objective-c

jano@jano.com.es

Contact: http://jano.tel/

What is it?

objects are C structs

reflection

```
@interface Person @end
@implementation Person @end
```

```
[NSString alloc]
```

Objective-C is object oriented and dynamic.

```
struct Person {
  Class isa;
}
```

```
objc_msgSend(objc_getClass("NSString"),
             sel_registerName("alloc"))
```

alloc	0x00002710

```
struct objc_class {
  Class isa;
  Class super_class
  char *name
  long version
  long info
  long instance_size
  objc_ivar_list *ivars
  objc_method_list **methodLists
  objc_cache *cache
  objc_protocol_list *protocols
}
```

“Objective-c has message passing, not method calls”

The runtime is a library that implements Objective-C on top of C.

Objective-C is dynamic and object oriented. Both properties are implemented in the runtime.

Objective-C is object oriented because it has objects. Those objects are written in Objective-C and translated to C functions and structs by the compiler.

Objective-C is dynamic because it has features that other languages only have during compile time, like reflection.
Reflection is the ability to observe and alter its own state. In other words, “the ability of introspect and intervene on reified language elements during execution”.

To accomplish reflection, all elements of an Objective-C program, like objects, methods, properties, categories, protocols, etc. are represented by C structs defined in the runtime. It also provides functions to observe and change these elements during run time.

If you look at a language like C, calls to C functions are replaced with a jump to the address of that function implementation. Dynamic languages don’t work that way. There is a mechanism embedded with your program (the runtime) that decides what implementation to run for each method. This information is kept on a “virtual table” that may be array (C++) or hash based (Objective-C). This makes manipulation much easier.

Method or functions calls are translated by the compiler as a jump to a fixed point in code. A message send however, requires a decision in runtime to associate the message with its implementation.

The phrase “Objective-C has message passing, not method calls” is related to this. In Objective-C you don’t jump to the implementation directly, you pass a message saying you want to execute a method, and the runtime decides what implementation to run in response. If later you want to change the implementation of a method, you do it on the virtual table (for example, loading a category).

Let’s say what is a message and what is a method call... [next slide]

A message call

```
// method signature
+ (NSString*)randomStringWithLength:(NSUInteger)length {
    // ... method implementation
}

// message
[NSString randomStringWithLength:10]

// selector
randomStringWithLength:

// receiver
NSString

// parameters
10
```

A **message** is an invocation of the method. A **function call** is a jump to an implementation address.

The difference is that a message requires a decision in run time to associate the message with its implementation. One happens at run time, the other at compile time.

For example, let's say I send the message [NSString randomStringWithLength:10]. That's not a method of the class NSString, unless I add it during run time.
How do I do that? I can use a category, load a module using the runtime (objc_loadModules) or NSBundle, or create the code. All these actions change a pair (method name, implementation address) on a method hash table of the object.

*A **module** is the minimum independent compilable unit of code. A m file generates one module per object.

So, the compiler uses the data structures and function calls of the runtime to generate a runtime layer for every Objective-C program created. This means that the runtime library (/usr/lib/libobjc.A.dylib) is linked on all programs, and is therefore, available to use in your own code.
Apple's runtime is mostly written in C, with pieces in C++ and assembler.

Runtime features

Introspection
Dynamic typing, binding, linking
Categories
Code creation and swizzling
Forwarding proxy support
Non-fragile instance variables

Every solution is one indirection away. These are some features that depend on method and ivar indirection.

Introspection. The ability to inspect the properties and methods of a class.
Late method binding. The implementation of a message is decided during runtime by the object that receives the message.

Class and method creation and swizzling. Replacement and creation of existing code, eg: KVO, auto synthesize, @dynamic.
Forwarding proxy support. The ability for a class to work as proxy of another. This can be used to mimic multiple inheritance.
Categories. Adds methods to an existing class at runtime without recompiling.
Non-fragile instance variables. The ability of compiled code to work with new superclasses that have more instance variables than they had when the code was compiled. eg: auto synthesize.

Fragile base class

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

@interface Person : NSObject
@property NSString *_name;
@end

@implementation Person {
    NSInteger age;
}
@end
```

NSView (Def Leopard)	
0	Class isa
4	NSRect bounds
20	NSView *superview
24	NSColor *bgColor
28	**NSSet *touchedPaws**

PetShopView	
0	Class isa
4	NSRect bounds
20	NSView *superview
24	NSColor *bgColor
28	**NSSet *touchedPaws**
32	NSArray *kittens
36	NSArray *puppies

http://www.sealiesoftware.com/blog

A class is fragile when any change requires recompilation of all its subclasses. Even if the change was just adding private methods or variables, binary compatibility is lost.

Objective-C always had non-fragile methods due to dynamic dispatch. Objective-C 2.0 added a layer of indirection in variables that lets the dynamic linker adjust instance layout at runtime.
In this image, the runtime detects a new version of the base class and increases the offset of the existing ivars to accommodate the new ivars of the base class.

Non-fragile instance variables allows the following improvements:
- ivar autosynthesis. This is the runtime creation of property ivars not declared in the class interface.
- ivars declared outside a class's interface.

@dynamic

@dynamic means: don’t auto synthesize the accessors, I’ll provide the code myself.
Core Data uses this in managed objects to generate the accessors during runtime.

@dynamic example 1/2

```objc
#import <objc/runtime.h>
#import <Foundation/Foundation.h>

@interface Person : NSObject
@property (copy) NSString *name;
@end

@interface Person()
@property (nonatomic,strong) NSMutableDictionary *dic;
@end

@implementation Person
@dynamic name;

-(id) init {
    if (self = [super init]){
        _dic = [NSMutableDictionary dictionary];
    }
    return self;
}

@end

int main(int argc, char *argv[]){
    @autoreleasepool {
        Person *p = [Person new];
        p.name = @"Dolores";
        NSLog(@"%@",p.name);
    }
}
```

Let’s say we have this Person class with a name property and a mutable dictionary.
We are going to generate accessors to store the property in the dictionary, instead of using the property.
Let’s imagine that this is Core Data and that dictionary is sqlite :P

@dynamic example 2/2

```objc
// generic getter
id propertyIMP(id self, SEL _cmd) {
    return [((Person*)self).dic objectForKey:NSStringFromSelector(_cmd)];
}


// generic setter
void setPropertyIMP(id self, SEL _cmd, id value) {

    // setName --> name
    NSMutableString *key = [NSStringFromSelector(_cmd) mutableCopy];
    [key deleteCharactersInRange:NSMakeRange(0, 3)];
    [key deleteCharactersInRange:NSMakeRange([key length]-1, 1)];
    [key replaceCharactersInRange:NSMakeRange(0, 1) withString:[[key substringToIndex:1]
lowercaseString]];

    [((Person*)self).dic setObject:value forKey:key];
}


+ (BOOL)resolveInstanceMethod:(SEL)aSEL {
    if ([NSStringFromSelector(aSEL) hasPrefix:@"set"]) {
        class_addMethod([self class], aSEL, (IMP)setPropertyIMP, "v@:@");
    } else {
        class_addMethod([self class], aSEL,(IMP)propertyIMP, "@@:");
    }
    return YES;
}
```

And this is how we create the dynamic accessors for any property at all.
Also, an example use of resolveInstanceMethod:.

(explain the code)

type encoding is return type, id, self, arguments

Swizzling

Swizzling is the replacing of methods of classes in an instance.
When you replace a method the cache is also filled with it.

When we add KVO to a class:
• Apple code creates a subclass that overrides methods with versions that send -willChangeValueForKey: and -didChangeValueForKey: to any registered observer before and after calling the original class methods.
• The isa pointer of the original class is replaced with this new subclass.
• When all the observers are removed, the original class is swizzled back.

The subclass replaces the class method to try to hide its existence. Example:
[person class]; // Person
object_getClass(person)); // NSKVONotifying_Person

-- SWIZZLING STUFF BUT NOT MUCH TALKING ABOUT THE RUNTIME --

Why not a category?

• If you use a category and the method is overridden in another category, it's undefined which implementation wins.
• If you use a category, the original implementation is lost.

Method Swizzling

• Impacts every instance of the class
• Highly transparent. All objects retain their class.
• Requires unusual implementations of override methods, like C functions with self and _cmd as parameters. Or you can use method_exchangeImplementations instead C function pointers.
• The swizzling fills the cache.

ISA Swizzling

• Only impacts the targeted instance
• Objects change class (though this can be hidden by overriding class)
• Override methods are written with standard subclass techniques

Problems with Swizzling

• Method swizzling is not atomic. Do it in +load if you can to avoid concurrency issues.
• Changes behavior of un-owned code. App may be rejected if you swizzle Apple's API.
• Possible naming conflicts. You can use function pointers directly to avoid this. See Dangers of swizzling.
• Swizzling changes the method's arguments
• The order of swizzles matters. Use +load so there is an order: superclass first.
• Difficult to understand (may look recursive)
• Difficult to debug

KVO trickery: Dynamic Subclass

```
- (Class) dynamicallySubclass:(id)instance {

    const char * prefix = "DynamicSubclass_";
    Class instanceClass = [instance class];
    NSString * className = NSStringFromClass(instanceClass);

    BOOL isDynamicSubclass = strncmp(prefix, [className UTF8String], strlen(prefix)) == 0;
    if (isDynamicSubclass) { return [instance class]; }

    NSString *subclassName = [NSString stringWithFormat:@"%s%@", prefix, className];
    Class subclass = NSClassFromString(subclassName);

    BOOL classExists = subclass!=nil;

    if (classExists) {
        object_setClass(instance, subclass);

    } else {
        subclass = objc_allocateClassPair(instanceClass, [subclassName UTF8String], 0);
        if (subclass != nil) {

            // subclass created, now change it:
            //   1. create the a kvoProperty method for each property
            //   2. replace the imp of each property with the imp of the kvoProperty method

            // method swizzling would be:
            //   IMP newImp = class_getMethodImplementation([self class], @selector(kvoProperty));
            //   class_addMethod(subclass, @selector(name), newImp, "v@:");

            objc_registerClassPair(subclass);
        }
    }

    return subclass;
}
```

The KVO code is not public, but we can imagine that the subclass is created dynamically like we see here.

(explain the code)

the kvoProperty could be a generic method that finds out the property name from the _cmd, I don’t know, maybe the GNUStep project has an example

KVO trickery: ISA Swizzling

```
#import <objc/runtime.h>
 #import <Foundation/Foundation.h>

 @interface NSObject (IsaSwizzling)
 - (void)setClass:(Class)aClass;
 @end
```

```
@implementation NSObject (IsaSwizzling)

- (void)setClass:(Class)aClass {
    NSAssert(class_getInstanceSize([self class]) == class_getInstanceSize(aClass),
             @"Classes must be the same size to swizzle. Did you add ivars?");
    object_setClass(self, aClass);
}

@end
```

Once created we replace the isa pointer.

Be careful not to add instance variables or the increased size will overwrite memory crashing the app!

Implementation

~~NeXT~~

~~Apple's Legacy runtime (32bit)~~

Apple's Objective-C 2.1

~~Étoilé Runtime~~

GNUStep

NeXT. Never released. Superseded by Apple.
Apple's Legacy runtime. An improved version of the NeXT Objective-C runtime. Open source but not portable outside Darwin.
Apple's Objective-C 2.1. Open source.
Étoilé Runtime. A research prototype merged with GNUStep.
GNUStep
- GNUStep is a GNU implementation of the Objective-C runtime and Cocoa.
- The license is GPL for applications, and LGPL for libraries.
- It is available as binary in macports, or source in svn, mirrored in github/gnustep.
- NSObject is in NSObject.m (svn), or NSObject.m (github).

Apple's runtime is open source: http://www.opensource.apple.com/source/objc4/

Compilers

GCC

~~Apple's GCC fork 4.2.1~~

~~LLVM-GCC~~

Clang

the object struct

What is an object?

```
// Foundation.framework/NSObject.h
@interface NSObject <NSObject> {
    Class isa;
}
// ... bunch of methods
@end

struct NSObject {
    Class isa;
}

// /usr/include/objc/objc.h
typedef struct objc_class *Class;

struct NSObject {
    objc_class *isa;
}

typedef struct objc_object {
  Class isa;
} *id;
```

Objective-C is mostly C and objects.

Objects respond to messages, can be queried at runtime without knowing their exact type, placed in collections, compared for equality, and share a common set of behavior.

Because Class is a struct with a pointer to its metaclass, it is an object too.

The missing piece is objc_class...

Legacy class struct

```
// /usr/include/objc/runtime.h
struct objc_class {
    Class isa;
#if !__OBJC2__
    Class super_class                                       OBJC2_UNAVAILABLE;
    const char *name                                        OBJC2_UNAVAILABLE;
    long version                                            OBJC2_UNAVAILABLE;
    long info                                               OBJC2_UNAVAILABLE;
    long instance_size                                      OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars                            OBJC2_UNAVAILABLE;
    struct objc_method_list **methodLists                   OBJC2_UNAVAILABLE;
    struct objc_cache *cache                                OBJC2_UNAVAILABLE;
    struct objc_protocol_list *protocols                    OBJC2_UNAVAILABLE;
#endif
} OBJC2_UNAVAILABLE;
```

We notice two things:

- The contents are deprecated in Objective-C 2.0.
- It points to another Class, commonly referred as "metaclass".

Before Objective-C 2.0 all Objective-C objects were instances of the C struct objc_object. Now the type is opaque and this struct is marked deprecated. You can still access the fields using C functions, but not reference them directly. This gives Apple engineers more liberty to change the layout of the struct without breaking compatibility. The struct is still worth a look because the current version is probably very similar.

Because the first field is an isa, message sending works just as with NSObject. So a Class is an object too, and we can send it messages with selectors of class methods. The following is equivalent:

```
[NSObject alloc]
[[NSObject class] alloc]
```

Just as this objc_method_list of the current class points to the instance methods, the objc_method_list of the metaclass points to the class methods. The metaclass isa just points to itself.

You can use the debugger to create an object and then do p *object. You will see a struct with a field Class isa, but since the type is opaque, there are no more fields to print than isa. We have to use functions of the runtime.

-- SKIP, XCODE EXAMPLE --

```
expr (long)class_getInstanceSize((Class)[o class])
p (int)class_getVersion((Class)[o class])
```

If that syntax seems foreign to you, read the LLDB tutorial.

The class describes the data (allocation size, ivar types, layout) and behaviour (methods it implements). Because there is only one isa pointer, there is no multiple inheritance.

Modern class struct 1/3

```
typedef struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
#ifdef __LP64__
    uint32_t reserved;
#endif
    const uint8_t * ivarLayout;
    const char * name;
    const method_list_t * baseMethods;
    const protocol_list_t * baseProtocols;
    const ivar_list_t * ivars;
    const uint8_t * weakIvarLayout;
    const property_list_t *baseProperties;
} class_ro_t;
```

Read only.

ivars are in the read only section. Categories can’t change them.

Modern class struct 2/3

```
typedef struct class_rw_t {
    uint32_t flags;
    uint32_t version;
    const class_ro_t *ro;
    union {
        method_list_t **method_lists;  // RW_METHOD_ARRAY == 1
        method_list_t *method_list;    // RW_METHOD_ARRAY == 0
    };
    struct chained_property_list *properties;
    const protocol_list_t ** protocols;
    struct class_t *firstSubclass;
    struct class_t *nextSiblingClass;
} class_rw_t;
```

Read/write.

firstSubclass and nextSiblingClass point to the subclasses of the class. They can’t be used because access to them use a lock which is not accessible outside the runtime library. Without using that lock you could crash if another thread is reading them.

Modern class struct 3/3

```
typedef struct class_t {
    struct class_t *isa;
    struct class_t *superclass;
    Cache cache;
    IMP *vtable;
    // class_rw_t * plus custom rr/alloc flags
    uintptr_t data_NEVER_USE;
    class_rw_t *data() const {
        return (class_rw_t *)(data_NEVER_USE & ~(uintptr_t)3);
    }
    // ...
} class_t;
```

And finally, this is the class type.

Objective-C: a thin layer on top of C

Example: [Person new]

```
#import <objc/runtime.h>

@interface Person
@end

@implementation Person
+(id)new {
    Class cls = objc_getClass("Person");
    id obj = class_createInstance(cls, class_getInstanceSize(cls));
    return obj;
}
@end


int main(int argc, char *argv[]){
    @autoreleasepool {
        [Person new];
    }
}
```

A root object. Since it doesn't inherit new, I'm creating one. I'm using createInstance, which internally does a calloc of the struct and assigns the class. You can look at it in the Object.m, which suports the legacy runtime. The modern runtime version is probably very similar.

We can rewrite this objective-c to C++ using "clang -rewrite-objc main.m". The answer will be mostly C.

Person.cpp 1/3

```
// ...

#define __OFFSETOFIVAR__(TYPE, MEMBER) ((long long) &((TYPE *)0)->MEMBER)
#include <objc/runtime.h>

#ifndef _REWRITER_typedef_Person
#define _REWRITER_typedef_Person
typedef struct objc_object Person;
#endif

/* @end */

// @implementation Person

static id _C_Person_new(Class self, SEL _cmd) {
    Class cls = objc_getClass("Person");
    id obj = class_createInstance(cls, class_getInstanceSize(cls));
    return obj;
}
// @end


int main(int argc, char *argv[]){
    @autoreleasepool {
        ((id (*)(id, SEL))(void *)objc_msgSend)(objc_getClass("Person"), sel_registerName("new"));
    }
}

struct _objc_method {
     SEL _cmd;
     char *method_types;
     void *_imp;
};
```

The new method is now a C function.

The call to the C function is made with an objc_msgSend function call. This function is written in assembler in file objc-msg-x86_64.s of the runtime, and chooses the implementation of the messages we send.

The objc_method is the definition of Method. It has a name, pointer to implementation, and a string describing the return and parameter types. The format is in “Objective-C Runtime Guide: Type Encodings”.

Person.cpp 2/3

```
static struct {
     struct _objc_method_list *next_method;
     int method_count;
     struct _objc_method method_list[1];
} _OBJC_CLASS_METHODS_Person __attribute__ ((used, section ("__OBJC, __cls_meth")))= {
     0, 1
     ,{{(SEL)"new", "@16@0:8", (void *)_C_Person_new}
      }
};

struct _objc_class {
     struct _objc_class *isa;
     const char *super_class_name;
     char *name;
     long version;
     long info;
     long instance_size;
     struct _objc_ivar_list *ivars;
     struct _objc_method_list *methods;
     struct objc_cache *cache;
     struct _objc_protocol_list *protocols;
     const char *ivar_layout;
     struct _objc_class_ext  *ext;
};

static struct _objc_class _OBJC_METACLASS_Person __attribute__ ((used, section ("__OBJC, __meta_class")))=
{
     (struct _objc_class *)"Person", 0, "Person", 0,2, sizeof(struct _objc_class), 0
     , (struct _objc_method_list *)&_OBJC_CLASS_METHODS_Person
     ,0,0,0,0
};

static struct _objc_class _OBJC_CLASS_Person __attribute__ ((used, section ("__OBJC, __class")))= {
     &_OBJC_METACLASS_Person, 0, "Person", 0,1,0,0,0,0,0,0,0
};
```

At the top there is a class method initialization.

Then we have the class definition, and the initialization of the class and metaclass. Some values are 0 because they are initialized at run time. The metaclass is the class of the class. The same structure is used, but the metaclass is used to store class methods.

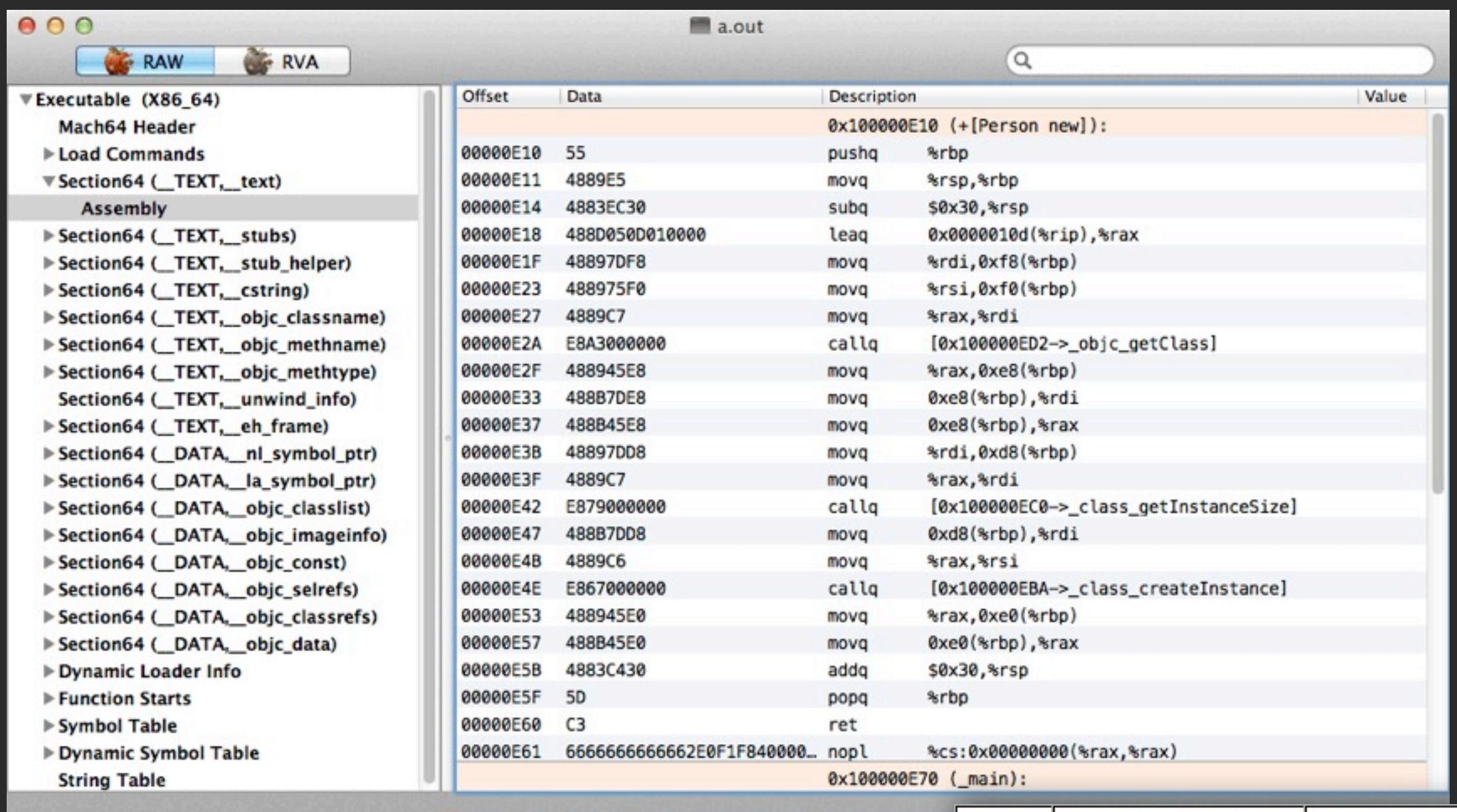
Person.cpp 3/3

```
struct _objc_symtab {
     long sel_ref_cnt;
     SEL *refs;
     short cls_def_cnt;
     short cat_def_cnt;
     void *defs[1];
};

static struct _objc_symtab _OBJC_SYMBOLS __attribute__((used, section ("__OBJC, __symbols")))= {
     0, 0, 1, 0
     ,&_OBJC_CLASS_Person
};


struct _objc_module {
     long version;
     long size;
     const char *name;
     struct _objc_symtab *symtab;
};

static struct _objc_module _OBJC_MODULES __attribute__ ((used, section ("__OBJC, __module_info")))= {
     7, sizeof(struct _objc_module), "", &_OBJC_SYMBOLS
};
```

A module is the smallest unit of code and data that can be linked. A objc_module structure is created for each .m file with an object declaration. _symtab contains the number of classes and categories declared in the module.

The __OBJC means this is a segment with information for the Objective-C runtime library.

MachOView



	objc_msgSend objc_msgSend_fpret		objc_msgSend_stret	
	receiver	SEL	receiver	SEL
i386	eax*	ecx	eax*	ecx
x86_64	rdi	rsi	rsi	rdx

```
Desktop — bash — 104×46
~/Desktop $ clang -lobjc main.m
~/Desktop $ otool -L a.out
a.out:
```

clang -lobjc main.m
otool -L a.out
otool -L /usr/lib/libobjc.A.dylib
otool -L /usr/lib/libSystem.B.dylib
nm a.out
class-dump a.out

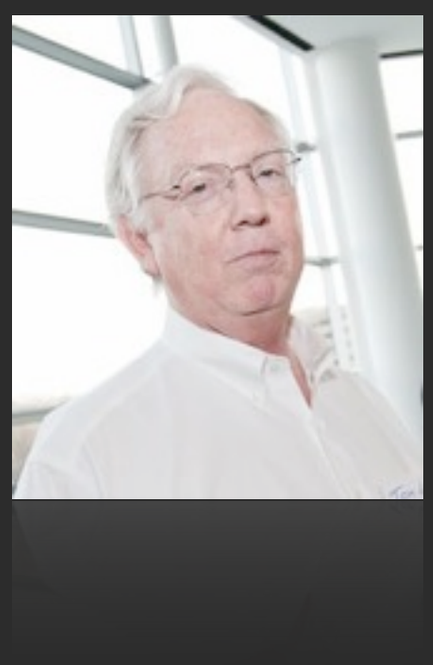
There is more info on this in the Mach-O documentation. Mach-O is ABI (application binary interface) for OS X. It's the equivalent of ELF in Linux or PE in Windows. In other words, the Mach-O defines the structure of an executable, where is the data, the executable code, the architecture, etc.

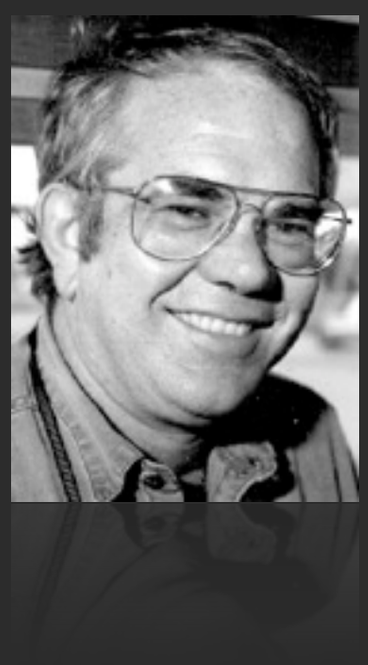
You can use otool or the "MachOViewer" (a 3rd party app) to see the structure of an executable.

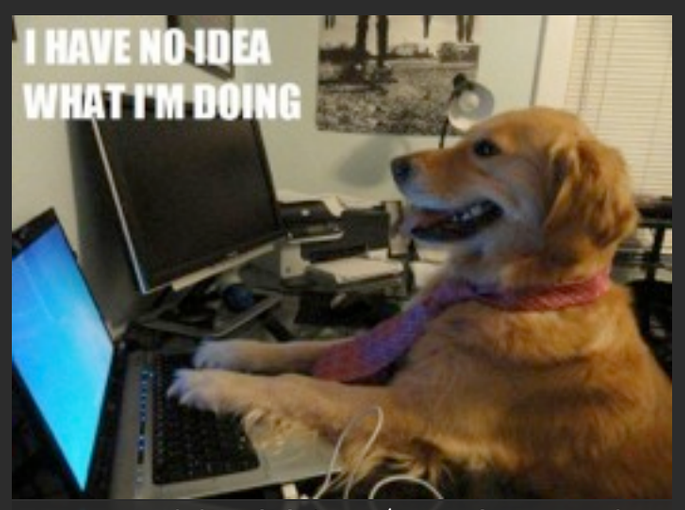
If we do "otool -L a.out" we get the linked libraries. We can repeat on the libraries to see their dependencies. [CLICK] [CLICK]
Use "nm a.out" to print the symbol table.

The registers used indicate the architecture. [CLICK]

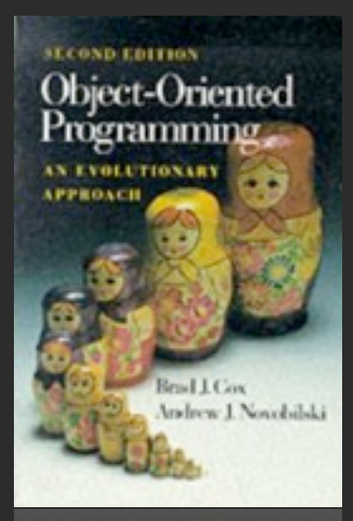
This one is Hopper [CLICK]. It's 29€. If you don´t want to spend the money, there is a command line tool called "otx" which also produces annotated assembler.

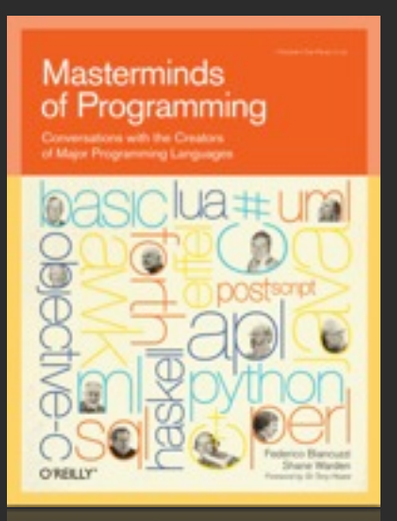






ninjasandrobots.com/you-need-some-experience





SmallTalk Issue Of Byte, Volume 6, Number 8, August 1981

Brad Cox and Tom Love created Objective-C because they needed object orientation. At the time they were trying to figure out which language to use for international teams building telephone switches. Brad had read an article about Smalltalk on the Byte magazine, and thought he could implement the same features in C without much fuss. Not that he had a lot of choices. The popular languages were Ada, Pascal, COBOL, FORTRAN, C, Lisp, and Smalltalk. Because Xerox wouldn't sell Smalltalk, they built their own language on top of C.

The first Objective-C compiler was really a preprocessor translating objc to C.

objc_sendMsg

objc msgSend

```
id objc_msgSend(id receiver, SEL name, arguments...) {
    IMP function =
        class_getMethodImplementation(receiver->isa, name);
    return function(arguments);
}
```

Every Objective-C method is translated to an msgSend function. The function is automatically given 2 arguments with the receiver and the name of the method. The rest are the arguments of the original method.

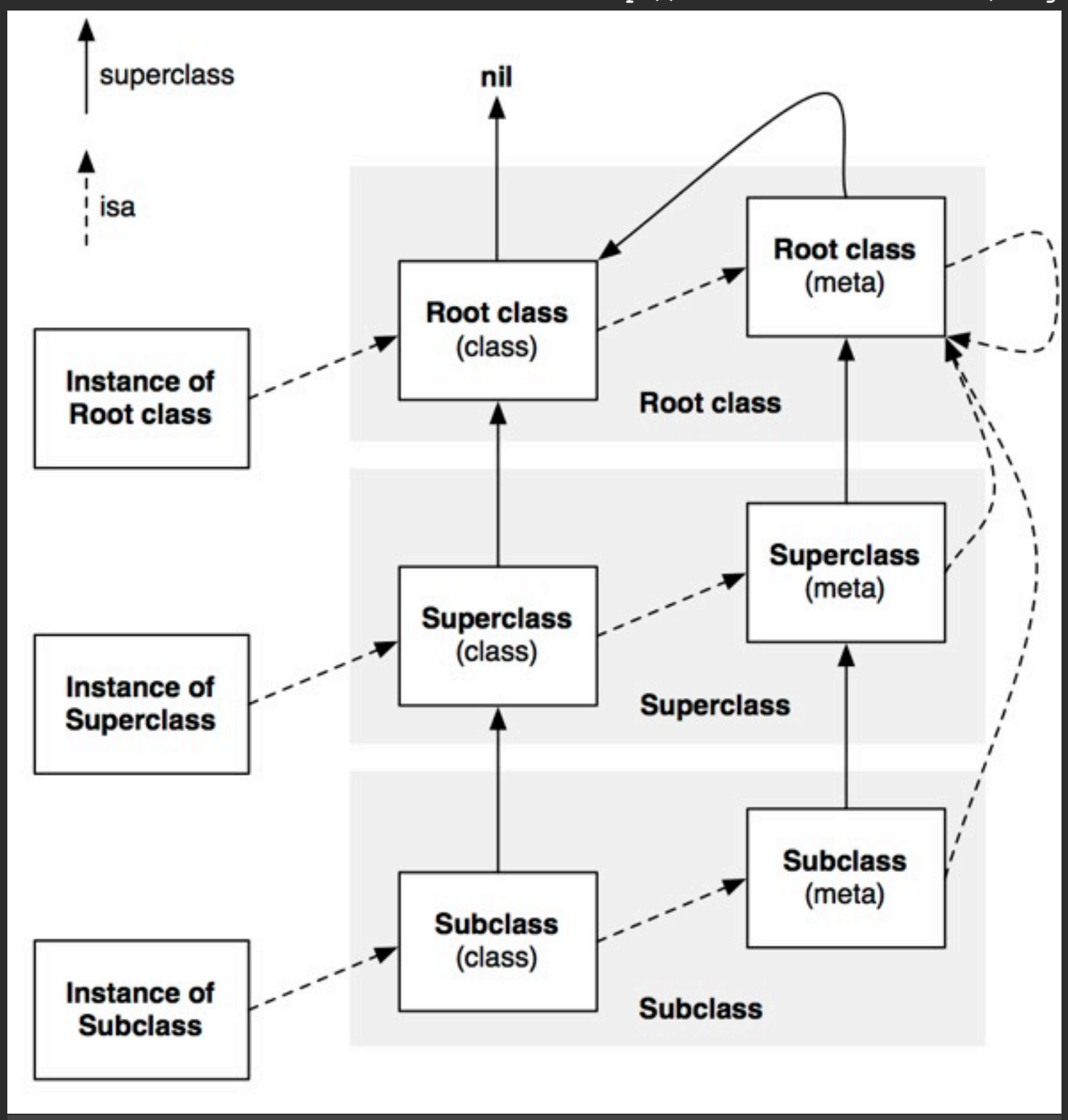
Depending on the receiver (super, other) and the value returned (simple, struct, float), there are several versions of this function.

This function call is usually generated by the compiler. If you write it by hand, you have to choose the correct version:

- objc_msgSend. Used when a simple value is returned.
- objc_msgSend_fpret. Only used when a float is returned and the processor is Intel.
- objc_msgSend_stret. Used when a struct is returned.
- objc_msgSendSuper. Used when a simple value is returned and the message is sent to the superclass.
- objc_msgSendSuper_stret. Used when super returns a struct and the message is sent to the superclass.

Apple´s implementation is in objc4-532/runtime/Messengers.subproj/objc-msg-x86_64.s It´s written in assembly.

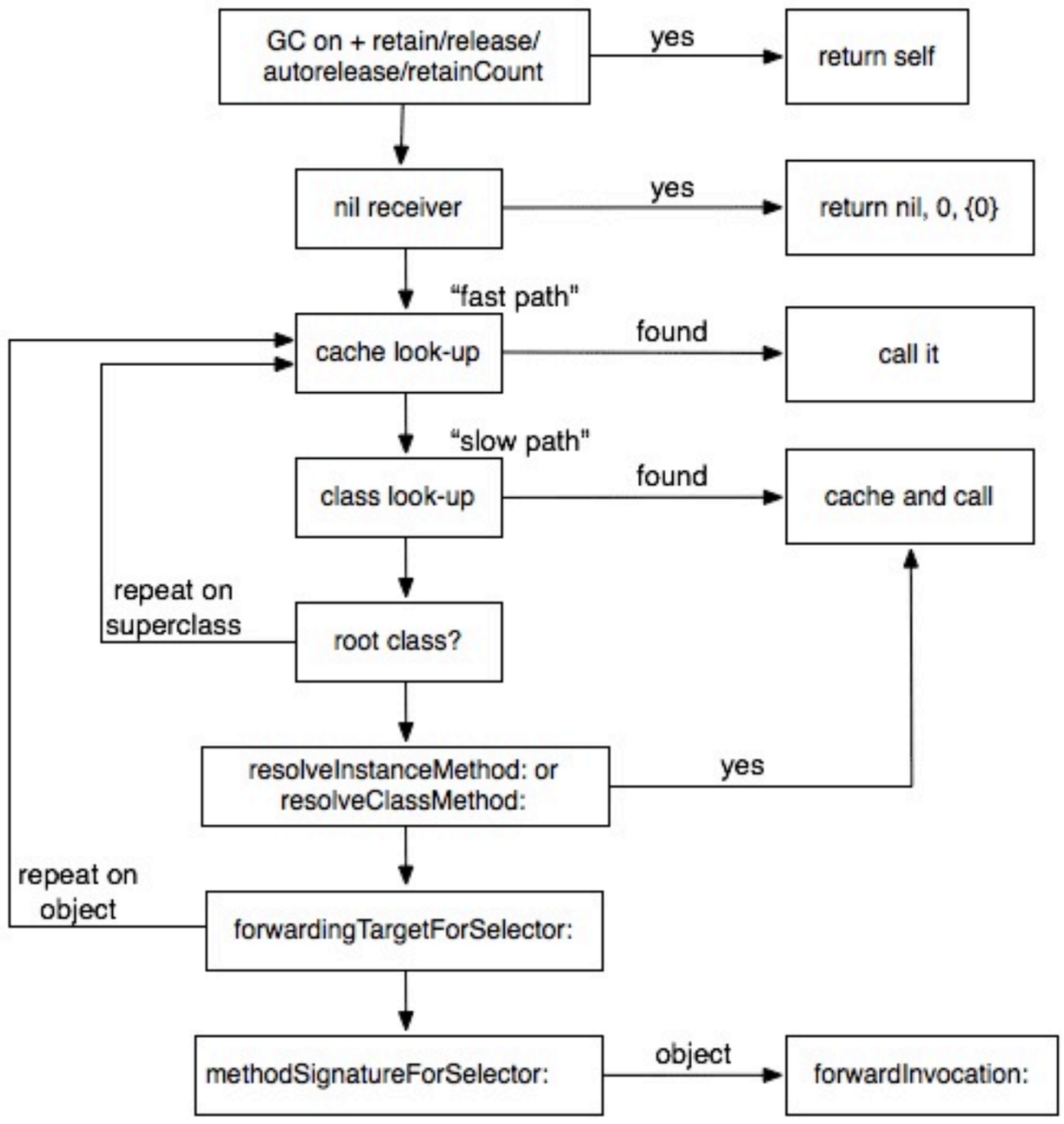
http://www.sealiesoftware.com/blog



Each instance has a pointer to its class (the isa pointer).
Each class has pointers to its superclass and metaclass.
Each metaclass has a pointer to its meta superclass.

The class structure is the same for class and meta class, but some pointers are unused:

- A metaclass doesn´t have a meta meta class so the pointer points to the root meta class.
- A root meta class doesn´t have a meta superclass so the pointer points to its class.
- A root class doesn´t have a superclass so the pointer points to nil.



If GC is on, and the selector is retain, release, autorelease, or retainCount, ignore it and just return self.

If the receiver is nil return nil or 0 or 0.0 or {0} depending on the return type. See 1, 2. Why not throw an exception instead? it makes the language more elegant. Consider [[[somePerson office] telephone] lastNumberDialed] as the same with ifs on each step.

If the selector is in the cache, call it. This is called the "fast path". It is implemented in the function IMP _class_lookupMethodAndLoadCache(Class cls, SEL sel) of objc-class.m. There is a cache per class. The cache contains all the methods of the class and its superclasses. Caches are filled as the methods are called.

Note: The objc_sendMsg call always uses tail call elimination so it doesn't show in the stack if you pause execution. This implies that you can't know the caller in an optimized program.

Note: Every time the runtime is looking for a selector, it uses pointer comparison, not string comparison. To accomplish this, all selectors of a process are uniqued. When you install OS X and you see at the end optimizing system, it is "uniquing" the framework selectors. That the selector is unique also produces a fast and stable (in time) search.

Note: A selector is the same for all methods that have the same name and parameters — regardless of which objects define them, whether those objects are related in the class hierarchy, or actually have nothing to do with each other.

If the selector is in the class, add it to the cache and call it. This is called the "slow path". The slow path happens only the first time we call this selector. However, a swizzle or category load will flush the cache of the class.

If everything fails, follow the pointer to the superclass and repeat.

If you reach NSObject, the runtime calls the following methods:

- Call resolveInstanceMethod: or resolveClassMethod:. If they return YES, go back to the slow path step.
  (these methods provide the chance to add a method dynamically using runtime functions)

- Call forwardingTargetForSelector:. If it returns an object, repeat on the given object.
  (this method provides a chance to mimic multiple inheritance).

- Call methodSignatureForSelector:. If it returns a method signature, wrap it on a NSInvocation and call forwardInvocation:. The forwardInvocation: inherited from NSObject calls doesNotRecognizeSelector:, which throws an exception.

Virtual table

objc_msgSend_vtable0	allocWithZone:
objc_msgSend_vtable1	alloc
objc_msgSend_vtable2	class
objc_msgSend_vtable3	self
objc_msgSend_vtable4	isKindOfClass:
objc_msgSend_vtable5	respondsToSelector:
objc_msgSend_vtable6	isFlipped
objc_msgSend_vtable7	length
objc_msgSend_vtable8	objectForKey:
objc_msgSend_vtable9	count
objc_msgSend_vtable10	objectAtIndex:
objc_msgSend_vtable11	isEqualToString:
objc_msgSend_vtable12	isEqual:
objc_msgSend_vtable13	retain (non-GC) hash (GC)
objc_msgSend_vtable14	release (non-GC) addObject: (GC)
objc_msgSend_vtable15	autorelease (non-GC) countByEnumeratingWithState:objects:count: (GC)

http://www.sealiesoftware.com/blog

In C there is no indirection (aka no “virtual call”), so it’s always the fastest.

C++ uses a “virtual table” per class, which is an array based structure. This structure is initialized in the constructor and doesn’t change during execution. There are no standards on the internal implementation of virtual tables, but all compilers do it that way. Because C++ has multiple inheritance there may be more one virtual table per base class.
The C++ compiler may create non virtual calls for some cases.

Objective-C uses a hash table to store pairs of selector/IMP. But once a method is resolved an entry is added to a per class cache, which makes the call only slighter slower than a C++ or C call. This is because Objective-C has duck typing so there isn’t a known set of methods in advance to create an array with, and also because we can alter the caches loading categories or using the runtime. So instead an array, duck typed languages use hash tables with string keys.

Because the keys have to be unique, the selectors have to be unique for each class, and there can’t be polymorphism.

For OS X frameworks, caches are created in advance. It’s the “optimizing the system” at the end of OS X installation.

Objective-C 2.1 (64-bit Mac OS X 10.6+) keeps a "virtual table" with 16 selectors most used everywhere and rarely overridden. These selectors make for 30%-50% of the selector calls in any application. Calls to those selectors are replaced at runtime with objc_msgSend_vtable functions. There are 16 functions.

Because Objective-C methods are C functions underneath, it's also possible to skip the messaging system and call the implementation directly, but it's an unusual optimization that saves little compared to a cached method call.

Calling the implementation directly

```objc
#import <objc/runtime.h>

@interface Person
@end

@implementation Person
+(id)new {
    Class cls = objc_getClass("Person");
    id obj = class_createInstance(cls, class_getInstanceSize(cls));
    return obj;
}
@end
```

```objc
int main(int argc, char *argv[]){
    @autoreleasepool {

        // [Person new]
        Person *person;
        SEL newSel = sel_registerName("new");
        Class personClass = objc_getClass("Person");
        Method method = class_getClassMethod(personClass, newSel);
        IMP newImp = method_getImplementation(method);
        id (*new)(id,SEL) = (id (*)(id,SEL)) newImp;
        person = new(personClass,newSel);

    }
}
```

Calling a method means running the C function behind every Objective-C method, and pass the arguments plus 2 more: self (the receiver) and _cmd (the method). Both _self and _cmd are available to use on every method.

In Objective-C you can also get the address of a method and call it directly without objc_msgSend.

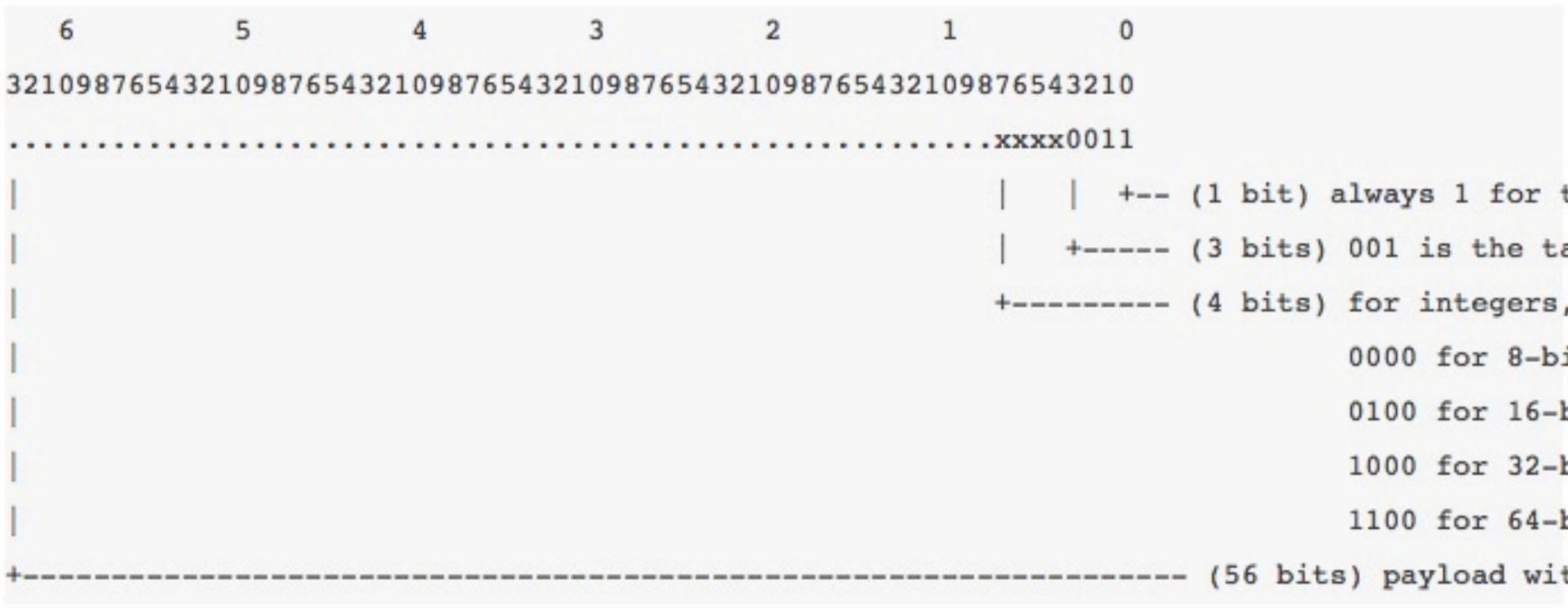
Once resolved, we could store the method implementation on a pointer, and use it to perform function calls skipping the dispatch table. This is an optimization we can apply to methods that are run many times. In fact, the compiler generates an equivalent function call.

This saves little because once resolved, methods are cached in the objc_cache struct. So only the cache look-up is skipped.

Note that the first arguments are id and SEL as told in the documentation for IMP.

Tagged Pointers

tagged Pointers



objectivistc.tumblr.com/post/7872364181/tagged-pointers-and-fast-pathed-cfnumber-integers-in

A x86 64 pointer has 64 bits, but only uses 48 bits for the pointer address. The reason is that 48 bit provides a 256 TB address space, which is a lot. Using 64 bits would be wasteful because it would require more transistors in the CPU. So the potential address space is 64bit, but current CPUs are only able to use 48.
Because of this, the ending bits of a pointer are 0 because they are left unused.

Tagged pointers are pointers which contain data instead an address. Example, let's say we want to store the integer 42 in a pointer. We could create a NSNumber and then point to it, or we could replace the address with a 42, in which case we can skip object creation.

To distinguish between a normal pointer to an object and a tagged pointer, we can use the bits at the end.
OS X Lion has tagged pointers, and uses the last bits as seen in the image.

The tagged object class is integer, managed object, or date. The next 4 bits are for information about the object class.
To read the value of the tagged pointer, the value is shifted 8 bits to discard the meta information on the image.
To know what the class is for each tagged pointer, there is a table in the runtime that maps the 4 bits to a Class. This table is used by object_getClass(), which means that we shouldn't access the isa pointer directly for NSNumbers, because it may not work if it is a tagged pointer.

Not every NSNumber can be turned into a tagged pointer. In those cases, the real object is created.

Tagged pointers save memory and increase performance.
Tagged pointers can be turned off with a compiler switch

--Talk about NSConstantString and compile time constants--

oh hi, a number literal: http://jens.ayton.se/blag/objc-constant-objects/

--SKIP THIS IF THERE IS NO TIME--

In a 64 bit architecture, **memory access happens in chunks** of 8 bytes (64 bits), and each memory access starts at an address which is a multiple of that size.

Example: let's say we want to read a 64 bit pointer which is not aligned, that is, half of it is in one chunk and the other half in the next. We will need two read operations and another to put the two halves together. Another problem is that the read is not atomic, so we could be reading invalid state.
This is just an example to illustrate that **data alignment is required for performance and atomic operations**. Most likely the CPU will throw an exception if we attempt to do such a thing.

The alignment depends on the size of the data. For example, a char is 1 byte, so it will be 1 byte aligned. In a 64 bit architecture we would be able to store 8 chars in one chunk (8*8 = 64).

In reality, x86-32 ABI and **x86-64 ABI require 16 byte alignment**. Example: let's say we need to allocate memory and we use malloc. Because malloc doesn't know what we are going to store, it assumes the worst case scenario, which is we are going to use Intel SIMD instructions which require 16 byte alignment.

Toll free bridge

At one point Apple was developing Carbon (a C API) and Cocoa (a pure Objective-C API).
Core Foundation was a C API common to both of them.

Toll Free Bridging is a mechanism that makes some Core Foundation classes interchangeable with its Cocoa counterparts.

Example, we can cast between NSString and CFStringRef as if they were the same class. Actually NSString is a “class cluster”, which means that the visible class (NSString) is an abstract class, abstract in the sense that you never get a NSString instance but a NSString subclass (even when you call it NSString).

How it works?
- Some CF/NS objects have the same memory layout. In this case NS methods use the CF implementation.
- Some CF/NS objects wrap each other methods in C and objective-c. That is, their code is split between the CF and NS counterparts. This is good because you get advantages from both the static nature of C and the dynamic nature of Objective-C. Every function that works with TFB objects need to check if the object is one kind or another. When the object is NS, the class pointer has an special value.

Toll free bridge exists, but programmers shouldn’t rely on it.

Blocks

Blocks

http://www.opensource.apple.com/source/libclosure/

```
struct Block_literal_1 {
    void *isa; // &_NSConcreteStackBlock or &_NSConcreteGlobalBlock
    int flags;
    int reserved;

    // reference to the C function that implements this block
    void (*invoke)(void *, ...);

    struct Block_descriptor_1 {
        unsigned long int reserved;     // NULL
        unsigned long int size;         // sizeof(struct Block_literal_1)

        // optional helper functions
        void (*copy_helper)(void *dst, void *src);     // IFF (1<<25)
        void (*dispose_helper)(void *src);             // IFF (1<<25)

        // required ABI.2010.3.16
        const char *signature;                         // IFF (1<<30)
    } *descriptor;
    // ... imported variables
};
```

The Block Implementation Specification (http://clang.llvm.org/docs/Block-ABI-Apple.txt) describes how blocks are implemented at the runtime level.
The project is called libclosure, and it is open source: http://www.opensource.apple.com/source/libclosure/
To download tarballs: http://www.opensource.apple.com/release/mac-os-x-1068/

There are a couple more documents in the root of the libclosure directory. One called “Language Specification for Blocks” and the other is an outdated version of the Block-ABI.

As an overview, here is a block definition.

The **isa pointer** to an object. If the block doesn’t capture variables the block is NSConcreteGlobalBlock (or NSGlobalBlock for Core Foundation), which is, I think, a static implementation that doesn’t require memory management (which is implemented with copy_helper and dispose_helper functions defined below).
Otherwise the block is created in the stack which means it doesn’t survive the scope in which it was declared. That scope is defined by braces ({}). It can be a method, or the braces of a loop inside the method or whatever. This is an optimization that assumes that the block won’t be needed outside its definition scope. If this is not the case, we have to “copy” the block, which puts the block in the heap.

The **flags** indicate which kind of block it is.

reserved is used internally as a retain counter (I think). This flag may be always 1 for a global block, or change if it is a stack block and we “copy” the block. To indicate we shouldn’t rely on this field, the method “retainCount” always returns 1 for a block.

The **invoke** function calls the block implementation, which is a C function. First argument is this struct, and the next are the arguments for the block.

The **descriptor** has the size of the block, a reserved field which is maybe unused, I don’t know and the docs don’t explain either. Functions for memory management. The signature is (I think) the version of the Application Binary Interface.

The imported variables at the bottom are the variables captured by the block. This variables are a “const” copy made on the stack. If they are primitives they become immutable, and if they are objects, the pointer is immutable but the pointed object is not.
If you copy the block, the variables are copied (malloc’ed) in the heap. You can actually see the change of address for the variable if you print it.
Calling “retain” on a block doesn’t make a copy on the heap, only “copy” does.


Game over man! that was my last slide