Adopting Automatic Reference Counting

Session 406 / 416

Malcolm Crawford
Technical Writer

Talk Roadmap

- Manual memory management
- What is ARC?
- How does it work?
- How do you switch?

Manual Memory Management

What's the Problem?

- Memory management problems lead to crashing
- Crashing leads to unhappiness
- Unhappiness leads to rejection

Memory Management Problems:

Memory Management Problems:

Reason for apps crashing

Memory Management Problems:

#1

Reason for apps crashing

#1

Cause of app rejection

What were the problems with traditional Objective-C memory management?

Programming with Retain and Release

Straightforward rules, but…

Programming with Retain and Release

Straightforward rules, but…

- Lots of details

Programming with Retain and Release

Straightforward rules, but…

- Lots of details
  - Naming conventions

Programming with Retain and Release

Straightforward rules, but…

- Lots of details
  - Naming conventions
  - Autorelease pools

Programming with Retain and Release

Straightforward rules, but…

- Lots of details
  - Naming conventions
  - Autorelease pools
  - Block_copy

Programming with Retain and Release

Straightforward rules, but…

- Lots of details
  - Naming conventions
  - Autorelease pools
  - Block_copy
  - Autoreleasing or not?

Programming with Retain and Release

Straightforward rules, but…

- Lots of details
  - Naming conventions
  - Autorelease pools
  - Block_copy
  - Autoreleasing or not?
    - [NSString stringWith…

Programming with Retain and Release

Straightforward rules, but…

- Lots of details
  - Naming conventions
  - Autorelease pools
  - Block_copy
  - Autoreleasing or not?
    - [NSString stringWith…
    - [[NSString alloc] initWith…]

Programming with Retain and Release

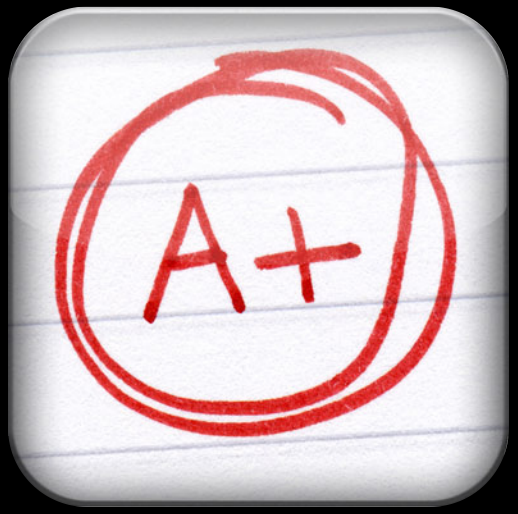
Straightforward rules, but…

- Lots of details
  - Naming conventions
  - Autorelease pools
  - Block_copy
  - Autoreleasing or not?
    - [NSString stringWith…
    - [[NSString alloc] initWith…]

Cognitive overload

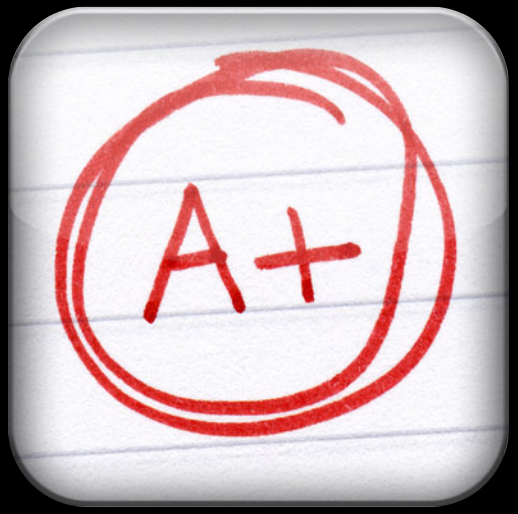
Perfection Required

- Never forget the rules



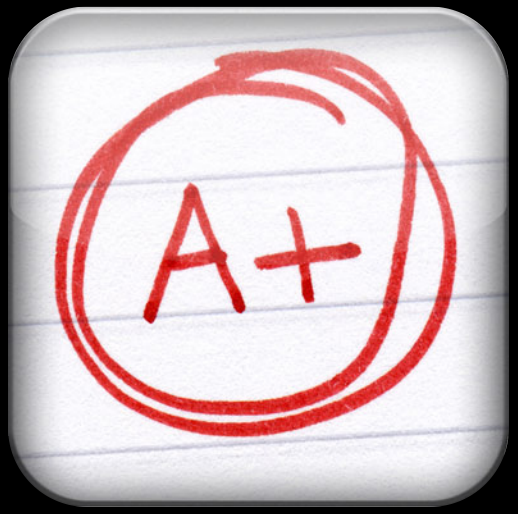
Perfection Required

- Never forget the rules
- Never forget a release on an error case



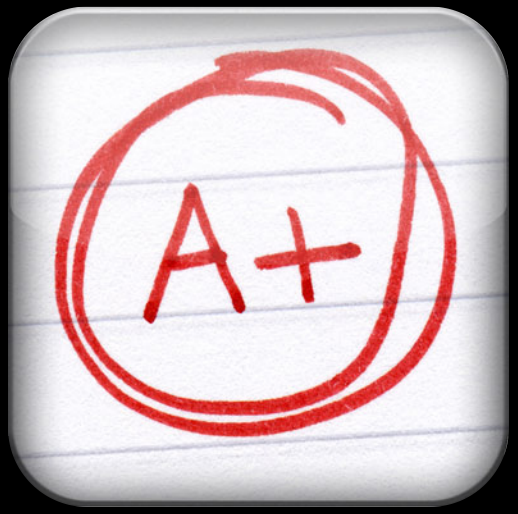
Perfection Required

- Never forget the rules
- Never forget a release on an error case
- Never dereference a dangling pointer



Perfection Required

- Never forget the rules
- Never forget a release on an error case
- Never dereference a dangling pointer



Not what people are best at

Adhering to Rules Is Hard

Even with help from tools

- Instruments
  - Allocations, Leaks, Zombies
- Static Analyzer
- Heap
- ObjectAlloc
- vmmap
- MallocScribble
- Debugger watchpoints
- …and lots more

Xcode Static Analyzer

Tells you when you did not follow the rules

```
NSObject *objectID = 0;

for (NSUInteger i=0; i < count; ++i) {          Looping back to the head of the loop

  NSObject *object = [trackedElements objectAtIndex:i];

  if ([object isMemberOfClass:[NSString class]])
  {
    objectID = [[NSString alloc] initWithString:aString];
                Method returns an Objective-C object with a +1 retain count (owning reference)
  }

  if (objectID != nil)
  {
    [objectID release];                         Object released
                                                Reference-counted object is used after it is released
  }
}
```

Lessons Learned from the Analyzer

Lessons Learned from the Analyzer

- Dozens or hundreds of leaks are common
  - Developers think the analyzer is buggy

Lessons Learned from the Analyzer

- Dozens or hundreds of leaks are common
  - Developers think the analyzer is buggy
- Common sources of false positives:
  - Core Foundation APIs
  - Unions, structs, etc.

Lessons Learned from the Analyzer

- Dozens or hundreds of leaks are common
  - Developers think the analyzer is buggy
- Common sources of false positives:
  - Core Foundation APIs
  - Unions, structs, etc.
- Cocoa code has strong patterns
  - Naming conventions

Xcode Static Analyzer

Tells you when you did not follow the rules

```
NSObject *objectID = 0;

for (NSUInteger i=0; i < count; ++i) {          Looping back to the head of the loop

  NSObject *object = [trackedElements objectAtIndex:i];

  if ([object isMemberOfClass:[NSString class]])
  {
    objectID = [[NSString alloc] initWithString:aString];
                Method returns an Objective-C object with a +1 retain count (owning reference)
  }

  if (objectID != nil)
  {
    [objectID release];                         Object released
                                                Reference-counted object is used after it is released
  }
}
```

ARC formalizes conventions,
automates the rules…

ARC formalizes conventions,
automates the rules…

what compilers are best at

Evolution of Objective-C

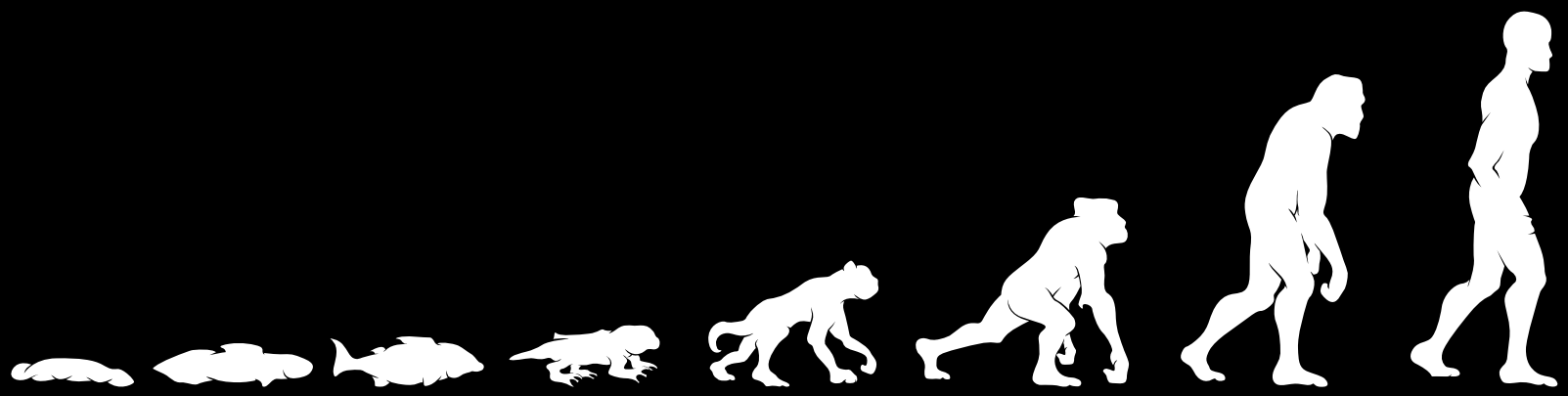
Simpler, safer through automation

- Object-Oriented C
- Retain Counting
- Properties
- Blocks

Evolution of Objective-C

Simpler, safer through automation

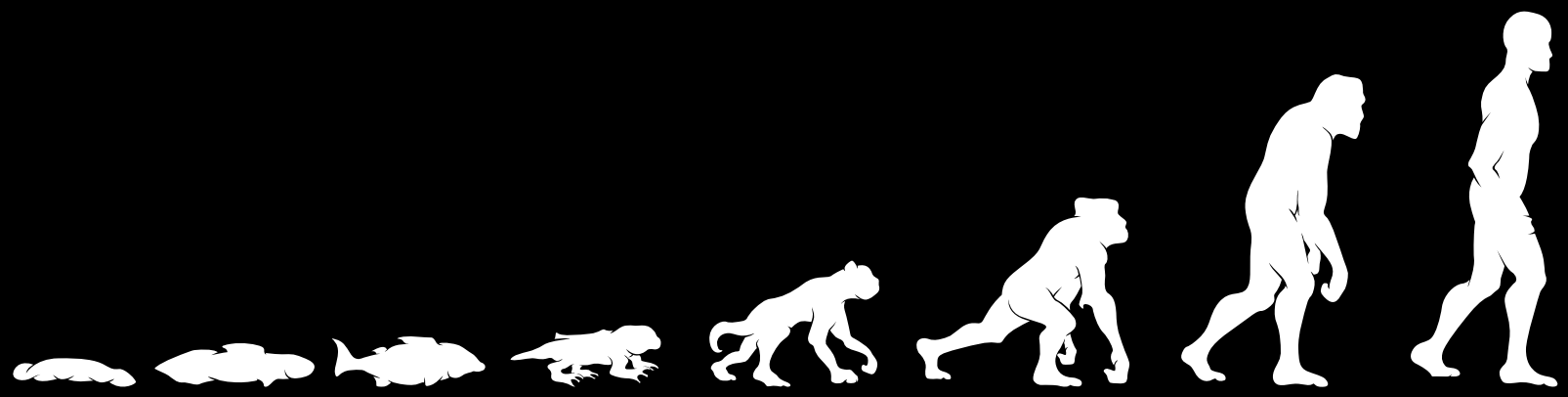
- Object-Oriented C
- Retain Counting
- Properties
- Blocks



Evolution of Objective-C

Simpler, safer through automation

- Object-Oriented C
- Retain Counting
- Properties
- Blocks
- ARC



What is ARC?

What Is ARC?

What Is ARC?

- Automatic memory management of Objective-C objects
  - Compiler obeys and enforces existing conventions

What Is ARC?

- Automatic memory management of Objective-C objects
  - Compiler obeys and enforces existing conventions
- Full interoperability with manual retain and release

What Is ARC?

- Automatic memory management of Objective-C objects
  - Compiler obeys and enforces existing conventions
- Full interoperability with manual retain and release
- New runtime features:
  - Weak pointers
  - Advanced performance optimizations

What ARC Is Not…

What ARC Is Not…

- No new runtime memory model

What ARC Is Not…

- No new runtime memory model
- No automation for malloc/free, CF, etc.

What ARC Is Not…

- No new runtime memory model
- No automation for malloc/free, CF, etc.
- No garbage collector
  - No heap scans
  - No whole app pauses
  - No non-deterministic releases

How Does ARC Work?

- Compiler adds retain/release/autorelease for you

How Does ARC Work?

- Compiler adds retain/release/autorelease for you

```
- (NSString*)greeting {
    return [[NSString alloc] initWithFormat:@"Hi %@", bloke.name];
}
```

How Does ARC Work?

- Compiler adds retain/release/autorelease for you

```
- (NSString*)greeting {
    return [[[NSString alloc] initWithFormat:@"Hi %@", bloke.name] autorelease];
}
```

How Does ARC Work?

- Compiler adds retain/release/autorelease for you

```
- (NSString*)greeting {
    return [[[NSString alloc] initWithFormat:@"Hi %@", bloke.name] autorelease];
}
- (NSString*)greeting {
    return [NSString stringWithFormat:@"Hi %@", bloke.name];
}
```

How Does ARC Work?

- Compiler adds retain/release/autorelease for you

```
- (NSString*)greeting {
    return [[[NSString alloc] initWithFormat:@"Hi %@", bloke.name] autorelease];
}
- (NSString*)greeting {
    return [NSString stringWithFormat:@"Hi %@", bloke.name];
}
```

- Different APIs that achieve same result have same cognitive load

How Do You Switch It On?

How Do You Switch It On?

- Project setting in Xcode

How Do You Switch It On?

- Project setting in Xcode

- New projects default to ARC

How Do You Switch It On?

- Project setting in Xcode

- New projects default to ARC
- Existing code must migrate to ARC

A Lot of Code Goes Away

```
- (void)dealloc {
    [identifier release];
    [title release];
    [HTML release];
    [category release];
    [super dealloc];
}
```

A Lot of Code Goes Away

Blocks Just Work

Blocks Just Work

Manual Reference Counting

```
dispatch_block_t getfoo(int i) {
  return [[^{
    print(i);
  } copy] autorelease];
}
```

Automatic Reference Counting

```
dispatch_block_t getfoo(int i) {
  return ^{
    print(i);
  };
}
```

ARC Must Be Reliable

- Problem:
  - Cannot rely on heuristics
    - Like the static analyzer does
- Solution:
  - Formalize and automate best practice
  - Four new rules

ARC Must Be Reliable

- Problem:
  - Cannot rely on heuristics
    - Like the static analyzer does
- Solution:
  - Formalize and automate best practice
  - Four new rules
    - Enforced by the compiler

Rule #1—No Access to Memory Methods

Rule #1—No Access to Memory Methods

- Memory management is part of the language
  - Cannot call retain/release/autorelease…
  - Cannot implement these methods

Rule #1—No Access to Memory Methods

- Memory management is part of the language
  - Cannot call retain/release/autorelease…
  - Cannot implement these methods
- Solutions
  - The compiler takes care of it
  - NSObject performance optimizations
  - Better patterns for singletons

Broken code, Anti-pattern

```
while ([x retainCount] != 0)
  [x release];
```

Rule #2—No Object Pointers in C Structs

Rule #2—No Object Pointers in C Structs

```
struct Pair {
  NSString *Name;  // retained
  int Value;
};
```

```
Pair *P = malloc(...);
...
...
free(P);  // Must drop references
```

Rule #2—No Object Pointers in C Structs

- Compiler must know when references come and go
  - Pointers must be zero initialized
  - Release when reference goes away

```
struct Pair {
  NSString *Name;  // retained
  int Value;
};
```

```
Pair *P = malloc(...);
...
...
free(P);  // Must drop references
```

Rule #2—No Object Pointers in C Structs

- Compiler must know when references come and go
  - Pointers must be zero initialized
  - Release when reference goes away

```
struct Pair {
  NSString *Name;  // retained
  int Value;
};
```

```
Pair *P = malloc(...);
...
...
free(P);  // Must drop references
```

- Solution: Just use objects
  - Better tools support
  - Best practice for Objective-C

Rule #3—No Casual Casting id ↔ void*

- Compiler must know whether void* is retained
- New APIs cast between Objective-C and Core Foundation-style objects

```
CFStringRef W = (__bridge CFStringRef)A;
NSString *X   = (__bridge NSString *)B;
CFStringRef Y = (CFStringRef)CFBridgingRetain(obj);
NSString *Z   = (NSString *)CFBridgingRelease(ref);
```

Rule #3—No Casual Casting id ↔ void*

- Compiler must know whether void* is retained
- New APIs cast between Objective-C and Core Foundation-style objects

```
CFStringRef W = (__bridge          CFStringRef)A;
NSString *X   = (__bridge          NSString *)B;
CFStringRef Y = (__bridge_retain   CFStringRef)C;
NSString *Z   = (__bridge_transfer NSString *)D;
```

Rule #4—No NSAutoreleasePool

- Compiler must reason about autoreleased pointers
- NSAutoreleasePool not a real object—cannot be retained

```
int main(int argc, char *argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Rule #4—No NSAutoreleasePool

- Compiler must reason about autoreleased pointers
- NSAutoreleasePool not a real object—cannot be retained

```
int main(int argc, char *argv[]) {
  @autoreleasepool {
    return UIApplicationMain(argc, argv, nil, nil);
  }
}
```

- Works even without ARC

How Do I Think About ARC?

How Do I Think About ARC?

- Think about ownership
  - Strong references keep objects alive
  - Objects deallocated when no more strong references remain

How Do I Think About ARC?

- Think about ownership
  - Strong references keep objects alive
  - Objects deallocated when no more strong references remain
- Think about your object graph

How Do I Think About ARC?

- Think about ownership
  - Strong references keep objects alive
  - Objects deallocated when no more strong references remain
- Think about your object graph
- Stop thinking about retain/release/autorelease

Weak References

Weak References

- Safe, nonretaining reference
  - Drop to nil automatically

Weak References

- Safe, nonretaining reference
  - Drop to nil automatically
- Now supported in ARC:

```
@property (weak) NSView *myView; // property attribute
id __weak myObject;              // variable keyword
```

What About Object Graph Cycles?

Like before, cycles cause leaks in ARC

- Standard patterns still work
  - Set property values to nil
  - Use weak references

Performance

- No performance difference from manual memory management
  - Peak memory high water mark lower
- No GC overhead
  - No delayed deallocations
  - No app pauses, no nondeterminism

This is ARC

- Simplified memory management model
  - Easier to learn
  - More productive
  - Easier to maintain
  - Safer and more stable

This is ARC

- Simplified memory management model
  - Easier to learn
  - More productive
  - Easier to maintain
  - Safer and more stable

You should use it

How ARC Works

Dave Zarzycki
Runtime Team

Memory Management Is Hard

- Lots of rules and conventions
- High hurdles for new developers
- Constant attention for existing developers
- Requires perfection

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [NSMutableArray array];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
@end
```

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
@end
```

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
@end
```

This array is autoreleased
System will deallocate it out from under us

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
@end
```

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

Now it is leaked
Need a dealloc method

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

This might invalidate x
Works only if there are other references to it

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [[_array lastObject] retain];
  [_array removeLastObject];
  return x;
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [[_array lastObject] retain];
  [_array removeLastObject];
  return [x autorelease];
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

Let's Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [[_array lastObject] retain];
  [_array removeLastObject];
  return [x autorelease];
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

Now we're returning a retained pointer
Violates convention; probably a leak

Let’s Write a Stack

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [[NSMutableArray array] retain];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [[_array lastObject] retain];
  [_array removeLastObject];
  return [x autorelease];
}
- (void) dealloc { [_array release]; [super dealloc]; }
@end
```

ARC Is Easy

- Write code naturally
- Break cycles when necessary
- Stop worrying about retains
- Write great apps

How ARC Works

- Automates what programmers do anyway
- Applies local rules
- Guarantees local correctness
- Optimizes away redundancies

Rules for Variables

- Objective-C, block pointers
- Locals, globals, parameters, instance variables…
- Four different kinds of ownership

Strong References

```
NSString *name;
```

- Default for all variables
- Like a retain property

Strong References

```
NSString *name;
```

```
__strong NSString *name;
```

- Default for all variables
- Like a retain property

Creating a Variable

```
NSString *name;
```

- Implicitly initialized to nil
- Safer for ARC-generated code
- Safer for your code

Creating a Variable

```
NSString *name;
```

```
NSString *name = nil;
```

- Implicitly initialized to nil
- Safer for ARC-generated code
- Safer for your code

Destroying a Variable

```
if (i < 10) {
  NSString *name = ...;
}
```

- Current value is implicitly released
- All variables, even ivars

Destroying a Variable

```
if (i < 10) {
  NSString *name = ...;
}
```

```
if (i < 10) {
  NSString *name = ...;
  [name release];
}
```

- Current value is implicitly released
- All variables, even ivars

Reading and Writing

```
name = newName;
```

- Nothing special for reads
- For writes:
  - Retain the new value
  - Release the old value

Reading and Writing

```
name = newName;
```

```
[newName retain];
NSString *oldName = name;
name = newName;
[oldName release];
```

- Nothing special for reads
- For writes:
  - Retain the new value
  - Release the old value

Autoreleasing References

```
- (void) runWithError:
    (NSError **) err {
  if (!valid_)
    *err = ...;
}
```

- Describes out-parameters
- Only on the stack
- Not for general use

Autoreleasing References

```
- (void) runWithError:
    (NSError **) err {
  if (!valid_)
    *err = ...;
}
```

```
- (void) runWithError:
    (__autoreleasing NSError **) err {
  if (!valid_)
    *err = [[... retain] autorelease];
}
```

- Describes out-parameters
- Only on the stack
- Not for general use

Unsafe References

```
__unsafe_unretained NSString *unsafeName = name;
```

- Like a traditional variable, or an assign property
- Not initialized
- No extra logic
- No restrictions
- Useful in global structs with constant @“…” strings

Weak References

```
__weak NSString *weakName = name;
char c = [weakName
            characterAtIndex: 0];
```

- Runtime manages reads and writes
- Becomes `nil` as soon as object starts deallocation

Weak References

```
__weak NSString *weakName = name;
char c = [weakName
            characterAtIndex: 0];
```

```
__weak NSString *weakName = nil;
objc_storeWeak(&weakName, name);
char c = [objc_readWeak(&weakName)
            characterAtIndex: 0];
objc_storeWeak(&weakName, nil);
```

- Runtime manages reads and writes
- Becomes `nil` as soon as object starts deallocation

Rules for Return Values

- Does this transfer ownership?
- Return values can be “returned retained”
- Similar concepts for other transfers into or out of ARC
- Decided by method family

Method Families

- Naming convention
- First "word" in first part of selector
- `alloc`, `copy`, `init`, `mutableCopy`, `new` transfer ownership
- Everything else does not

Normal Returns

```
- (NSString*) serial {
  return _serial;
}
```

- No transfer
- Retain immediately
- Autorelease after leaving all scopes

Normal Returns

```
- (NSString*) serial {
  return _serial;
}
```

```
- (NSString*) serial {
  NSString *returnValue
    = [_serial retain];
  return [returnValue autorelease];
}
```

- No transfer
- Retain immediately
- Autorelease after leaving all scopes

Retained Returns

```
- (NSString*) newSerial {
  return _serial;
}
```

- Passes back ownership
- Like a normal return without the autorelease

Retained Returns

```
- (NSString*) newSerial {
  return _serial;
}
```

```
- (NSString*) newSerial {
  NSString *returnValue
    = [_serial retain];
  return returnValue;
}
```

- Passes back ownership
- Like a normal return without the autorelease

Accepting a Retained Return

```
- (void) logSerial {
  NSLog(“%@\n”, [self newSerial]);
  ...
}
```

- Takes ownership
- Return value released at end of statement

Accepting a Retained Return

```
- (void) logSerial {
  NSLog("%@\n", [self newSerial]);
  ...
}
```

```
- (void) logSerial {
  NSString *returnValue
    = [self newSerial];
  NSLog("%@\n", returnValue);
  [returnValue release];
  ...
}
```

- Takes ownership
- Return value released at end of statement

Back to the Stack

```
- (id) init {
  if (self = [super init])
    _array = [NSMutableArray array];
  return self;
}
```

Back to the Stack

```
- (id) init {
  if (self = [super init])
    _array = [NSMutableArray array];
  return self;
}
```

```
- (id) init {
  if (self = [super init]) {
    NSMutableArray *oldArray = _array;
    _array = [[NSMutableArray array]
                retain];
    [oldArray release];
  }
  return self;
}
- (void) dealloc {
  [_array release];
  [super dealloc];
}
```

Back to the Stack

```
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
```

- Local rules make this work
- Unnecessary retain/release are optimized away

Back to the Stack

```
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
```

```
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  id result = [x retain];
  return [result autorelease];
}
```

- Local rules make this work
- Unnecessary retain/release are optimized away

Back to the Stack

```
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
```

```
- (id) pop {
  id x = [[_array lastObject] retain];
  [_array removeLastObject];
  id result = [x retain];
  [x release];
  return [result autorelease];
}
```

- Local rules make this work
- Unnecessary retain/release are optimized away

Back to the Stack

```
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
```

```
- (id) pop {
  id x = [[_array lastObject] retain];
  [_array removeLastObject];
  id result = [x retain];
  [x release];
  return [result autorelease];
}
```

- Local rules make this work
- Unnecessary retain/release are optimized away

Back to the Stack

```
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
```

```
- (id) pop {
  id x = [[_array lastObject] retain];
  [_array removeLastObject];
  return [x autorelease];
}
```

- Local rules make this work
- Unnecessary retain/release are optimized away

Natural Code Just Works

```
@implementation Stack { NSMutableArray *_array; }
- (id) init {
  if (self = [super init])
    _array = [NSMutableArray array];
  return self;
}
- (void) push: (id) x {
  [_array addObject: x];
}
- (id) pop {
  id x = [_array lastObject];
  [_array removeLastObject];
  return x;
}
@end
```

Putting It Together

- ARC follows the conventions for you
- Stop worrying about retains
- Focus on making great apps

Migrating to ARC

Migrating to ARC

Overview

- Migration Steps
- Common Issues
- Deployment Options
- Demo

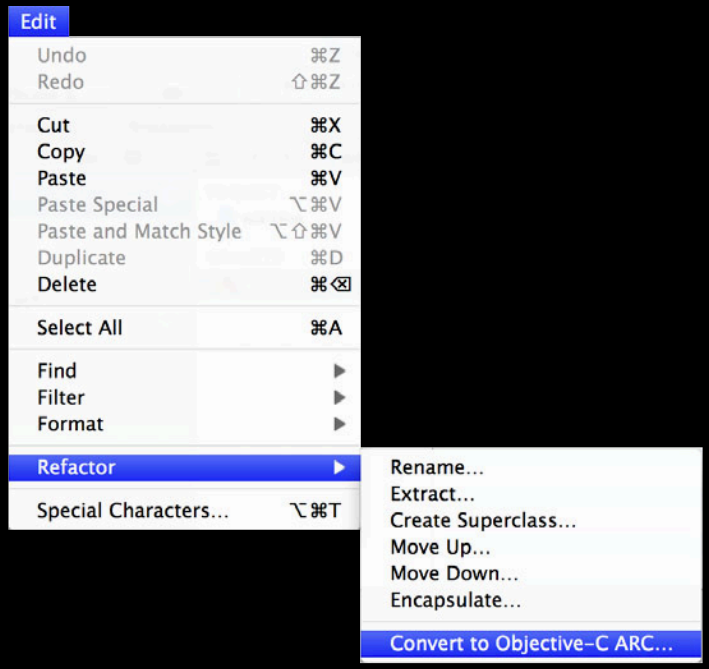
Migration Steps

- Compile your code with the latest LLVM compiler (3.0 or later)
- Use Convert to Objective-C ARC command in Xcode
- Fix issues until everything compiles
- Migration tool then modifies your code and project

“Convert to Objective-C ARC”

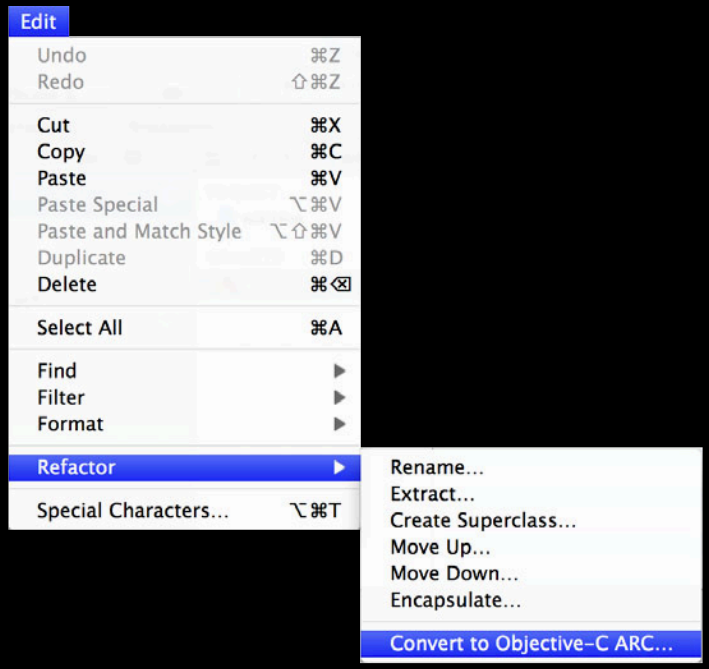
“Convert to Objective-C ARC”

- Supported in Xcode 4.2



“Convert to Objective-C ARC”

- Supported in Xcode 4.2
- Makes your source ARC compatible
  - Removes all calls to retain, release, autorelease
  - Replaces NSAutoreleasePool with @autoreleasepool
  - @property(assign) becomes @property(weak) for object pointers



Two Phases of ARC Migration

- Analysis
  - Migration problems presented as compile errors
  - Fix errors, run analysis again
- Conversion
  - After successful analysis, changes automatically applied
  - Turns on the ARC build setting in Xcode

Migrating to ARC

Missing method declarations

```
    @interface WHImageViewController : UIViewController
    @property (nonatomic, readonly) NSString *identifier;
    @end
    - (void)pushViewControllerWithIdentifier:(NSString *)identifier
                                     animated:(BOOL)animated {
  UIViewController *viewController;
  viewController = [[WHImageViewController alloc]
                    initWithIdentifier:identifier];

  [self pushViewController:viewController animated:animated];
  [viewController release];
}
```

Migrating to ARC

Missing method declarations

```
@interface WHImageViewController : UIViewController
@property (nonatomic, readonly) NSString *identifier;
@end
- (void)pushViewControllerWithIdentifier:(NSString *)identifier
                                animated:(BOOL)animated {
UIViewController *viewController;
viewController = [[WHImageViewController alloc]
                  initWithIdentifier:identifier];
```

Receiver type 'WHImageViewController' for instance message does not declare a method with selector 'initWithIdentifier:'

```
  [self pushViewController:viewController animated:animated];
  [viewController release];
}
```

Migrating to ARC

Unstructured Autorelease Pools

```
- (void)encodeFile {
  /* setup */
  BOOL done = NO, shouldDrain = NO;
  NSAutoreleasePool *loopPool = [NSAutoreleasePool new];
  while (!done) {
    /* part A. */
    if (shouldDrain) {
      [loopPool drain];
      loopPool = [NAutoreleasePool new];
    }
    /* part B. */
  }
  [loopPool drain];
}
```

Migrating to ARC

Unstructured Autorelease Pools

```
- (void)encodeFile {
  /* setup */
  BOOL done = NO, shouldDrain = NO;
  NSAutoreleasePool *loopPool = [NSAutoreleasePool new];




  while (!done) {
    /* part A. */
    if (shouldDrain) {
      [loopPool drain];
      loopPool = [NAutoreleasePool new];
    }
    /* part B. */
  }
  [loopPool drain];
}
```

Migrating to ARC

Unstructured Autorelease Pools

```
- (void)encodeFile {
  /* setup */
  BOOL done = NO, shouldDrain = NO;
  NSAutoreleasePool *loopPool = [NSAutoreleasePool new];




  while (!done) {
    /* part A. */
    if (shouldDrain) {
      [loopPool drain];
      loopPool = [NAutoreleasePool new];
    }
    /* part B. */
  }
  [loopPool drain];
}
```

Migrating to ARC

Unstructured Autorelease Pools

```
- (void)encodeFile {
  /* setup */
  BOOL done = NO, shouldDrain = NO;
  NSAutoreleasePool *loopPool = [NSAutoreleasePool new];
```

'NSAutoreleasePool' is unavailable: not available in automatic reference counting mode

```
  while (!done) {
    /* part A. */
    if (shouldDrain) {
      [loopPool drain];
      loopPool = [NAutoreleasePool new];
    }
    /* part B. */
  }
  [loopPool drain];
}
```

Migrating to ARC

Block-structured @autoreleasepool

Migrating to ARC

Block-structured @autoreleasepool

```
- (void)encodeFile {
  /* setup */
  BOOL done = NO;
  while (!done) {
    @autoreleasepool {
      /* part A. */
      /* part B. */
    }
  }
}
```

Migrating to ARC

Block-structured @autoreleasepool

```
- (void)encodeFile {
  /* setup */
  BOOL done = NO;
  while (!done) {
    @autoreleasepool {
      /* part A. */
      /* part B. */
    }
  }
}
```

Migrating to ARC

Block-structured @autoreleasepool

```
- (void)encodeFile {
  /* setup */
  BOOL done = NO;
  while (!done) {
    @autoreleasepool {
      /* part A. */
      /* part B. */
    }
  }
}
```

- Empty @autoreleasepool on Lion is 6x faster than Snow Leopard

Migrating to ARC

Declarations after case labels

```
- (void)decide {
  switch (currentState) {
  case INITIAL_STATE:;
    NSDate *date = [NSDate date];
    NSLog(@"started at %@", date);
    break;
  case MIDDLE_STATE:
  /* … */
  case FINAL_STATE:
    /* date is in-scope here */
    break;
  }
}
```

Migrating to ARC

Declarations after case labels

```
- (void)decide {
  switch (currentState) {
  case INITIAL_STATE:;
    NSDate *date = [NSDate date];
    NSLog(@"started at %@", date);
    break;
  case MIDDLE_STATE:



  /* … */
  case FINAL_STATE:
    /* date is in-scope here */
    break;
  }
}
```

Migrating to ARC

Declarations after case labels

```
- (void)decide {
  switch (currentState) {
  case INITIAL_STATE:;
    NSDate *date = [NSDate date];
    NSLog(@"started at %@", date);
    break;
  case MIDDLE_STATE:



  /* … */
  case FINAL_STATE:
    /* date is in-scope here */
    break;
  }
}
```

Migrating to ARC

Declarations after case labels

```
- (void)decide {
  switch (currentState) {
  case INITIAL_STATE:;
    NSDate *date = [NSDate date];
    NSLog(@"started at %@", date);
    break;
  case MIDDLE_STATE:
```

Switch case is in protected scope

```
  /* … */
  case FINAL_STATE:
    /* date is in-scope here */
    break;
  }
}
```

Migrating to ARC

Declarations after case labels require curly braces

```
- (void)decide {
  switch (currentState) {
  case INITIAL_STATE:
  {
    NSDate *date = [NSDate date];
    NSLog(@"started at %@", date);
    break;
  }
  /* … */
  case FINAL_STATE:
    /* date is in-scope here */
    break;
  }
}
```

Migrating to ARC

Singleton pattern

```
/* Shared Network Activity Indicator */
@interface ActivityIndicator : NSObject {
  int count;
}
+ (ActivityIndicator *)sharedIndicator;
- (void)show;
- (void)hide;
@end
```

Singleton Pattern

Overriding retain/release/autorelease

```
@implementation ActivityIndicator

- (id)retain { return self; }
- (oneway void)release {}
- (id)autorelease { return self; }
- (NSUInteger) retainCount
{ return NSUIntegerMax; }

 + (ActivityIndicator *)sharedIndicator {...}

 @end
```

Singleton Pattern

Overriding retain/release/autorelease

```
@implementation ActivityIndicator

- (id)retain { return self; }
- (oneway void)release {}
- (id)autorelease { return self; }
- (NSUInteger) retainCount
{ return NSUIntegerMax; }

 + (ActivityIndicator *)sharedIndicator {...}

 @end
```

Singleton Pattern

Overriding retain/release/autorelease

```
@implementation ActivityIndicator

- (id)retain { return self; }
- (oneway void)release {}
- (id)autorelease { return self; }
- (NSUInteger) retainCount
{ return NSUIntegerMax; }

 + (ActivityIndicator *)sharedIndicator {...}

 @end
```

ARC forbids implementation of ‘retain’

ARC forbids implementation of ‘release’

ARC forbids implementation of ‘autorelease’

ARC forbids implementation of ‘retainCount’

- ARC forbids these overrides

Singleton Pattern

Overriding +allocWithZone:

Singleton Pattern

Overriding +allocWithZone:

```
@implementation ActivityIndicator

+ (id)allocWithZone:(NSZone *)zone {
  static id sharedInstance;
  if (sharedInstance == nil)
    sharedInstance = [super allocWithZone:NULL];
  return sharedInstance;
}

+ (ActivityIndicator *)sharedIndicator {...}

@end
```

Singleton Pattern

Overriding +allocWithZone:

```
@implementation ActivityIndicator

+ (id)allocWithZone:(NSZone *)zone {
  static id sharedInstance;
  if (sharedInstance == nil)
    sharedInstance = [super allocWithZone:NULL];
  return sharedInstance;
}

+ (ActivityIndicator *)sharedIndicator {...}

@end
```

- init method will have to guard against multiple calls on the shared instance

Singleton Pattern

Use simple shared instance method

```
@implementation ActivityIndicator
+ (ActivityIndicator *)sharedIndicator {
  static ActivityIndicator *sharedIndicator;
  if (sharedIndicator == nil) sharedIndicator = [ActivityIndicator new];
  return sharedIndicator;
}
- (void)show {...}
- (void)hide {...}
@end
```

Singleton Pattern

Use dispatch_once for thread-safety

```
@implementation ActivityIndicator
+ (ActivityIndicator *)sharedIndicator {
  static ActivityIndicator *sharedIndicator;
  static dispatch_once_t done;
  dispatch_once(&done, ^{ sharedIndicator = [ActivityIndicator new]; });
  return sharedIndicator;
}
- (void)show {...}
- (void)hide {...}
@end
```

Singleton Pattern

Classes are singletons

Singleton Pattern

Classes are singletons

```
@implementation ActivityIndicator
```

Singleton Pattern

Classes are singletons

```
@implementation ActivityIndicator

static NSInteger count;
```

Singleton Pattern

Classes are singletons

```
@implementation ActivityIndicator

static NSInteger count;

+ (void)show {
  if (count++ == 0)
  [UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
}

+ (void)hide {
  if (count && --count == 0)
  [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
}

@end
```

Migrating to ARC

Delegate pattern

```
@protocol PageViewDelegate;
@interface PageView : NSView {
  id <PageViewDelegate> delegate;
  /* ... */
}
@property(assign) id <PageViewDelegate> delegate;
@end

@implementation PageView
@synthesize delegate;
/* ... */
@end
```

Delegate Pattern

Assign migrates to weak

```
@protocol PageViewDelegate;
@interface PageView : NSView {
  __weak id <PageViewDelegate> delegate;
  /* ... */
}
@property(weak) id <PageViewDelegate> delegate;
@end

@implementation PageView
@synthesize delegate;
/* ... */
@end
```

ARC Deployment

ARC Deployment

Lion and Memory Management

Lion and Memory Management

- ARC is really successful
  - No heap scans
  - No whole app pauses
  - No non-deterministic releases

Lion and Memory Management

- ARC is really successful
  - No heap scans
  - No whole app pauses
  - No non-deterministic releases
- ARC matches our framework semantics

Lion and Memory Management

- ARC is really successful
  - No heap scans
  - No whole app pauses
  - No non-deterministic releases
- ARC matches our framework semantics

GC is now deprecated

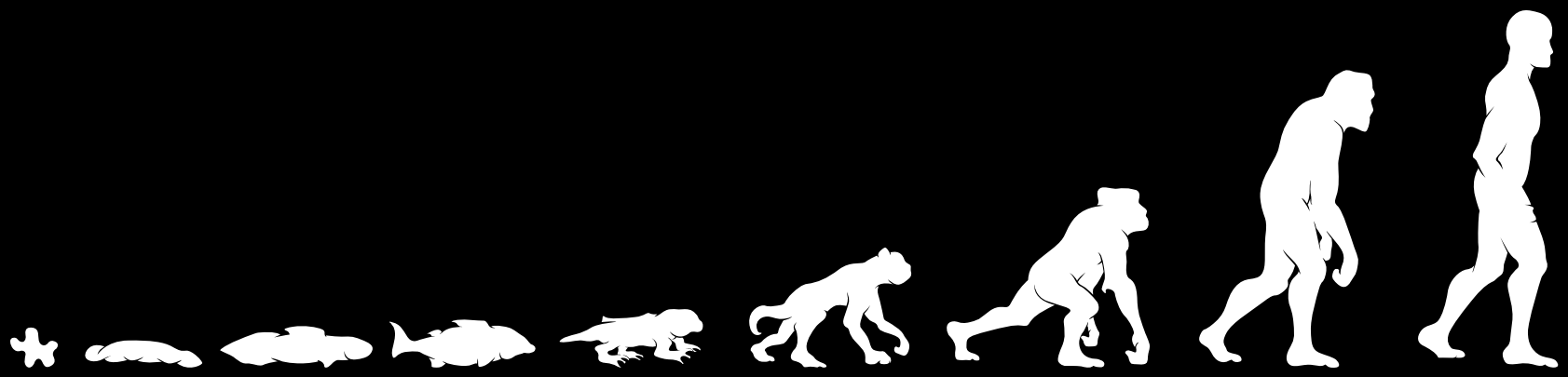
Demo

ARC Migration with Xcode 4.5

Evolution of Objective-C

Simpler and safer through automation

- Object-Oriented C
- Retain and Release
- Properties
- Blocks
- ARC



More Information

Michael Jurewitz
Developer Tools Evangelist
jury@apple.com

Programming with ARC Release Notes
http://developer.apple.com

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

Related Sessions

Modern Objective-C	Presidio Wednesday 10:15AM
What's New in LLVM	Pacific Heights Thursday 9:00AM
Migrating to Modern Objective-C	Nob Hill Thursday 3:15PM

Labs

Objective–C and Automatic Reference Counting Lab	Developer Tools Lab A Wednesday 2:00PM
Objective–C and Automatic Reference Counting Lab	Developer Tools Lab C Thursday 2:00PM

WWDC2012