    Byte *bytes = (Byte *)data.bytes;
    uint offset = alphaOffset(point.x, point.y, self.image.size.width);
    return (bytes[offset] > 85);
}
```

Summary

`UIViews` and their underlying layers provide the onscreen components your users see. Touch input lets users interact directly with views via the `UITouch` class and gesture recognizers. As this article has shown, even in their most basic form, touch-based interfaces offer easy-to-implement flexibility and power. In a future article, we will expand your skill set with onscreen drawing, touch-based painting, creating smooth drawings, multi-touch interactions, and more.

— Erica is an expert in iOS development and writes frequently on the topic. She holds a Ph. D. in Computer Science from Georgia Tech's Graphics, Visualization and Usability Center. This article was excerpted from a late draft of her **Core iOS 6 Developer's Cookbook, 4th Ed.** *(http://is.gd/DgwDcn). Although it has been reviewed technically, final presentation and content in the book may vary from what was presented here.*

Comment

From the Vault

Managing Memory on iOS

This feature from last December shows how understanding memory usage patterns on Apple devices can help you program to conserve memory very effectively.

— *DDJ*

By José R.C. Cruz

One challenge faced by new iOS developers is how to work with the limited memory on Apple's handheld products. Products must optimize memory use, avoid leaks, and reduce the overall footprint.

This article explains how an iOS application should manage its allocated memory. It describes the lifecycle of a Cocoa object and how that cycle differs on iOS, and then explains how to reduce the memory footprint and to prevent memory leaks. It also explains how to detect and react to a low-memory signal from iOS.

Readers must have a working knowledge of Xcode and Cocoa.

Life of a Cocoa Object

A typical Cocoa object undergoes three distinct stages in its lifecycle (Figure 1). First, the object is created. This is done by sending an `alloc` message to the appropriate class. The class reserves the memory needed by the object and returns the object itself as a result. For example, the statement below creates an instance of the `NSString` class and stores it in the variable `tFoo`:

```
tFoo = [NSString alloc];
```

If the allocation fails due to insufficient memory, the class will return a `nil` object.

Next, the object is initialized. This is where it assumes a default state and value. It is where the object defines its delegates and prepares its parent. Initialization happens when the object gets an `init` message. Using our `NSString` example, this next statement initialize the object to a `null` string:

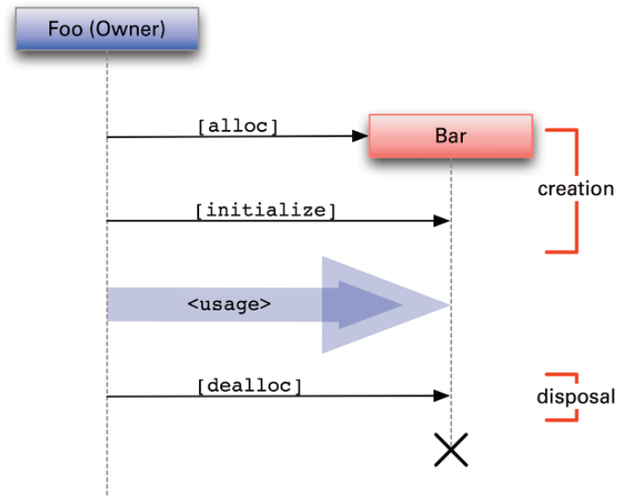


Figure 1.

```
[tFoo init];
```

Now most Cocoa classes have custom methods with which to initialize their respective objects. And many of those methods have two or more arguments to pass data or states to the object. In the case of `NSString`, for instance, the method `initWithString` gives the object with an initial string value:

```
[tFoo initWithString:@"foobar"];
```

Finally, the object reaches a time when it has to be disposed of. It cleans up after itself and frees up all the memory used by its properties and methods. Disposal usually happens when the object gets a `dealloc` message:

```
[tFoo dealloc];
```

But this could cause problems, especially when two or more other objects have to work with the affected object. Thankfully, there is a better way of disposing a Cocoa object, and it involves the use of reference counts.

Objects on Reference

By default, a newly created object has a reference count of 1. When that count hits 0, the object starts disposing of itself. In short, it literally self-destructs.

So to dispose of the object, send it a release message:

```
[tFoo release];
```

This will decrease the object's reference count by 1. The object runs its `dealloc` code, as well those of its parent. Once the object is disposed of, its variable then points to a "bad" address.

But suppose we want to add the object to a collection, or we want to use the object as a return value. For these cases, we must keep the object from self-destructing on its own.

To prevent disposal, send a `retain` message to the object as follows:

```
[tFoo retain];
```

This will increase the object's reference count by 1. With a count greater than 1, the object stays valid and will be unable to dispose of itself.

Now it is important that the object gets an equal set of `retain`s and `release`s (Figure 2). If the number of `retain`s exceeds those of `release`s by at least one, the end result is a memory leak. Conversely, if the number of `release`s is greater, the result is a bad access error.

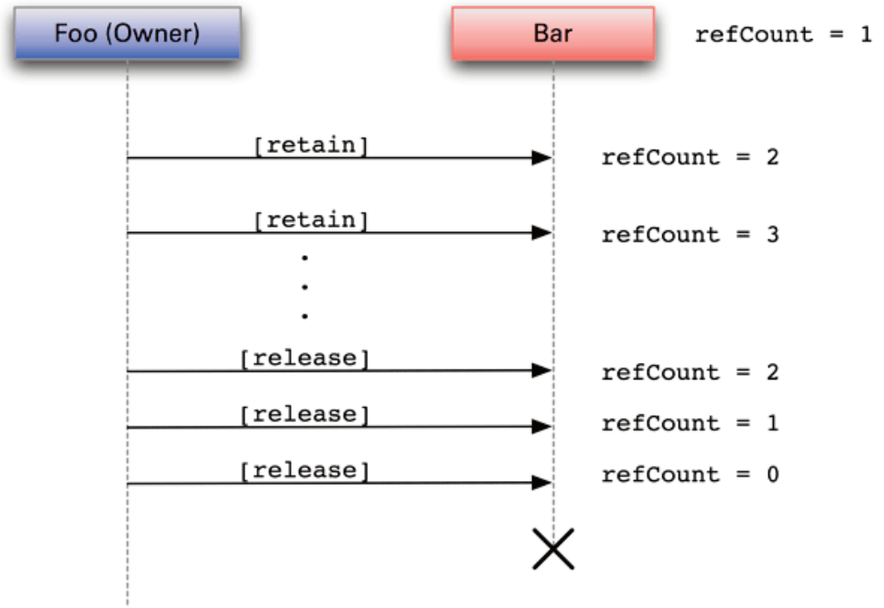


Figure 2.

An alternative way to dispose of an object is to mark it for `autorelease`. The marked object goes into an `autorelease` pool, created just before the application starts its event loop. Periodically, the pool checks its collection of objects, locating those with a reference count of 1, which are out of scope. When such an object is found, the pool disposes of it with a `release` message.

To place an object into the autorelease pool, send it an autorelease message:

```
[tFoo autorelease];
```

To prevent the pool from disposing of the object prematurely, use the `retain` message as described earlier.

Most Cocoa classes, however, can produce objects already marked for `autorelease`. This is done by using any one of the factory methods from each class. Consider again our `NSString` example. Its factory method `stringWithString` will create and initialize an instance of `NSString`. Furthermore, that same instance will be slated for `autorelease`.

```
tFoo = [NSString stringWithString:@"foobar"];
```

The iOS Difference

Now the same Cocoa object will have a similar lifecycle on iOS. Creating and initializing an object use the same messages. Retention and disposal are also the same. This is not surprising, of course, since iOS is a variant of MacOS X, but one optimized for handheld use.

iOS has its own notable quirks. First, it has a smaller amount of physical memory than its desktop cousins. Physical memory can be as small as 256 MB on an iPhone 3GS, or 512 MB on an iPad2. The system itself takes up at least 64 MB of that memory for its own needs. This leaves a smaller amount for all the applications to share. Flash memory, which figures in the gibabyte range, is used solely for storage. Furthermore, that same physical memory is not user-upgradeable.

Second, though iOS has the same virtual memory engine as OS X, its engine does not write out inactive resources to volatile pages. Instead, it expects each iOS app to dispose of its unneeded resources and free up the occupied memory.

Third, iOS favors the active application session, which has the user's immediate attention. It will signal background tasks and inactive apps to frequent memory purges. In severe cases, iOS may terminate the apps themselves.

Finally, as of this writing, iOS does not have any garbage collection service. Instead, it expects each application to manage its memory share properly and frugally. Again, iOS will terminate those apps that habitually hog the memory store.

On Optimizing Footprint

Now the first step to prepare an application for iOS is to reduce its memory footprint. Too large a footprint can degrade an application's performance and that of its host system. It reduces the amount of available memory and marks the application for immediate termination.

Apple offers four guidelines on how to reduce an app's memory footprint. Follow these guidelines closely when working with your iOS projects.

1. Locate and fix all possible memory leaks.

As stated earlier, a memory leak happens when a Cocoa object fails to dispose of itself properly. It may be due to one too many retains, leaving the object with a reference count of 1 or more. It may even be

due to the object using `malloc()` to allocate itself some memory, then failing to call `free()` to release said memory.

Consider the sample class in Listing One. This class defines a typical view controller. It declares three properties, which link the controller to three widgets on its window view (lines 5-7). Then it uses the `@property` and `@synthesize` keywords to declare and define the accessors for those outlets (lines 11-13, 23-25).

Listing One

```
01 -- CLASS:CONTROLLER:DECLARATION
02 @interface MyViewController :
      UIViewController <UITextFieldDelegate>
03 {
04    // -- properties:outlets
05    UITextField *textField;
06    UILabel *label;
07    NSString *string;
08 }
09
10 // -- accessors:outlets
11  @property (nonatomic, retain) IBOutlet UITextField
*textField;
12 @property (nonatomic, retain) IBOutlet UILabel *label;
13 @property (nonatomic, copy) IBOutlet NSString *string;
14
15 // -- methods:actions
16 - (IBAction) changeGreeting:(id)sender;
17 @end
18
19
20 -- CLASS:CONTROLLER:DEFINITION
21 @implementation MyViewController
22 // -- accessors:outlets
23 @synthesize textField;
24 @synthesize label;
25 @synthesize string;
26
27 // -- The user has entered new text data
28 - (IBAction) changeGreeting:(id)sender;
29 {
30    // read the entered string
31    self.string = textField.text;
32
33    // prepare the name string
34    NSString *nameString = string;
35
36    // should a default name be used?
37    if ([nameString length] == 0)
38    {
39          nameString = @"World";
40    }
41
42    // prepare the string data
43    NSString *greeting =
44       [[NSString alloc] initWithFormat:@"Hello, %@!",
     nameString];
45
46    // display the string
47    label.text = greeting;
48    [greeting release];
49 }
50
51 // The user has pressed the <Return> key
52 - (BOOL)textFieldShouldReturn:(UITextField *)aFld
53 {
54    // check the calling widget
55    if (aFld == textField)
56    {
57          [textField resignFirstResponder];
58    }
59    return (YES);
60 }
61
62
63 // -- The view has been disposed
64 - (void)viewDidUnload
65 {
66    // pass the message to the parent
67     [super viewDidUnload];
68 }
```

```
69
70
71 // -- The controller is about to be destroyed
72 - (void)dealloc
73 {
74    // dispose the following outlets
75    [textField release];
76    [label release];
77    [string release];
78
79    // pass the message to the parent
80     [super dealloc];
81 }
82 @end
```

Note the three lines in the controller's `dealloc` routine (lines 75-77). They send a release message to each of the outlets. But what if we forget to add these three lines? The result will be each outlet remaining valid after the view controller has self-destructed. We then get a small leak, which grows every time the iOS app recreates and disposes of the same controller.

Next, consider the sample method in Listing Two. This method gets an `NSString` input, which holds a path string. First, the method copies the string into the local `tNom` (line 9). It uses the instance method `pathComponents` to separate the path into its component items (line 12). The separated items are then returned in the form of an `NSArray`. Next, the method reads the last item of that array (15). If the array has only two entries, a root separator and name, the method prepares an empty string (line 17). And it returns the result as an `NSString` object.

Listing Two

```
01 // Return the file or directory name
02 - (NSString *)getItemName:(NSString *)aPth
03 {
04    NSString *tNom;
05    NSArray *tTmp;
06    NSInteger tIdx;
07
08    // create a string object
09    tNom = [NSString stringWithString:aPth];
10
11    // extract a path item
12    tTmp = [tNom pathComponents];
13    tIdx = [tTmp count]
14    if (tIdx > 1)
15          tNom = [tTmp objectAtIndex:(tIdx - 1)];
16    else
17          tNom = [NSString string];
18
19    // return the extraction result
20    return (tNom);
21 }
```

Note that all `NSString` instances are made with calls to factory methods. This means these instances are marked for `autorelease`. But suppose we send a retain message to the final instance:

```
[tNom retain];
```

At first, this looks innocuous. After all, it only increases the reference count by 1. But if we fail to send a matching release message, this `NSString` instance will remain in the pool, taking up precious memory, thus creating a memory leak.

2. Use smaller resources.

Each time an iOS loads a resource, that resource has to take up memory. The resource may stay resident for the duration of a view or task or, in some cases, for the entire app session. Since loaded resources take up precious memory space, we must try to keep their sizes as small as possible.

Consider graphics files, for instance. They hold the icons and logos that appear on an iOS view. The preferred format is a 32-bit PNG, usually no larger than 64x64 square pixels (for an iPhone target). To prepare the graphics file, use a preprocess tool such as `pngcrush` or `ImageOptim`. Both resample and recompress the file for the smallest possible size and load speed. Usage notes for both tools are available on their respective websites.

Now consider text files. These hold text localized for a specific region. Their data should be rendered in serialized form, as opposed to Unicode or UTF-8. This can cut down the extraneous bytes needed for each text character.

Listing Three shows one way to handle serialized text. First, the method `loadSerialText` gets the path to the text resource `foobar.text` (line 13). Having a valid path, the method loads the resource into an `NSData` object (line 18). It then extracts the raw character bytes and uses them to create an `NSString` object (lines 22-28).

Listing Three

```
01 // Load the serialised text data
02 - (NSString *)loadSerialText
03 {
04     NSString *tPth, tStr;
05     NSData *tDat;
06     NSUInteger tLen;
07     char *tBuf;
08
09     // prepare the default result
10     tDat = [NSString string];
11
12     // locate the text resource
13     tPth = [[NSBundle mainBundle] pathForResource:@"foobar"
14                                   ofType:@"text"];
15     if (tPth != nil)
16     {
17           // read the resource data
18           tDat = [NSData dataWithContentsOfFile:tPth];
19           if (tDat != nil)
20           {
21           // extract the data bytes
22           tLen = [tDat length];
23           tBuf = malloc(tLen * sizeof(char));
24           [tDat getBytes:tBuf length:tLen];
25
26           // create the string object
27           tStr = [NSString stringWithCString:(const char *)tBuf
28                  encoding:NSNonLossyASCIIStringEncoding];
29
30           // dispose the buffer
31           free(tBuf);
32           }
33     }
34
35     // return the retrieval result
36     return (tStr);
37 }
38
39 // Save the serialised text data
40 - saveSerialText:(NSString *)aTxt
41 {
42     NSString *tPth;
43     NSData *tDat;
44     NSUInteger tLen;
45     char *tBuf;
46     BOOL tChk;
47
48     // prepare the character buffer
49     tLen = [aTxt length];
50     tBuf = malloc(tLen * sizeof(char));
51
52     // initialise the buffer
53     tChk = [aTxt getCString:tBuf
54            maxLength:tLen encoding:NSNonLossyASCIIStringEncoding];
55
56     // locate the text resource
57     tPth = [[NSBundle mainBundle] pathForResource:@"foobar"
58                                   ofType:@"text"];
59
60     // write out the buffer
61     tDat = [NSData dataWithBytes:(const void *)tBuf
62            length:tLen];
63     tChk = [tDat writeToFile:tPth atomically:YES];
```

```
64
65      // dispose the buffer
66      free(tDat);
67  }
```

The method `saveSerialText` gets an `NSString` for input and extracts the raw character bytes (lines 49-54). It locates the resource file `foobar.text` (line 60) and stores the raw bytes into an `NSData` object (lines 61-62). Finally, it writes the object itself to the resource (line 63).

3. Avoid pre-fetching.

Some developers let their applications load their resource at launch time. Their intent is to avoid reloading often-used resources and to make those resources readily available. While this practice may be sensible with OS X apps, it will make an iOS app a memory hog.

A better way is to load those resources on demand. Then discard the resource when it is no longer needed. Doing so cuts down the memory needed by the app, and avoids inducing a low-memory state.

4. Optimize for size.

Executable size can also influence the overall memory footprint. When an iOS app launches, it loads as much of its executable code into memory as possible. The code stays resident until the application terminates.

Thus, it is important that we keep the executable size as small as possible. The best way to achieve this is to compile the project with an `mthumb` option. This option directs GCC to optimize the final executable by size. We get a smaller executable size, at the cost of a slight decrease in performance.

Figure 3 shows how to prepare the compile settings. First, choose "Edit Project Settings" from Xcode's Project menu to get the settings

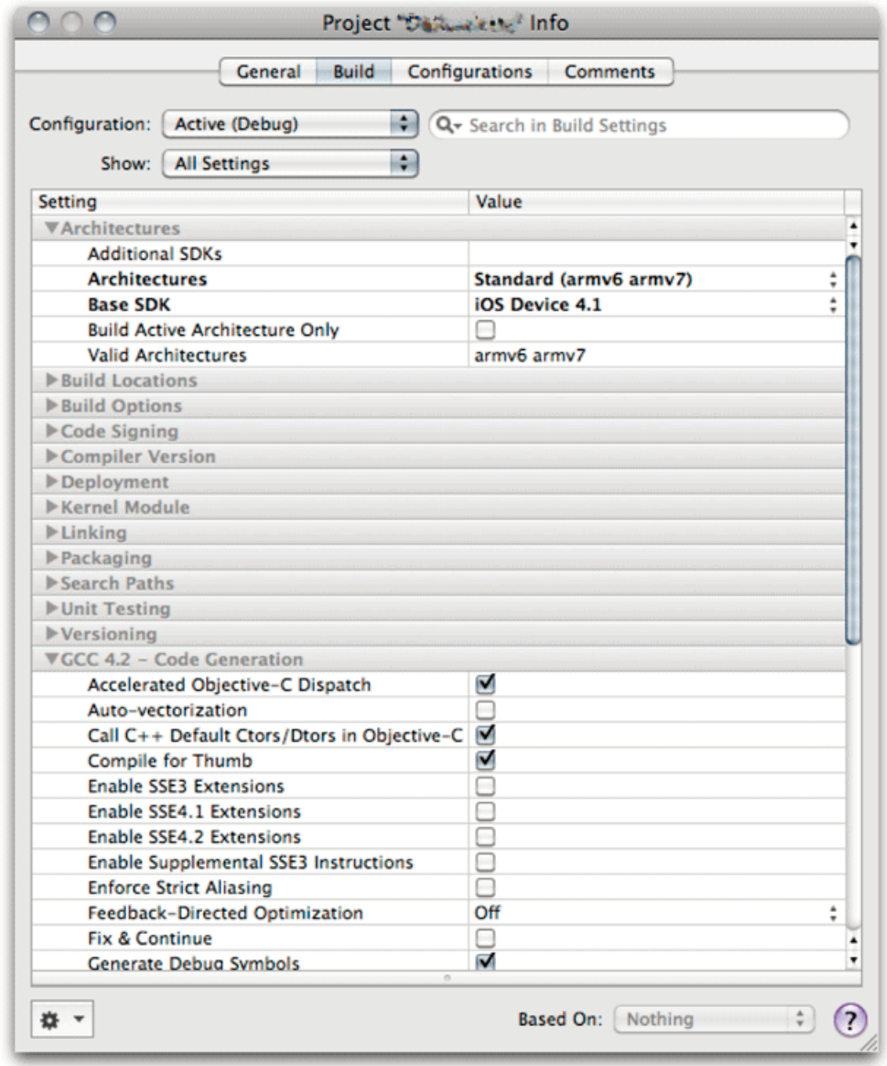


Figure 3.

dialog. Locate the group "GCC 4.2 - Code Generation," and check the option "Compile for Thumb." If the option is not set, click to set the adjacent checkbox.

Most iOS projects benefit greatly with size optimization, but there is a small group of projects where the costs of optimization may outweigh its benefits. These projects perform a large number of floating-point tasks while using arm6 opcodes. If their executables are optimized for size, their runtime performance may be severely reduced. In this situation, switching off the mthumb option may be a better choice.

Managing Memory

Apple also provides three guidelines on how an iOS app should manage its share of memory. These guidelines take into account the lack of garbage collection and the lack of volatile paging. So let us go through them, shall we?

1. Reduce the number of autoreleased objects.

Naturally, objects marked for autorelease linger longer that those with normal retain/release cycles. When there are too many autoreleased objects, the autorelease pool will need more memory to maintain its collection of objects.

Consider the sample method back in Listing Two. This method uses three autoreleased objects to perform its task. Two of those objects, `tNom` and `tTmp`, (lines 9,12) are valid only with the method scope and are disposed of on exit. The third object, also named `tNom`, (line 20) stays valid outside scope due to being a return value. Its disposal is then left to the calling method.

Now consider the sample method in Listing Four.

Listing Four

```
01 // Simple word parser
02 - (NSArray *)str2word:(NSString *)aTxt
03 {
04    NSArray *tWrd;
05    NSCharacterSet *tSet;
06    NSUInteger tLen;
07
08    // parametre check
09    tLen = [aTxt length];
10    if (tLen == 0)
11          tWrd = [NSArray array];
12    else
13    {
14          // prepare the set of tokens
15          tSet = [NSCharacterSet
16                characterSetWithCharactersInString:
            @".?:!,; *&()%#@_-{}[]"];
17
18          // divide the text into words
19          tWrd =
       [aTxt componentsSeparatedByCharactersInSet:tSet];
20    }
21
22    // return the extraction results
23    return (tWrd)
24 }
```

At first, this one appears to use only two `autoreleased` objects (lines 15, 23). Object `tSet` is valid only within the method scope, while object `tWrd` stays valid outside scope. What is not obvious is that `tWrd` is an array of `NSString` objects. Since those objects are made by the factory method `componentsSeparatedByCharactersInSet` (line 19), they are all marked for `autorelease`. So if the input `NSString` argument `aTxt` has a particularly long string, the `autorelease` pool may end up using more memory to hold that array.

2. Place limits on resource loading.

Most iOS apps load their resources in their entirety. This is not a problem when the resource itself is reasonably small. But suppose the resource is at least a quarter the size of available memory. Then loading it will take up more space, leaving less memory for other tasks.

A better way to deal with large resources is to load the data piecemeal, again on demand. Use the library routines `mmap()` and `munmap()` to copy only the needed portions into memory. Alternatively, you can prepare the resource as a database. Then use Core Data to access each resource portion as a record.

3. Avoid unbound problem sets.

Unbound sets are sets with no discernible limits. They could be data structures that kept growing with every use. They could be routines that use repetition to perform their tasks.

Listing Four was a good example of an unbound set as a data structure. There, the data supplied by the `NSString` argument `aTxt` dictates the size of the `NSArray` object `tWrd`. If the string data consists of hundreds of words, then the array may get hundreds of entries. If the data has just a dozen or so words, the same array will have less entries.

Thus, there is no easy way to determine the final array size by just examining the input string. Perhaps a better solution is to rewrite the routine so it extracts each word in situ.

Listing Five contains an example of an unbound set in the form of a routine. This one gives the root value nearest to the argument `aX`. It uses a basic Newton-Raphson algorithm (lines 7-9) to compute a possible root for a given polynomial. If the root does not converge, the routine calls itself, passing along the computed value as input (lines 12-14).

Listing Five

```
01 // Computing for a polynomial root
02 - (float)doRoot:(float)aX
03 {
04    float *tZer, tDel, tNum, tDen;
05
06    // calculate using Newton-Raphson algorithm
07    tNum = poly_func(aX);
08    tDen = poly_deriv(aX);'
09    tZer = aX - (tNum / tDen)
10
11    // validate the result
12    tDel = abs(tZer - aX)
13    if (tDel > 0.0000001)
14          tZer = doRoot(tZer);
15
16    // return the zero result
17    return (tZer);
18 }
```

Since the routine uses recursion, its stack grows with each computation. This becomes a problem when the stack needs more memory than what is available. Perhaps a way to avoid this problem is to rewrite the routine to use iteration, not recursion.

When Memory Gets Low

The moment the amount of free memory hits a low point, iOS starts warning its apps, both active and inactive. The apps must respond by releasing any unneeded objects and disposing of any resources that can be rebuilt or reloaded. If an app fails to respond as such, it can be marked for termination.

There are two ways to trap a low-memory warning. One way is with the method `applicationDidReceiveMemoryWarning`. This is a delegate method, declared by the protocol `UIApplicationDelegate`. Its

single input argument is a pointer to the active `UIApplication` object. If the method is undefined, the default response of `UIApplication` is to terminate its session.

Listing Six shows how the delegate method might be implemented. Here, the method sends a `clearFilter` message to its data model `fooData` (line 6). Then it retrieves the top-most view controller and sends a `resetView` message to that controller (lines 9-11). This assumes, of course, that the controller has a valid `resetView` method.

Listing Six

```
01 - (void)applicationDidReceiveMemoryWarning:
       (UIApplication *)anApp
02 {
03    UIViewController *tTop;
04
05    // clear the filter cache
06    [self.fooData clearFilter];
07
08    // reset the active view controller
09    tTop = self.navigationController.topViewController;
10    if (tTop != nil)
11          [tTop resetView];
12 }
```

Another way to trap a low-memory warning is with the method `didReceiveMemoryWarning`. This one is an instance method, declared and defined by the `UIViewController` class. Its default response is to dispose of the view handled by that controller.

Listing Seven shows how `didReceiveMemoryWarning` might be overridden. The method just sends the same `resetView` message to the controller itself (line 4). But unlike the `UIApplication` delegate method, this one gets invoked for every controller present in the view stack.

Listing Seven

```
01 - (void)didReceiveMemoryWarning
02 {
03    // reset the active view
04    [self resetView];
05 }
```

The iPhone simulator can easily simulate a low-memory state. With it, we can test code written to respond to that state. To prepare the Xcode project, first make sure the simulator is the target device. Near the top-left corner of the project window, click the "Overview" pop-up menu and choose the menu item "Simulator." Also, from the same menu, choose the menu item "Debug" to enable the source-level debugger.

Next, place a breakpoint on any of the low-memory traps previously described. Rebuild the project by choosing "Build" from the Xcode "Build" menu. Then choose "Debug" from the "Run" menu to start a debug session.

Xcode will start up the simulator, load in the executable, and launch it afterwards. Once the app's main window appears, click the simulator's "Hardware" menu and choose "Simulate Memory Warning." This will cause the simulator to generate a low-memory signal. If all goes well, Xcode will come up front and highlight one of the breakpoints. To bring up the source-level debugger, choose "Debugger" from the Xcode "Run" menu.

Conclusion

Proper memory management is essential for a stable, well-behaved iOS application. The application must be frugal with its memory use, and it must know how to react in a low-memory situation.

In this article, we studied the lifecycle of a typical Cocoa object. We learned how iOS differs from its OS X cousin in terms of memory support. We reviewed guidelines on how to reduce our application's memory footprint, and on how to manage its limited memory. And we studied how to trap a low-memory signal, as well as test the trap on the iPhone simulator.

References

Apple Developer Connection. How Memory Management Works. Cocoa Fundamentals Guide. Apple Inc. December 2010. http://is.gd/IEJ0XR

Apple Developer Connection. Practical Memory Management. Memory Management Programming Guide. Apple Inc. December 2010. http://is.gd/Ejs57j

Apple Developer Connection. Tips for Allocating Memory. Memory Usage Performance Guide. Apple Inc. July 2008. pp. 15-23. http://is.gd/ZIZ3bu

Apple Developer Connection. Use Memory Efficiently. iOS Application Programming Guide. Apple Inc. December 2010. pp. 15-23. http://is.gd/ddt2Zn

Robert Clair. VTM Seattle. iPhone OS Memory Management. Seattle, WA. April 25, 2010. PDF. http://is.gd/RXi97F

Joel Ivory Johnson. Getting Started with iPhone and iOS Development. June 21, 2010. http://is.gd/pG0wxT

Stephen Kochan. ClassroomM, Inc. Really, Really Understanding Memory Management. October 17, 2010. PDF. http://is.gd/9fUR4Q

Brent Simmons. How I Manage Memory. June 28, 2010. http://is.gd/u2GhvC

Related Articles

Generating Code Metrics and Estimating Costs with Xcode http://is.gd/BWu8dK

Cocoa Memory Management (2006) http://is.gd/O0wxYi

— José R.C. Cruz is an engineering writer who frequently writes about iOS for Dr. Dobb's. He resides in North Vancouver, British Columbia.

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

CHRONIC REQUIREMENTS PROBLEMS

Three problems plague requirements and cost software projects untold dollars. Not least of these is failing to collect new requirements after a project is delivered and running.

http://www.drdobbs.com/240012797

TOTAL ECLIPSE OF ARDUINO

Al Williams would like to say that he really likes the Arduino. He'd like to say that, but just can't.

http://www.drdobbs.com/240143006

DEBUGGING: ART OR SCIENCE?

Every bug is different, so saying something that applies to the act of debugging requires finding something general to say about a bunch of unrelated specific cases that defy generalization by their very nature.

http://www.drdobbs.com/240142870

TESTING BROWSER APPS

The QUnit framework makes it easy to test Web apps directly in the browser. By showing the red/green results in the browser with links to the failing tests, it makes it possible to write, test, and correct quickly.

http://www.drdobbs.com/240009260

STORAGE LAYOUT OF POLYMORPHIC OBJECTS

Adding at least one virtual function to a class alters the storage layout for all objects of that class type.

http://www.drdobbs.com/240012098

DATA STRUCTURE AUDITS

If you can divide your data structures into a data part and a structure part, and you can write an audit program that is capable of rebuilding the structure from the data, you can use the auditor not only to make your programs more reliable, but to get them working more quickly than you might be able to do otherwise.

http://www.drdobbs.com/240142217

IN THIS ISSUE

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Sean Coady Webmaster, Dr. Dobb's
sean.coady@ubm.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

DR DOBB'S
UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000