New Features in Objective-C

David Chisnall

February 4, 2012
Some Buzzwords

Apple released OS X 10.7 and iOS 5 with ‘Apple’s LLVM 3.0 Compiler.’

- This is like everyone else’s LLVM 3.0 compiler, but with more bugs.
- Lots of new Objective-C features.
- Some require runtime support, some compiler support, most both.
- After being mocked for making Objective-C 2 the version after Objective-C 4, Apple no longer uses version numbers for Objective-C.
Summary

Better data hiding
- Better memory model
- Automatic reference counting
What Do We Support?

All of it!
(with clang/llvm 3.0 and the GNUstep Objective-C Runtime 1.6)

And some other stuff!
Better Data Hiding

- Ivars can now be declared in class extensions and `@implementation` contexts.
  - `@interface` is now *just* for the public interface.
  - Coming Soon: Modules.
Example

```c
// Foo.h
@interface Foo : NSObject
- (void)doStuff;
@end

// Foo_private.h
@interface Foo () {
@package
    id semiprivateState;
}
@end

// Foo.m
@implementation Foo {
    id privateState;
}
@end
```

// methods
@end
Method Families

new, alloc methods return a new owning reference to an instance of the receiver.

- **init** methods consume their receiver, return a new owning reference of the same type as the receiver.
- **copy, mutableCopy**, return a new owning reference of the same type as the receiver.

Information used by static analyser and by compiler
**Objective-C++ goes to 11!**

```cpp
auto foo = [NSMutableString new];
// ...  
[foo count];
```

```
$ clang str.mm -std=c++11  
str.mm:7:3: warning: 'NSMutableString' may not respond to 'count'
  [foo count];
  ^
```
Methods are functions that take two hidden arguments: \texttt{self} and \_\texttt{cmd}

- Block functions take one hidden argument: the block pointer
- How to map one to the other?
Blocks as IMPs

```
[obj doStuff: arg1 with: arg2];
// Calls:
imp(obj, @selector(doStuff:with:), arg1, arg2);
```

```
some_block_t block = ^(id, id, id) {...};
block(obj, arg1, arg2);
// Calls:
block->invoke(block, obj, arg1, arg2);
```

Block function as IMP needs the arguments rearranged!
A New Runtime Function

```c
__block int b = 0;
void * blk = ^(id self, int a) {
    b += a;
    return b;
};
// Apple provide this function:
IMP imp = imp_implementationWithBlock(blk);
// This is a GNUstep extension:
char *type = block_copyIMPTYPEEncoding_np(blk);
class_addMethod((objc_getMetaClass("Foo")),
    @selector(count:), imp, type);
free(type)
assert(2 == [Foo count: 2]);
assert(4 == [Foo count: 2]);
```
How It Works

imp_implementationWithBlock() returns a copy of a trampoline

- Trampoline stores the block and the invoke pointer just before the start of the function
- Moves argument 0 (self) over argument 1 _cmd.
- Copies block pointer over argument 0
- Jumps to block function
- Currently implemented for x86, x86-64 and ARM
Differences from Apple

block_copyIMPTypEncoding_np() works out the type of the method automatically, no need for explicit type encodings in source code.

- Block trampolines are dynamically allocated (and W^X safe!), no hard-coded limits on the number.
- One more thing...
Prototype-style OO in Objective-C

```objective-c
id obj = [SomeClass new];
id obj1 = [SomeClass new];
object_addMethod_np(obj, @selector(count:), imp, type);
[obj count: 2];
// This will throw an exception
// [obj1 count: 2];
obj1 = object_clone_np(obj);
// This will work
[obj1 count: 2];
```

Exposed via a category on NSObject in EtoileFoundation, used in LanguageKit.
Associated References

- `objc_setAssociatedObject()` - associate one object with another
  - `objc_getAssociatedObject()` - return the old object
  - Effectively adding slots to objects.
Automatic Reference Counting

More than just automatically inserting retain / release.

- Better memory model
- Explicit ownership qualifiers
- Denser code, better performance
- ABI compatible: ARC and non-ARC code can be mixed in the same binary (but not in the same compilation unit)
Explicit references to NSAutoreleasePool are not allowed in ARC mode.

- `@autoreleasepool` defines an autorelease pool scope
- Calls `objc_autoreleasePoolPush()` and `objc_autoreleasePoolPop()`
- These return / take `void*`
- Creating and destroying an autorelease pool just places a marker, does not have to allocate memory or create a new object

```c
id foo;
@autoreleasepool {
  foo = createsLoadsOfTemporaries();
}
```
Explicit Ownership Qualifiers

- **__strong** - Always holds an owning reference. Default for globals and ivars.
  - __autoreleasing - Holds an autoreleased variable. Default for locals.
  - __unsafe_unretained - Stores a pointer to an object or a nonsense value. User is responsible for ensuring it is valid.
  - __weak - Stores a zeroing weak reference. Will be set to 0 when the object is deallocated.
Weak References Are Deterministic

```objective_c
#import <Cocoa/Cocoa.h>
int main(void)
{
    __weak id foo;
    @autoreleasepool {
        id bar = [NSObject new];
        foo = bar;
    }
    printf("Weak reference: \%p\n", foo);
    return 0;
}
```

$ ./a.out
Weak reference: 0
Bridged Casts

```swift
id foo = bar;
void *ptr = foo; // <- This will error in ARC mode
```

- Object pointers are no just longer C pointers.
- Object pointers are not allowed in structures (except in C++).
- Casting from an object pointer to a C pointer requires a bridging cast.
- `(__bridge void*)obj` and `(__bridge id)ptr` do no ownership transfer.
- `(__bridge_retained void*)obj` gives a C pointer that is an owning reference.
- `(__bridge_transfer id)ptr` transfers ownership to an object pointer.
Example

```c
struct {
    void *ptr;
    // other stuff
} foo;
// Note: Not thread-safe.
void store(id obj) {
    foo.ptr = (__bridge_retained void*)obj;
}
id load(id obj) {
    id tmp = (__bridge_transfers id)foo.ptr;
    foo.ptr = NULL;
    return tmp;
}
```
Automatic Dealloc

ARC code may not explicitly send `-dealloc` messages.

- Classes compiled in ARC mode automatically have a `.cxx_destruct` method added that frees ivars.
- `-dealloc` is only for cleanup of other things (e.g. closing file descriptors).
- Call to `[super dealloc]` is implicit.
- With synthesized properties, you get accessors and dealloc for free.
Performance

Retain and release inserted by ARC are calls to `objc_retain()` / `objc_release()`.

- Smaller code and faster than a message send.
- Optimisers will elide redundant retain / release operations
- Less reason to use autoreleasing constructors: objects created with `+new` / `+alloc` will be automatically released when they go out of scope.
A common idiom is to retain and then autorelease an object then return it.

- This can result in it living in the autorelease pool for a long time.
- ARC has a mechanism for (roughly speaking) popping the top object from the autorelease pool.
- This un-autorelease means that the object is removed from the pool.
- Cheap autorelease pool scopes can mean a lot fewer temporaries.
ARC vs Legacy

- Loop calling accessor
- Accessor returns retained + autoreleased object

ARC version is more than twice as fast!
Objective-C++ and ARC

Objective-C objects are non-POD types
- Storing them in C++ containers Just Works™

```cpp
template <typename X> struct equal {
    bool operator()(const X a, const X b) const {
        return (a == b) || [a isEqual: b];
    }
};

template <typename X> struct hash {
    size_t operator()(const X s1) const {
        return (size_t)[s1 hash];
    }
};

// NSMutableArray equivalent:
std::vector<id> array;

// Dictionary from strings to weak objects:
std::unordered_map<NSString*, __weak id, hash<NSString*>, equal<NSString*> > d;
```
The ARC Migration Tool

Compile with `-ccc-arcmt-check` to flag things that will need manually changing for ARC code.

- Fix them.
- Compile with `-ccc-arcmt-modify` to rewrite the file to using ARC.
- Profit (from fewer bugs and simpler code)
- Think about object ownership, not about memory management.
Faster Message Sending

Implemented `objc_msgSend()` for x86, x86-64, ARM.

- Performance almost as fast as cached call.
- Varies between CPUs, generally now message send cost is less than double the cost of a function call.
- Better than the theoretical best speed with the classical GNU message lookup.
- Microbenchmark shows same speed as OS X, where OS X is using its fast, cached code path.
- 10% reduction in total code size for GNUstep-base.
- Enable with `-fno-objc-legacy-dispatch`
Message Sending Speeds

Message sending loop on an 800MHz ARM Cortex A8

- class message
- objc_msgSend()
- auto-cached
- instance message
- IMP call
Small Objects

Idea from Smalltalk (also in OS X 10.7), which Smalltalk stole from Lisp in the '70s.

- Small objects hidden in pointers.
- 32-bit architectures: 1 bit for small int flag.
- 64-bit architectures: 7 small object classes.
- Saves memory allocation for lots of short-lived temporaries.
Small Object Kinds

All can be used as ‘normal’ NSNumber or NSStrings:

- NSSmallInt - 31-bit / 61-bit signed integer.
- NSSmallExtendingDouble - double with last 1 bit of mantissa repeated.
- NSSmallRepeatingDouble - double with last 2 bits of mantissa repeated.
- GSTinyString - (up to) 7 ASCII characters in a string.

GNUstep-base allocates over 20 GSTinyString instances before main()!
And One More Thing...

@interface GSSomeClass
+ (void)foo;
@end

#ifndef __APPLE__
@compatibility_alias NSSomeClass GSSomeClass;
#endif

@implementation GSSomeClass
+ (void)foo {
    NSLog(@"Foo!");
}
@end

...  
// This works fine
[NSSomeClass foo];
// WTF? This doesn’t?
[NSClassFromString(@"NSSomeClass") foo];
And One More Thing...

@interface GSSomeClass
+ (void)foo;
@end

#ifndef __APPLE__
@compatibility_alias NSSomeClass GSSomeClass;
#endif

@implementation GSSomeClass
+ (void)foo {
    NSLog(@"Foo!");
}
@end

... // This works fine
[ NSSomeClass foo ];
// So does this! Thanks Niels!
[ NClassFromString( @"NSSomeClass" ) foo ];
And More Platforms!
Questions?