Objective-C Advancements In Depth

Session 322

Blaine Garst
Wizard of Runtimes

Roadmap

- New Objective-C Language Extensions
  - Extensions that work everywhere
  - Automatic Reference Counting (ARC) Extensions
- ARC Internals
  - ARC implementation specifics
  - Coding for ARC interoperability
  - Performance

Available in Apple
LLVM Compiler 3 in Xcode 4.2

@autoreleasepool

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
for (id a in collection) {
    [array addObject:[NSString stringWithFormat:..., a]];
}
[pool drain];
```

- Ever try `[pool retain]`? Don't, it raises!
- Use direct language construct instead:

```
@autoreleasepool {
   for (id a in collection) {
      [array addObject:[NSString stringWithFormat:..., a]];
   }
}
```

Instance Variable Declarations

```
@interface YourClass : NSObject {
    id ivar1;
}
```

Instance variables are declared
in the @interface class declaraction

Instance Variable Declarations


```
@interface YourClass : NSObject {
    id ivar1;
}
```

Instance variables are declared
in the @interface class declaraction

```
@interface YourClass () {
    id ivar2;
}
```

or in a class extension

```
@implementation YourClass {
    id ivar3;
}
```

or in your @implementation file!

Instance Variable Visibility Problem

```
// PUBLIC Elegant.h

#import <system/needed_for_Messy.h>
#import <Messy/MessyThing1.h>

@interface Elegant : NSObject {

   MessyDetail detail1;
   SecretStuff detail2;
   ...;
}
- (void)elegantMethod;
@end

// PRIVATE Elegant.m

@implementation Elegant

- (void)elegantMethod { ... }
@end
```

Messy public header

Single file implementation

Instance Variable Visibility Problem

```
// PUBLIC Elegant.h
@interface Elegant : NSObject

- (void)elegantMethod;
@end
```

Clean public header

```
// PRIVATE Elegant.m
#import <system/needed_for_Messy.h>
#import <Messy/MessyThing1.h>

@implementation Elegant {

   MessyDetail detail1;
   SecretStuff detail2;
   ...;
}
- (void)elegantMethod { ... }
@end
```

Details where important

More Than One Implementation File?

```
// PUBLIC Elegant.h

#import <system/needed_for_Messy.h>
#import <Messy/MessyThing1.h>

@interface Elegant : NSObject {
   MessyDetail detail1;
   SecretStuff detail2;
   ...;
}
- (void)elegantMethod;
@end


// PRIVATE Elegant.m

@implementation Elegant
- (void)elegantMethod { ... }
@end
```

Messy public header

Multiple implementation files

More Than One Implementation File?

```
// PUBLIC Elegant.h

@interface Elegant : NSObject
- (void)elegantMethod;
@end
```

Same clean header

```
// PRIVATE Internal.h
@interface Elegant () {
   MessyDetail detail1;
   SecretStuff detail2;
   ...;
}
```

Details moved to class extension

```
// PRIVATE Elegant.m

#import "Internal.h"

@implementation Elegant
- (void)elegantMethod { ... }
@end
```

Multiple implementation files

Deploying to Earlier Releases

Deploying to Earlier Releases

Deploying to Earlier Releases

Deploying to Earlier Releases

New

```
NS_CLASS_AVAILABLE(10_7, 5_0)
@class NSJSONSerialization
```

Assuring Class Availability

```
Class json = NSClassFromString(@"NSJSONSerialization");
if (json) {
   if ([json isValidJSONObject:myObject]) {
       ...
   }
}
```

Existing practice

```
if ([NSJSONSerialization self]) {
   if ([NSJSONSerialization isValidJSONObject:myObject]) {
       ...
   }
}
```

Weak linking for Classes!

- Weakly linked class references yield nil when not available
- Subclassing is supported!
- Deploys down to iOS 3.1 and Mac OS X 10.6.8

Stronger Type Checking

```
// buggy.m
#import <Foundation/Foundation.h>
void foo() {
   NSMutableArray *array = [[NSMutableSet alloc] init];
    [array addObject:[NSObject new]];
    [array objectAtIndex:0];  // throws exception!
}

$ clang -c buggy.m
buggy.m:4:20: warning: incompatible pointer types initializing
'NSMutableArray *' with an expression of type
      'NSMutableSet *'
   NSMutableArray *array = [[NSMutableSet alloc] init];
                    ^       ~~~~~~~~~~~~~~~~~~~~~~~~~~
/System/.../Foundation.framework/Headers/NSObject.h:72:1 note:
    instance method 'init' is assumed to return an instance of
its receiver type ('NSMutableSet *')
- init;
^
1 warning generated.
```

Runtime Error

Is Now a Compiler Warning!

Advanced ARC

ARC Summary

- Automatic Reference Counting (i.e. retain, release, autorelease)
- Automates Objective-C Objects only
  - Does not automate malloc() and free()
  - Does not automate CoreFoundation (CF) or CoreGraphics (CG) etc.
- Interoperates with existing manually coded retain/release classes
- How does the compiler get it right?
  - It has been taught Cocoa conventions
  - It relies on new object pointer ownership qualifiers

Cocoa Conventions

- Objective-C has not specified allocation, initialization, or destruction
- Cocoa, however, does:
  - Class methods beginning with new and alloc create objects
  - Instance methods beginning with init initialize objects
  - Instance methods beginning with copy or mutableCopy make copies
  - Use retain, release, and autorelease to manage reference counts
  - dealloc destroys objects after last release
- Under ARC, these rules are now part of the language!

Cocoa Convention for Ownership Transfer

- Methods beginning with alloc, init, new, copy yield +1 retained items
- Works perfectly for methods that follow this convention

```
// singleton or immutable value class
- copy { return self; }  // ARC: return objc_retain(self);
```

- What about names that don't follow convention?

```
// Unconventional.h
- (NSString *)copyRightNotice;
- (License *)copyLeftLicense;
- (Simulation *)copyMachine;
+ (id)makeNewThingie;
+ (id)createMagicInstrument;
```

Unconventional Naming Example

```
// License.m
- (License *) copyLeftLicense {
     ...
     return license;
}

// License Client

- someMethod {
   id l = [version2 copyLeftLicense];
   ...
}
```

Unconventional Naming Example

- Under ARC, retain and release balance, and it works!

```
// License.m
- (License *) copyLeftLicense {
      ...
      return objc_retain(license);
}
```

ARC compiled

```
// License Client

- someMethod {
    id l = [version2 copyLeftLicense];
    objc_release(l);
}
```

ARC compiled

If everything is compiled under ARC, it all just works!

Unconventional Naming Example

- Unbalanced objc_release causes major trouble!

```
// License.m
- (License *) copyLeftLicense {
     ...
     return license;
}
```

Non-ARC compiled

```
// License Client

- someMethod {
   id l = [version2 copyLeftLicense];
   ...
   objc_release(l);
}
```

ARC compiled

Unconventional Naming Example

- Unbalanced objc_retain leaks!

```
// License.m
- (License *) copyLeftLicense {
      ...
      return objc_retain(license);
}
```

ARC compiled

```
// License Client

- someMethod {
    id l = [version2 copyLeftLicense];
    ...
}
```

Non-ARC compiled

Unconventional Naming Remedy #1

- Rename the methods to conform to Cocoa Convention!

```
// Unconventional.h
- (NSString *)copyRightNotice;
- (License *)copyLeftLicense;
- (Simulation *)copyMachine;
+ (id)makeNewThingie;
+ (id)createMagicInstrument;
```

Compiler does name matching based on "CamelCase"

```
// Conventional.h
- (NSString *)copyrightNotice;
- (License *)copyleftLicense;
- (Simulation *)copymachine;
+ (id)newThingie;
+ (id)newMagicInstrument;
```

Remove the humps!

Use the "new" family

Unconventional Naming Remedy #2

- Use ownership transfer annotations

```
// Annotated Unconventional.h
- (NSString *)copyRightNotice  NS_RETURNS_NOT_RETAINED;
- (License *)copyLeftLicense   NS_RETURNS_NOT_RETAINED;
- (Simulation *)copyMachine    NS_RETURNS_NOT_RETAINED;
+ (id)makeNewThingie           NS_RETURNS_RETAINED;
+ (id)createMagicInstrument    NS_RETURNS_RETAINED;
```

Ownership Type Qualifiers

Object Pointers Are Ownership Qualified

- Four ownership type qualifiers

```
__strong
__weak
__unsafe_unretained
__autoreleasing
```

__strong Variables "retain" Their Values

- __strong is the default, you almost never have to type it
  - Stack local variables, including parameters, are __strong
    - They never hold dangling references!
- Values released sometime after last use

```
// Stack.m
- pop {
    id __strong result = [array lastObject];
    [array removeLastObject];
    return result;
}
```

__weak Variables Don't Retain Values

- They are great for breaking reference cycles!
- Safely yield nil as soon as referenced object starts deallocating
- Stack local __weak variables just work!

```
void testWeak() {
    id newObject = [NSObject new];
    __weak id weakValue = newObject;
    newObject = nil;
    assert(weakValue == nil);
}
```

__weak Variables Don't Retain Values

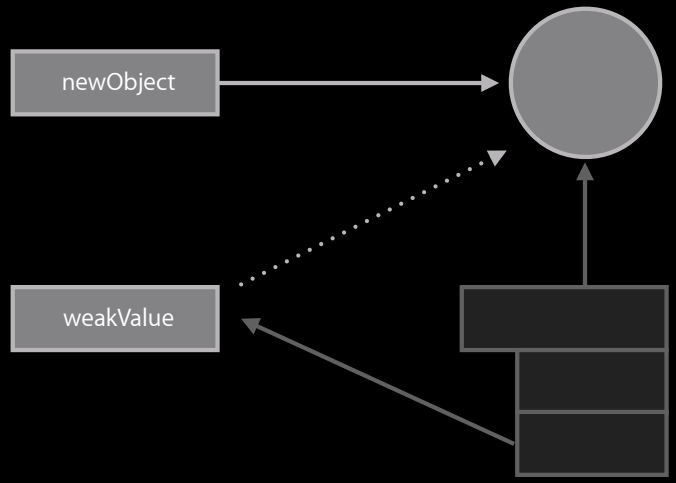
```
void testWeak() {
    id newObject = [NSObject new];
    __weak id weakValue = newObject;
    newObject = nil;
    assert(weakValue == nil);
}
```

__weak Variables Don't Retain Values

```
void testWeak() {
    id newObject = [NSObject new];
    __weak id weakValue = newObject;
    newObject = nil;
    assert(weakValue == nil);
}
```

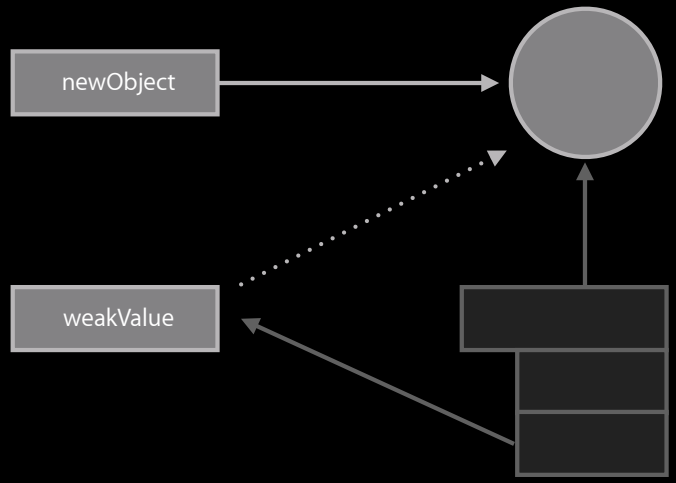
__weak Variables Don't Retain Values

```
void testWeak() {
    id newObject = [NSObject new];
    __weak id weakValue = newObject;
    newObject = nil;
    assert(weakValue == nil);
}
```



__weak Variables Don't Retain Values

```
void testWeak() {
    id newObject = [NSObject new];
    __weak id weakValue = newObject;
    newObject = nil;
    assert(weakValue == nil);
}
```



__weak Variables Really Don't Retain Values!

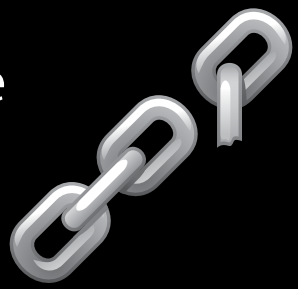
```
void testWeak2() {
   __weak id weakValue = [NSObject new];
   assert(weakValue == nil);
}
```

__weak Variables Really Don't Retain Values!

```
void testWeak2() {
    __weak id weakValue;
    id tmp = [NSObject new];
    objc_storeWeak(&weakValue, tmp);
    objc_release(tmp);
    assert(weakValue == nil);
}
```

__weak System Caveats

- __weak system only available on iOS 5 and Mac OS X 10.7
- Requires modifications to custom retain/release implementations
  - At Apple, many classes deleted their custom retain/release code
  - So should you!
- Some Apple provided classes don't participate
  - NSWindow, NSViewController, a few others
  - Hard crash if you attempt to form a weak reference to these
- 3rd party libraries may also need modifications
- Must use __unsafe_unretained as the alternative



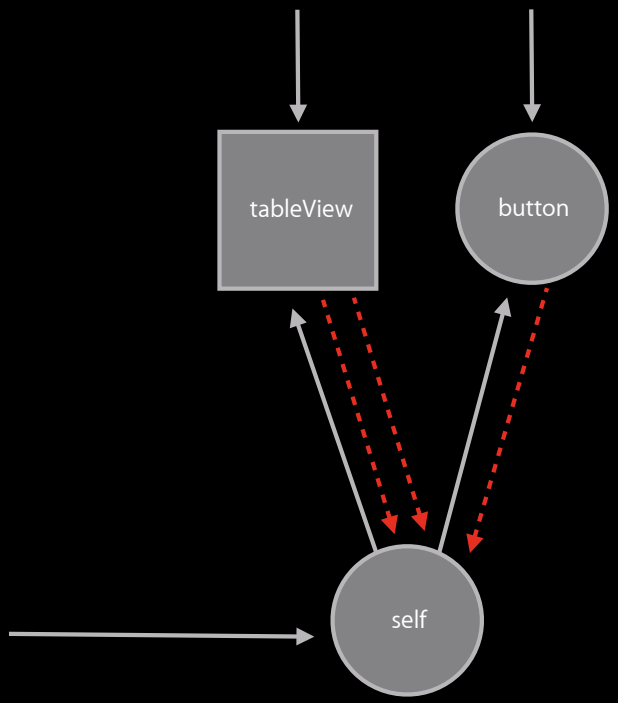
`__unsafe_unretained` Qualifier

- Familiar unretained object concept, now with a formal name
  - This is what `@property(assign)` variables are
  - Most delegates work this way
- Used to avoid cycles among cooperating objects
  - dealloc method *must* clear unretained references held elsewhere
- Can be used in structures and unions

```
typedef struct {
   __unsafe_unretained NSString *name;
   float x, y, z;
} Named_point_t;
```

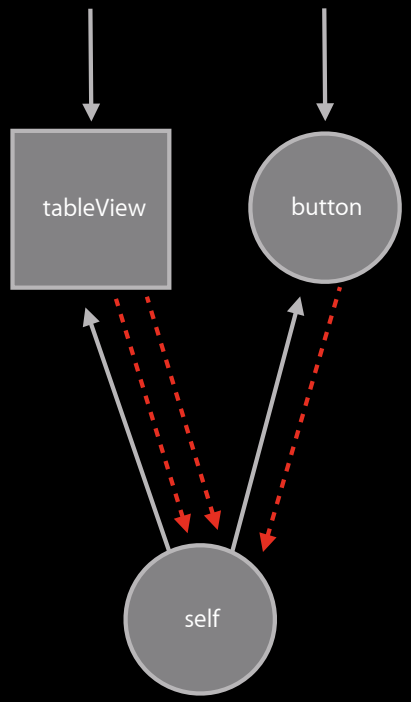
__unsafe_unretained Dealloc Dance

```
@implementation MyCustomDelegateController {
    NSTableView *tableView;
    NSButton *doItButton;
}
- (void)dealloc {
    [tableView setDelegate:nil];
    [tableView setDataSource:nil];
    [doItButton setTarget:nil];
}
@end
```



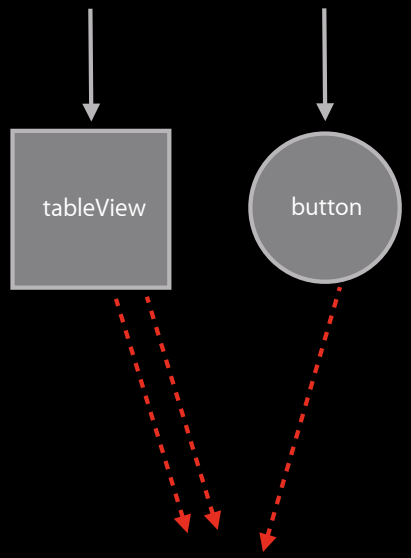
__unsafe_unretained Dealloc Dance

```
@implementation MyCustomDelegateController {
    NSTableView *tableView;
    NSButton *doItButton;
}
- (void)dealloc {
    [tableView setDelegate:nil];
    [tableView setDataSource:nil];
    [doItButton setTarget:nil];
}
@end
```



__unsafe_unretained Dealloc Dance

```
@implementation MyCustomDelegateController {
    NSTableView *tableView;
    NSButton *doItButton;
}
- (void)dealloc {



}
@end
```



`__autoreleasing` for Indirect Pointers

- Cocoa Convention does not transfer ownership via parameters
- Indirect pointers are treated like autoreleased return values
- Prior values are ignored when storing new autoreleased values

This:
```
- (void)kitMethod:param error:(NSError **)err;
```

Really means:
```
- (void)kitMethod:param
            error:(__autoreleasing NSError **)err;
```

- Passing ownership is possible by using __strong

```
- doSomethingAndCreateObject:(__strong id **)resultPtr;
```

@property Support for Ownership

```
@interface YourObject : NSObject
@property(strong)              id x;  // __strong, a.k.a. retain
@property(weak)                id y;  // __weak
@property(unsafe_unretained) id z;  // __unsafe_unretained, a.k.a. assign
@end
```

Cocoa Copy Convention for ARC Code

```
- (id)copyWithZone:(NSZone *)z {
    MyObject *result = [super copyWithZone:NULL];

    result->ivar1 = ivar1;
    result->ivar2 = [ivar2 mutableCopy];

    return result;
}
```

Works with NSCell too!

Cocoa Convention for Exceptions

- Exceptions indicate unrecoverable error (programmer mistake)
  - Use NSError ** convention instead for recoverable situations
- Objects are often leaked when exceptions are thrown

```
- someMethod {
    id result = [ivar copy];

    [result tickle];
    [result paintMyHouse];  // RAISES!

    return result;
}
```

- `__weak` stack and `__weak __block` variables, however, are unregistered

Blocks

Blocks in ARC

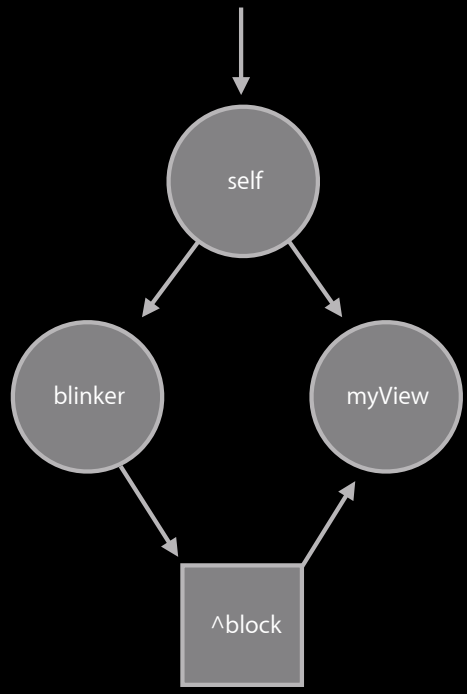
- Blocks start out on the stack, must be copied when:
  - Stored into instance variables or globals
  - Returned or indirectly assigned
- ARC automates this so you don't have to think about this!

```
return [[^{...} copy] autorelease];
ivar = [^{...} copy];
global = [^{...} copy];
```

Blocks work best under ARC!

Reference Cycle Via Captured Self

```
- (void)startBlinker {
    blinker = [BlinkerService register: ^{
        [myView blink];
    }];
}

- (void)stopBlinker {
    [blinker cancel];
    blinker = nil;
}

- (void)dealloc {
    [self stopBlinker];
}
```

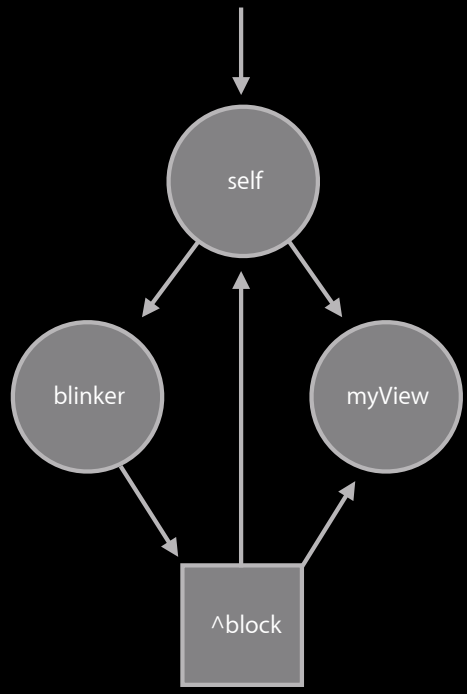


Intention…

Reference Cycle Via Captured Self

```
- (void)startBlinker {
    blinker = [BlinkerService register: ^{
        [self.myView blink];
    }];
}

- (void)stopBlinker {
    [blinker cancel];
    blinker = nil;
}

- (void)dealloc {
    [self stopBlinker];
}
```

Leaks only when blinking!

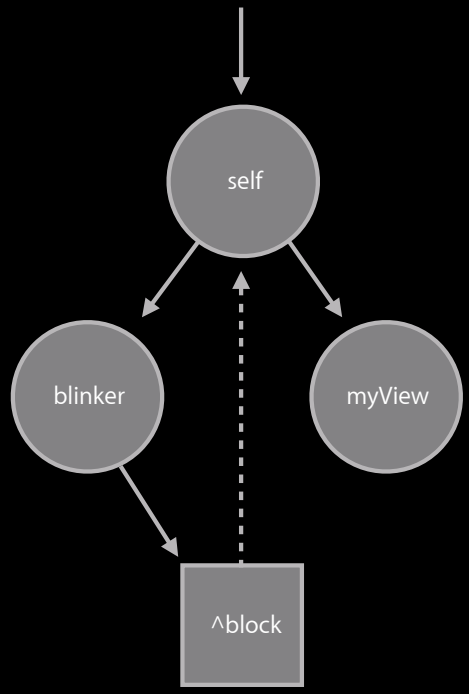


Reality

Reference Cycle Via Captured Self

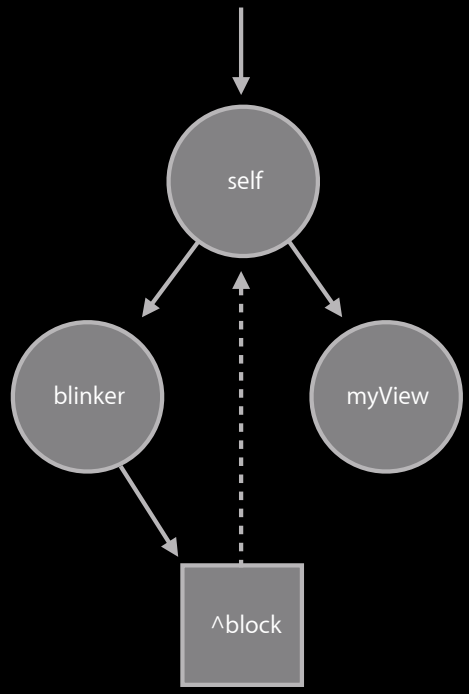
```
- (void)startBlinker {
    __weak MyClass *weakSelf = self;

    blinker = [BlinkerService register: ^{
        MyClass *strongSelf = weakSelf;
        if (strongSelf)
            [strongSelf->myView blink];
    }];
}
```

weakSelf->myView could crash!



Reference Cycle Via Captured Self

```
- (void)startBlinker {
    __weak MyClass *weakSelf = self;

    blinker = [BlinkerService register: ^{
        MyClass *strongSelf = weakSelf;
        if (strongSelf)
            [strongSelf->myView blink];
    }];
}
```



__block Behavior Change Under ARC

- __block object variables are __strong by default under ARC
- Under non-ARC, the default is essentially __unsafe_unretained
- As such, they have been used to break cycles:

```
__block id unretainedSelf = self;
```

- With behavior change, this use will likely cause a cycle under ARC!

```
__block id unretainedSelf = self; // really retainedSelf!!
```

- You should convert to use __weak if you can

```
__weak id weakSelf = self;
```

- If you can't, reintroduce the unretained behavior

```
__block __unsafe_unretained id unsafeSelf = self;
```

Objective-C++ ARC

Everything Works!

```
class Example {
   id                       x;  // non-POD
   __weak id                y;  // non-POD
   __unsafe_unretained id z;  // POD
public:
```

- Complete support for templates
- C++ Standard Library supports Objective-C objects in ARC
  - Containers require explicit ownership qualifiers

```
std::vector<__weak id> listeners;
std::map<std::string, __strong NSView *> activeViews;
```

ARC Internals

Greg Parker
Runtime Wrangler

ARC from the Inside

- Compiler adds retain and release calls
- Optimizer removes some of them

Strong Variables

```
-(id)swapWithValue:(id)newValue {
    id oldValue = self->value;
    self->value = newValue;
    return oldValue;
}
```

Strong Variables

```
-(id)swapWithValue:(id)newValue {
    [newValue retain];
    id oldValue = nil;
    oldValue = [self->value retain];
    [self->value release];
    self->value = [newValue retain];
    [newValue release];
    return [oldValue autorelease];
}
```

Strong Variables

```
-(id)swapWithValue:(id)newValue {
    objc_retain(newValue);
    id oldValue = nil;
    oldValue = objc_retain(self->value);
    objc_release(self->value);
    self->value = objc_retain(newValue);
    objc_release(newValue);
    return objc_autorelease(oldValue);
}
```

Strong Variables

```
-(id)swapWithValue:(id)newValue {
    objc_retain(newValue);
    id oldValue = nil;
    oldValue = objc_retain(self->value);
    objc_release(self->value);
    self->value = objc_retain(newValue);
    objc_release(newValue);
    return objc_autorelease(oldValue);
}
```

Strong Variables

```
-(id)swapWithValue:(id)newValue {
    objc_retain(newValue);
    id oldValue = nil;
    oldValue = objc_retain(self->value);
    objc_release(self->value);
    self->value = objc_retain(newValue);
    objc_release(newValue);
    return objc_autorelease(oldValue);
}
```

Strong Variables

```
-(id)swapWithValue:(id)newValue {
    objc_retain(newValue);
    id oldValue;
    oldValue = self->value;
    self->value = newValue;
    return objc_autorelease(oldValue);
}
```

Weak Variables

```
__weak id delegate;

-(void)setDelegate:(id)d {
    self->delegate = d;
}



-(id)delegate {
    return self->delegate;
}
```

Weak Variables

```
__weak id delegate;

-(void)setDelegate:(id)d {
    d = objc_retain(d);
    objc_storeWeak(&self->delegate, d);
    objc_release(d);
}

-(id)delegate {
    id temp = objc_loadWeak(&self->delegate);
    return objc_autorelease(objc_retain(temp));
}
```

Weak Variables

```
__weak id delegate;

-(void)setDelegate:(id)d {
    d = objc_retain(d);
    objc_storeWeak(&self->delegate, d);
    objc_release(d);
}

-(id)delegate {
    id temp = objc_loadWeak(&self->delegate);
    return objc_autorelease(objc_retain(temp));
}
```

Weak Variables

```
__weak id delegate;

-(void)setDelegate:(id)d {
    objc_storeWeak(&self->delegate, d);
}



-(id)delegate {
    return objc_loadWeak(&self->delegate);
}
```

NSError

```
-(BOOL) rescueKittens:(NSError **)error;

-(void) performRescue:(id)sender {
    NSError *err = nil;
    BOOL ok = [self rescueKittens:&err];
    if (!ok) NSLog(@"OH NOES %@", err);
}
```

NSError

```
-(BOOL) rescueKittens:(__autoreleasing NSError **)error;

-(void) performRescue:(id)sender {
    __strong NSError *err = nil;
    BOOL ok = [self rescueKittens:&err];
    if (!ok) NSLog(@"OH NOES %@", err);
    objc_release(err);
}
```

NSError

```
-(BOOL) rescueKittens:(__autoreleasing NSError **)error;

-(void) performRescue:(id)sender {
    __strong NSError *err = nil;
    __autoreleasing NSError *temp = nil;
    BOOL ok = [self rescueKittens:&temp];
    err = objc_retain(temp);
    if (!ok) NSLog(@"OH NOES %@", err);
    objc_release(err);
}
```

NSError

```
-(BOOL) rescueKittens:(NSError **)error;

-(void) performRescue:(id)sender {
    NSError *err = nil;
    BOOL ok = [self rescueKittens:&err];
    if (!ok) NSLog(@"OH NOES %@", err);
}
```

NSError

```
-(BOOL) rescueKittens:(NSError **)error;

-(void) performRescue:(id)sender {
    __autoreleasing NSError *err = nil;
    BOOL ok = [self rescueKittens:&err];
    if (!ok) NSLog(@"OH NOES %@", err);
}
```

Calls Added by the Compiler

- Basic memory management

  `objc_retain`

  `objc_release`

  `objc_autorelease`

- Weak reference system

  `objc_loadWeak`

  `objc_storeWeak`

Calls Added by the Compiler

- `NSObject` implementation

  `_objc_rootAlloc`

  `_objc_rootRetain`

- Autorelease pool implementation

  `objc_autoreleasePoolPush`

  `objc_autoreleasePoolPop`

Calls Added by the Compiler

- Autorelease optimization

  `objc_autoreleaseReturnValue`

  `objc_retainAutoreleasedReturnValue`

- Other optimizations

  `objc_storeStrong`

  `objc_destroyWeak`

Calls Added by the Compiler

- For informational purposes only
- Use declared API in public headers only

Death of an Object

Death of an Object

- ARC releases ivars and properties automatically
- ARC erases `__weak` references automatically

Deallocation Timeline

-release to zero

Subclass -dealloc

NSObject -dealloc

object_dispose()

free()

- Call destructors for C++ ivars
- Call `-release` for ARC ivars
- Release weak references
- Call `objc_destructInstance()`
- Call `[self dealloc]`
- Erase `__weak` references
- Call `free()`

ARC Adoption

Low-level ARC Adoption

- Edit Core Foundation usage
- Edit or remove custom retain/release implementations
- Edit or remove custom weak reference systems

ARC and Core Foundation

- ARC automates Objective-C objects and methods
- ARC does not understand Core Foundation code
- You must help ARC understand

CF versus ARC

```
NSString* name =
    (NSString *)ABRecordCopyCompositeName(...);
self.nameView.text = name;
```

CF versus ARC

```
NSString* name =
    objc_retain((NSString *)ABRecordCopyCompositeName(...));
self.nameView.text = name;
objc_release(name);
```

CF versus ARC

```
NSString* name =
    objc_retain((NSString *)ABRecordCopyCompositeName(...));
self.nameView.text = name;
objc_release(name);
```

CF versus ARC

```
NSString* name =
    objc_retain((NSString *)ABRecordCopyCompositeName(...));
self.nameView.text = name;
objc_release(name);

// `name` leaks
```

CF versus ARC

- Unmodified CF code with ARC may leak, or crash, or run correctly
- ARC disallows most casts between CF and Objective-C types
  - "Disallow" means "compile error"
- Use new functions and annotated casts instead

CF Recipes for ARC

- CF value was returned by a method; no CF memory management
- CF value came from somewhere else; no CF memory management
- CF value has CF memory management in your code

CF Values Returned by Methods

```
@interface UIImage
    -(CGImageRef) CGImage;
@end

id myCGImage = (id)[myUIImage CGImage];
[array addObject:myCGImage];
```

- Unchanged in ARC
- ARC uses Cocoa naming conventions

CF Recipes for ARC

- CF value was returned by a method; no CF memory management
  - Use a simple cast
- CF value came from somewhere else; no CF memory management
- CF value has CF memory management in your code

CF Values from Other Sources

```
CFStringRef str = (CFStringRef)[array objectAtIndex:...];
CFShow(str);

NSString *str = (NSString *)CFArrayGetValueAtIndex(...);
NSLog(@"%@", str);
```

CF Values from Other Sources

```
CFStringRef str = (__bridged CFStringRef)[array objectAtIndex:...];
CFShow(str);

NSString *str = (__bridged NSString *)CFArrayGetValueAtIndex(...);
NSLog(@"%@", str);
```

- No CF memory management involved
- ARC may retain and release the value

CF Recipes for ARC

- CF value was returned by a method; no CF memory management
  - Use a simple cast
- CF value came from somewhere else; no CF memory management
  - Use a `__bridged` cast
- CF value has CF memory management in your code

Handling CF Create, Copy, Retain

```
-(NSString *)firstName {
    NSString *result =
        (NSString *)ABRecordCopyCompositeName(...);
    return [result autorelease];
}
```

Handling CF Create, Copy, Retain

```
-(NSString *)firstName {
    NSString *result =
        (NSString *)ABRecordCopyCompositeName(...);
    return [result autorelease];
}
```

Handling CF Create, Copy, Retain

```
-(NSString *)firstName {
    NSString *result =
        (NSString *)ABRecordCopyCompositeName(...);
    return result;
}
```

Handling CF Create, Copy, Retain

```
-(NSString *)firstName {
    NSString *result =
        CFBridgingRelease(ABRecordCopyCompositeName(...));
    return result;
}
```

- Balances a previous CF Create or Copy or Retain
- Safely transfers the value to ARC

Handling CFRelease

```
CFStringRef str = (CFStringRef)[myNSString copy];
[...]
CFRelease(str);
```

Handling CFRelease

```
CFStringRef str = CFBridgingRetain([myNSString copy]);
[...]
CFRelease(str);
```

- Balanced by a subsequent CFRelease
- Safely accepts the value from ARC

CF Recipes for ARC

- CF value was returned by a method; no CF memory management
  - Use a simple cast
- CF value came from somewhere else; no CF memory management
  - Use a `__bridged` cast
- CF value has CF memory management in your code
  - Balance with `CFBridgingRetain` or `CFBridgingRelease`

Under Construction

- Your WWDC seed does not implement any of the above
- See the *Programming With ARC Release Notes* for details

Custom Retain and Release

- `-release` must coordinate with weak reference system
- Custom retain and release implementations must evolve or die

Custom Retain and Release

- Two new methods

```
-(BOOL)allowsWeakReference
-(BOOL)retainWeakReference
```

- Changes to existing methods
- Subtle race conditions
- No simple recipe
- Not recommended

Custom Retain and Release

- Recommendation: delete your custom retain/release implementation
- NSObject's retain and release is improved
- Use `__weak` variables instead of custom teardown logic
- Use simpler singleton patterns

Custom Retain and Release

- Recommendation: disallow weak references to your class

```
-(BOOL) allowsWeakReference {
    return NO;
}
```

Custom Weak Reference Systems

- Replace custom system with ARC `__weak` variables, or
- Rewrite custom system with the runtime's API

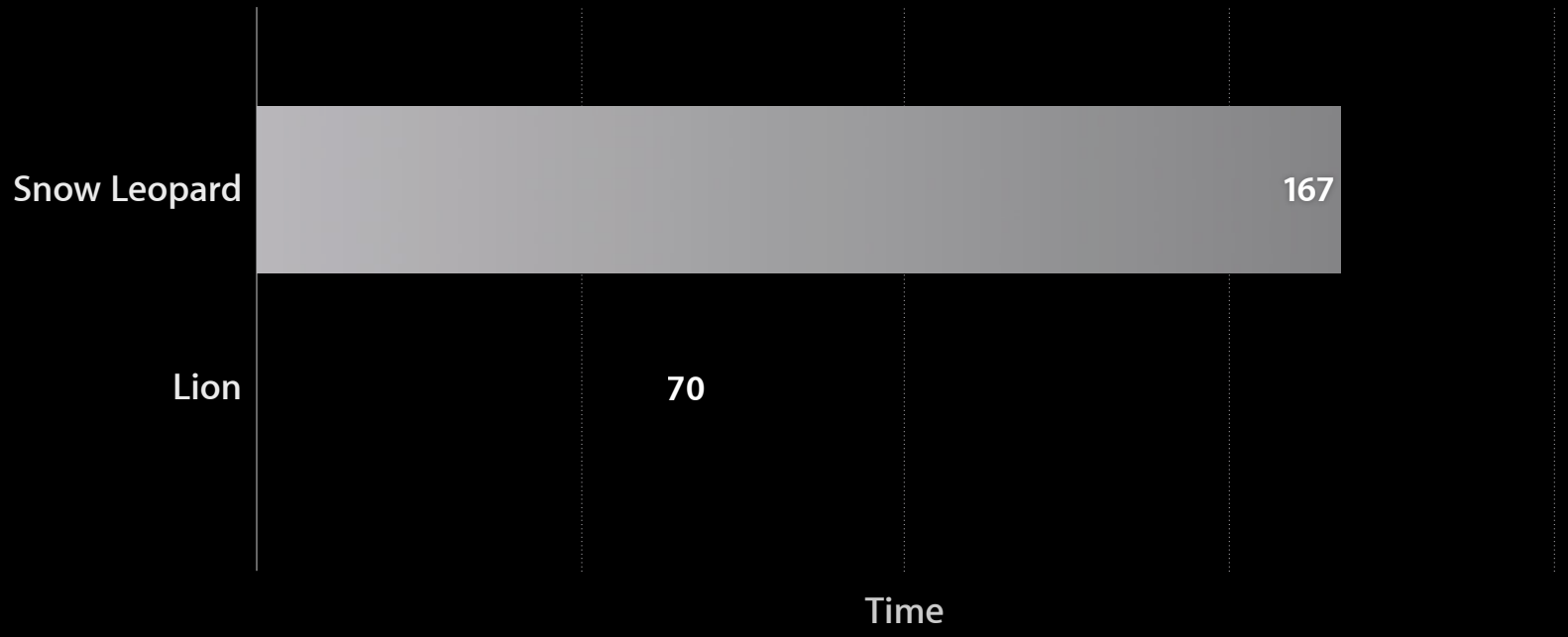
```
id objc_storeWeak(id *location, id value)
id objc_loadWeak(id *location)
```

objc_loadWeak and objc_storeWeak

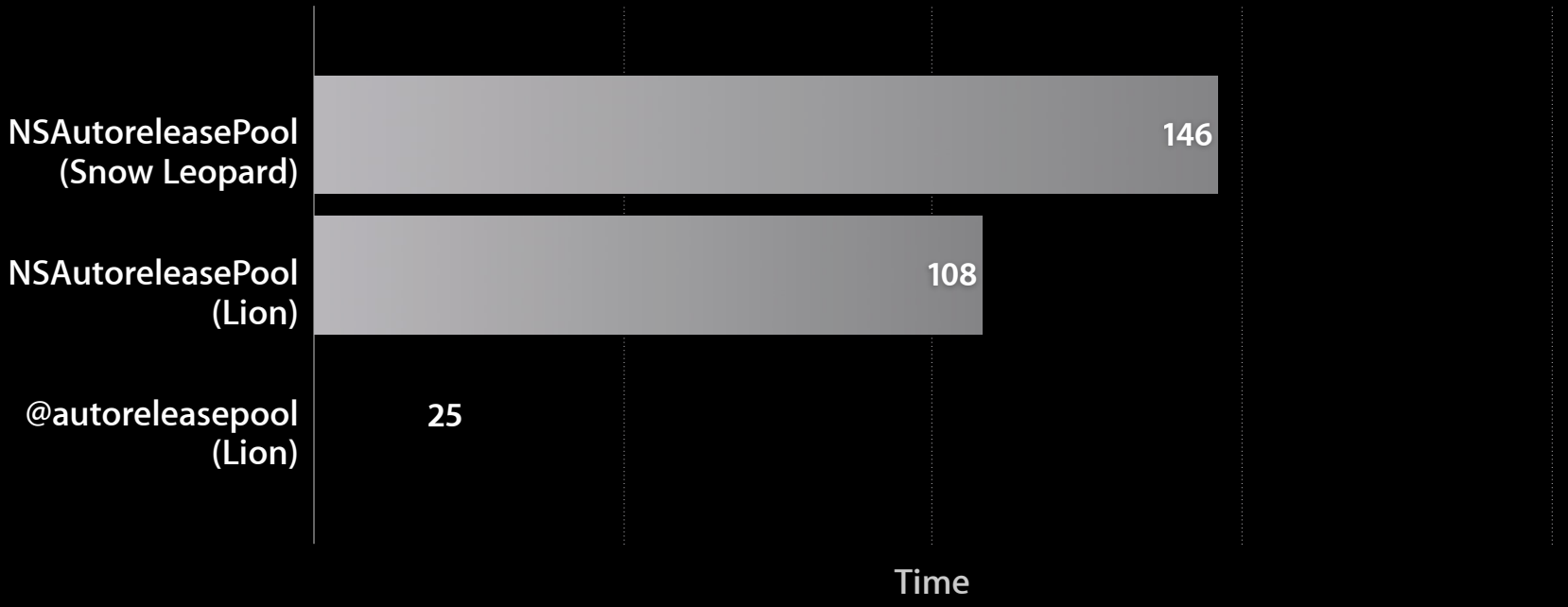
```
@implementation MyClass
-(id)value {
    return objc_loadWeak(&myWeakIvar);
}

-(void)setValue:(id)newValue {
    objc_storeWeak(&myWeakIvar, newValue);
}

-(void)dealloc {
    objc_storeWeak(&myWeakIvar, nil);
}
@end
```

Performance

Retain and Release



Autorelease Pool



Autoreleased Return Values

```
-(id)value {
    return [[self->value retain] autorelease];
}
```

```
-(void)takeValueFrom:(id)other {
    self->value = [[other value] retain];
}
```

Autoreleased Return Values

```
-(id)value {
    return objc_autoreleaseReturnValue([self->value retain]);
}
```

Save value in thread-local storage
Skip autorelease and retain

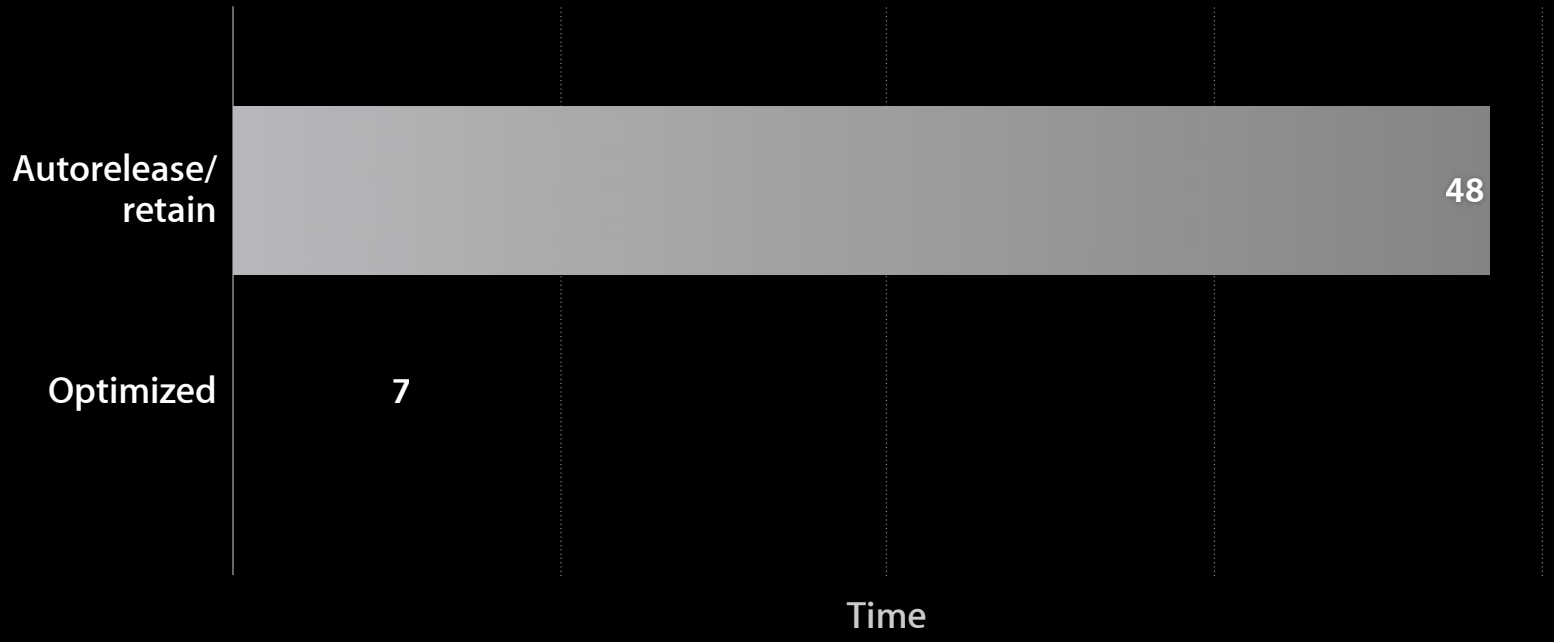
```
-(void)takeValueFrom:(id)other {
    self->value = objc_retainAutoreleasedReturnValue([other value]);
}
```

Autoreleased Return Values

```
-(id)value {
    return objc_autoreleaseReturnValue([self->value retain]);
}

-(void)takeValueFrom:(id)other {
    self->value = [[other value] retain];
}
```

No optimization
Calls autorelease and retain as usual

Autoreleased Return Values



Summary

- ARC: Automated Reference Counting
- Ownership qualifiers describe your objects' relationships
- You must modify some low-level memory management
- Performance improvements balance some of the costs

More Information

Michael Jurewitz
Developer Tools Evangelist
jurewitz@apple.com

Documentation
Programming With ARC Release Notes
http://developer.apple.com/

Apple Developer Forums
http://devforums.apple.com

Related Sessions

Introducing Automatic Reference Counting	Presidio Tuesday 4:30PM
Blocks and Grand Central Dispatch in Practice	Pacific Heights Wednesday 10:15AM
Moving to Apple LLVM Compiler	Nob Hill Wednesday 10:15AM

Q&A


WWDC2011