Thomas Bouldin
@inlined

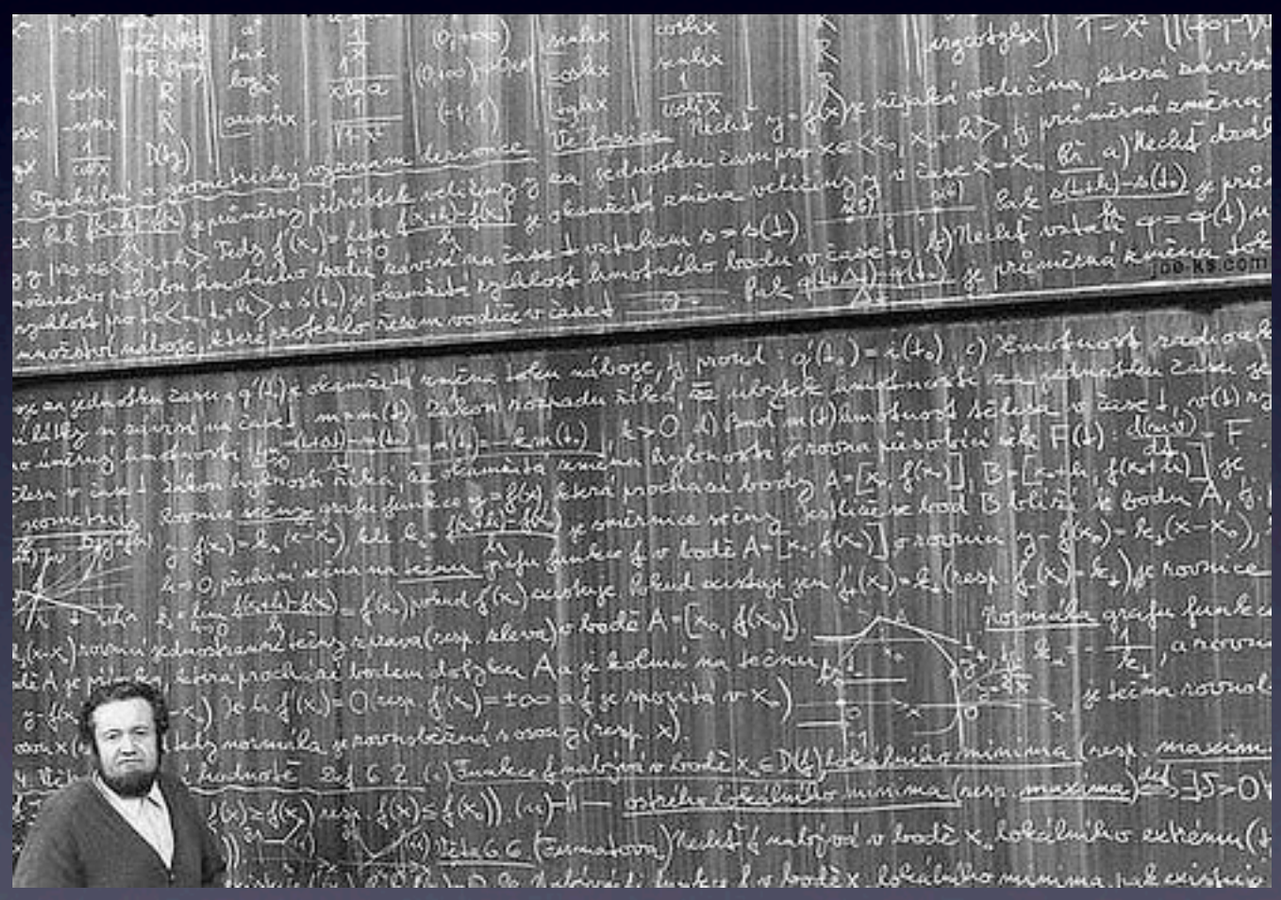
Objective-C: Under the Hood

Who am I? I've often called myself a systems programmer, which is something incredibly hard to explain to non-technical friends & family I tend to write software that you tend to only focus on when it's broken.

I've loved Objective-C since college (way in the days before iOS *gasp*), but only started using it professionally at Parse. I have had to understand the way languages and frameworks dispatch messages, however with my time at Microsoft. This was most visibly important when I wrote the original prototypes for and helped design the architecture of the Windows Web Application runtime host. You have to understand the relationships between objects & classes and how messages are dispatched to work at that level of the system.

Languages generally fall into a few categories, and industry best practices tend to guarantee similar implementations. This is especially true with the C-family of languages, which tend to build upon each-other by standardizing conventions first made possible in C.

Before we Start



This talk should be considered merely academic. It's useful to learn how your car works. Sometimes you can even benefit from knowing how to change your own oil, but very few people benefit from making their own catalytic converters. There's a lot of complexity here and I'm glossing over details I couldn't cover or worry I might misrepresent since I don't contribute to this project.

Itinerary:

1. Decide what we need in order to agree something is OO if we squint really hard
2. Demystify memory access and bootstrap some OO conventions with "jailbroken" C
3. Understand a simpler OO system via primitive C++ (primitive = still applies to Obj-C)
4. Learn the primitives and conventions that bootstrap Obj-C from the primitives in (3)

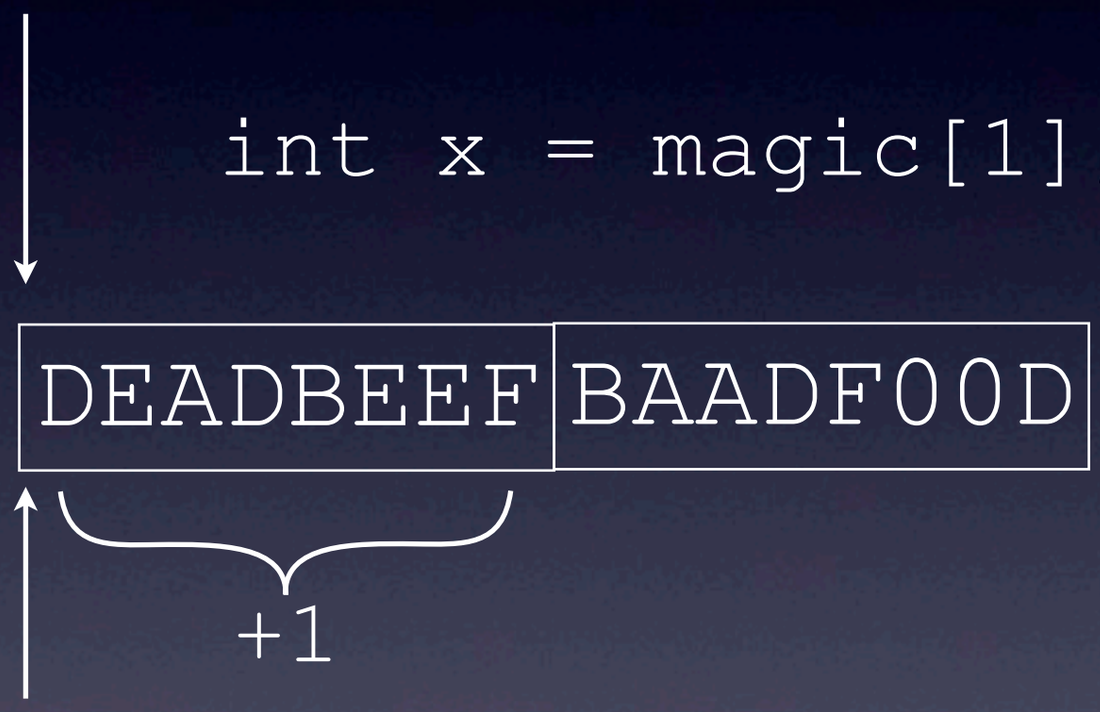
OOP

- Objective-C: The *other* "C with Classes"
- What is Object Oriented Programming?

For those unfamiliar, "C with Classes" was the original name for the original C++ draft. In many ways, C++ (sans templates) is a simpler system than Obj-C, and it's easier to bootstrap concepts with the C -> C++ transition. Objective-C is written in C++ anyway, so it's not really cheating.
For those reading, the three fundamental concepts of OOP (according to me) are:
1. Encapsulation of concerns (let's just call that a struct. It gathers related information. Trying to enforce private access is a compiler concern as much as a language one, and I personally think it's stupid)
2. Extension of implementations. Even without dynamic behavior, extension is important. All Obj-C objects are reference counting and that's really important even without...
3. Polymorphism. Delegating implementations to concrete subclasses just makes our lives easier

It's All Just Memory

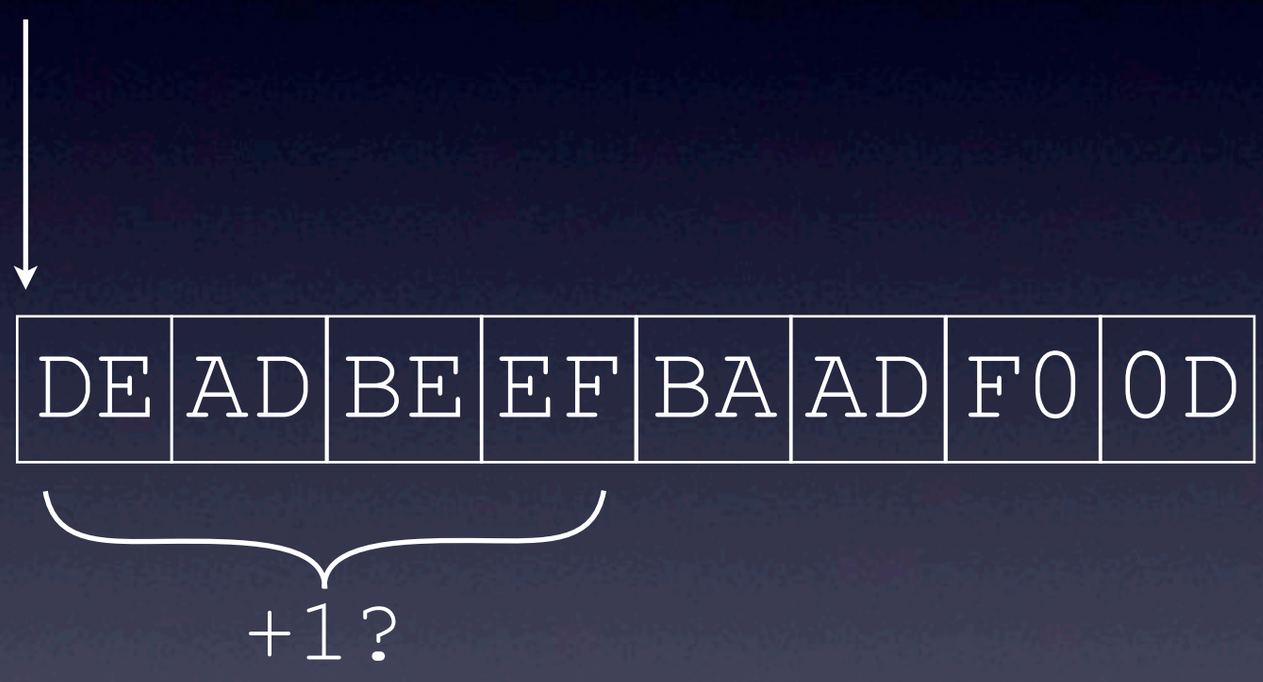
```
int magic[] = {0xDEADBEEF, 0xBAADF00D};
```



This is a visualization of how an array appears in memory and what it means to access the nth element. It's really a pointer + n from the array (when using typed pointers).

It's All Just Memory

```
void *scary_ptr = magic;
```




```
x = *(scary_ptr + 1)
```

When using void pointers, we no longer have type information. Our translation from array[n] to *(pointer + n) doesn't work

It's All Just Memory

```
void *scary_ptr = magic;
```


```
x = *(scary_ptr + 1 * sizeof(int))
```

We need to get back the benefit of type information. We know we expected an int and we know the size of an int.

<void pointer> + <index> * <element size> is an idiomatic way to access an element from an array-like thing

It's All Just Memory

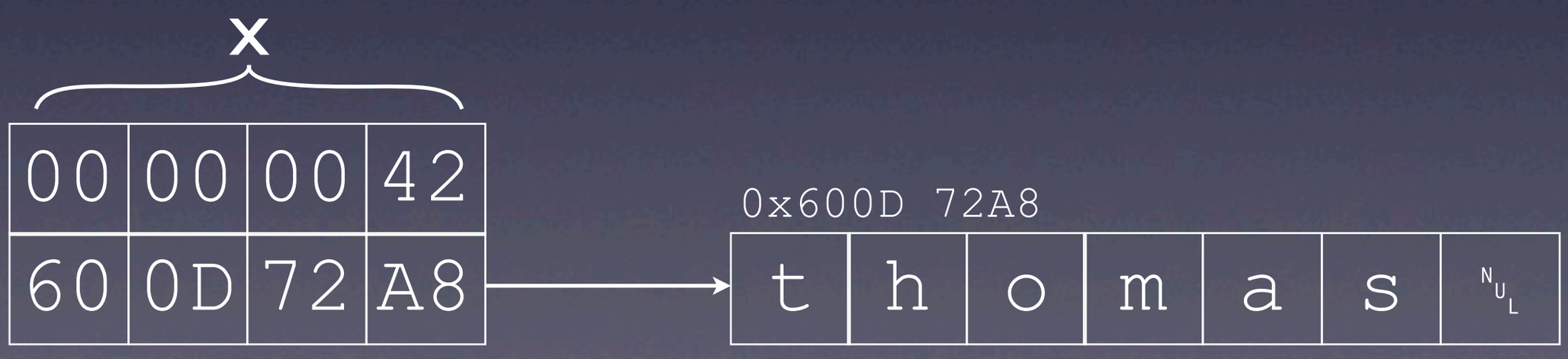
```
void *scary_ptr = magic;
```


```
x = *(int *)(scary_ptr + 1 * sizeof(int))
```

Of course we have to cast it back to something other than a void* to dereference it. This is the full way to access element 1, 0xBAADF00D, from a void * array.

It's All Just Memory

```
typedef struct {
  int id;       }x
  char *name;
} User;

User *thomas; // {42, "thomas"}
char *name = thomas->name;
```

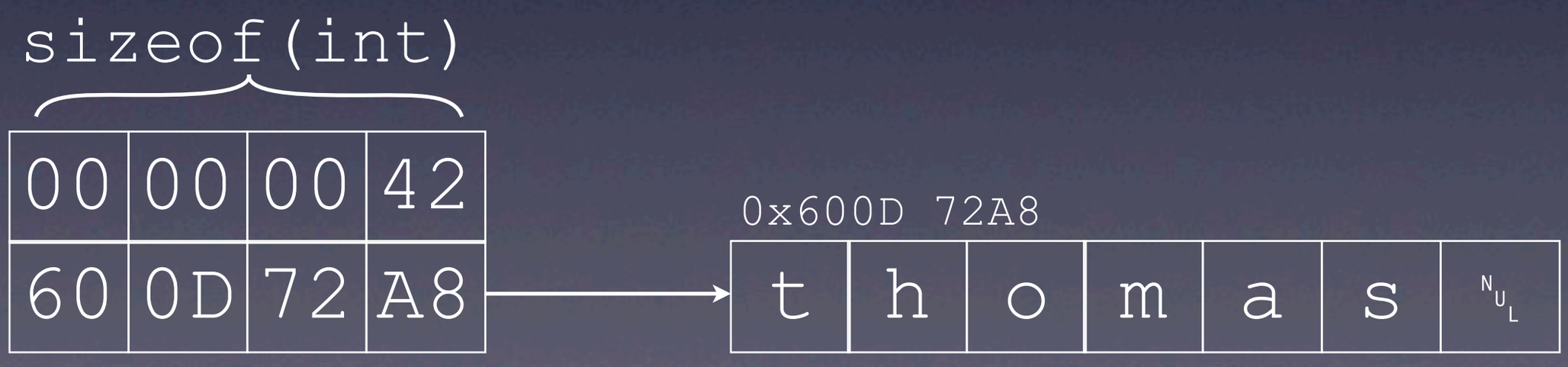


Structs are also contiguous blocks of memory. When I ask for thomas->name, what happens?
I have to start at thomas and skip past X. What's X?
The compiler determined this from the layout of the struct.

It's All Just Memory

```
typedef struct {
  int id;          }sizeof(int)
  char *name;
} User;
```

```
User *thomas; // {42, "thomas"}
char *name = thomas->name;
```



There is only one int between the start of the User struct and the name member

It's All Just Memory

```
typedef struct {
  int id;
  char *name;
} User;
```

```
char *name = (char *)
             ((void *)thomas +
              sizeof(int));
```

00	00	00	42
60	0D	72	A8

Nobody should ever do this, even to impress their friends.
It's good to know that this is what the compiler does, however, because we know that accessing the Nth member of a struct only depends on the layout of the previous members.

It's All Just Memory

```
typedef struct {
  int id;
  char *name;
  int created_at;
} Member;
char *name = (char *)
             ((void *)thomas +
              sizeof(int));
```

00	00	00	42
60	0D	72	A8

It's very important to understand this concept, because any function that expects a User pointer, such as printUser will also work with a Member pointer because member starts the same and the math works for either.

Extensions

- C-level extensions: It has the same prefix
- This is how the BSD socket library works
- Could call the struct* parameter “this” or “self”
- C++ does, but hides it from you
- Toll-free bridging of class clusters

This is how C++ non-virtual methods are implemented. They create a static C function with a hidden class * this parameter.
This is how toll-free bridging works: subclasses have the same prefix as their parent, so one CoreFoundation method works on an entire class hierarchy

Should possibly note that it is NOT OK to ever (de)serialize a dereferenced pointer to a struct that might be a different type. This is causes the “slicing problem”. Array of struct is already a struct *, so an array of struct ‘subclasses’ MUST be a struct **

Polymorphism

```
// Dog & Parakeet share Animal prefix
struct Animal, Dog, Parakeet;

// Please don't git blame this
void speak(Animal *animal) {
  if (Animal->kind == DOG) {
    puts("Woof!");
  } else if (Animal->kind == PARAKEET) {
    puts("Tweet!");
  } else {
    assert(false); // Can't happen
  }
}
```

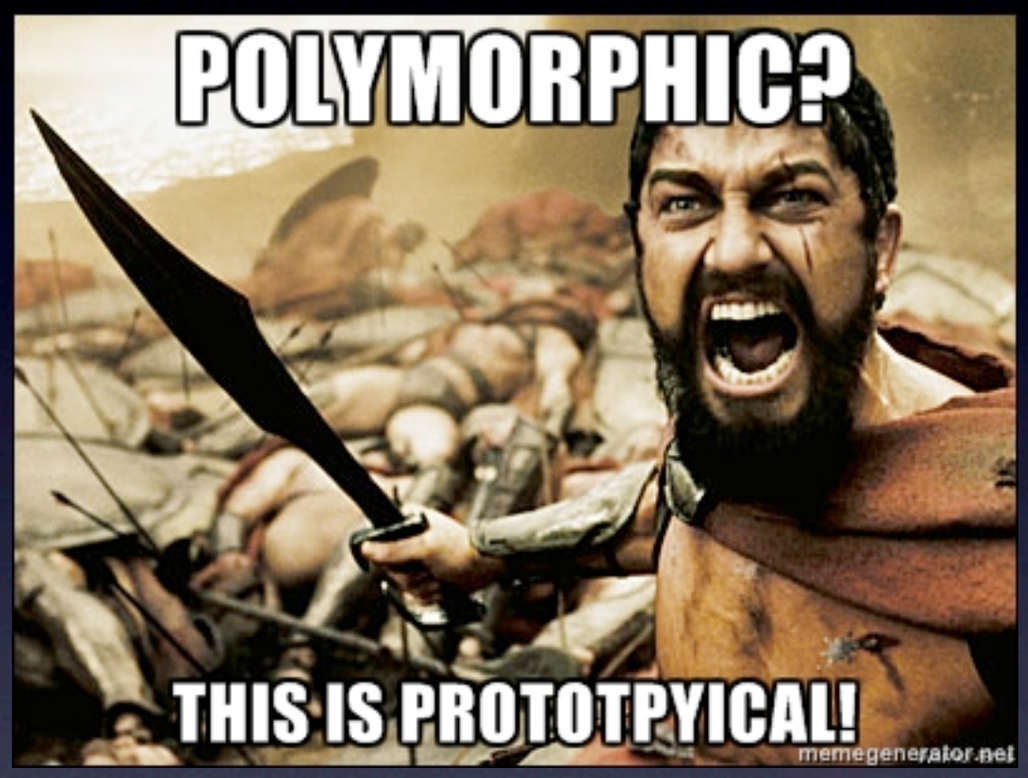
Putting dynamic behavior in a parent class sucks. You don't want this to be your legacy.

Polymorphism

```
typedef struct {
  void (*speak)(Animal *);
} Animal;

void Dog_speak(Animal *) {
  puts(“Woof”);
}

void Parakeet_speak(Animal *self) {
  puts(“%s wants a cracker”, self);
}

Dog hearshey = {&Dog_speak};
Parakeet jello = {&Parakeet_speak};
```

If the struct includes its behavior, we can concentrate on each extension’s behavior individually & extensions can behave dynamically.

Polymorphism?



- We cannot assume two objects of the same type behave the same
- Indirection can solve (or create) any problem

This isn't really polymorphism, because it's possible to initialize a Dog with Parakeet_speak

Polymorphism!

```
typedef struct {
  void (*speak)(Animal *);
} Animal_klass;

typedef struct {
  Animal_klass *isa;
} Animal;

extern Animal_klass
    _dog, _parakeet;
```

We can ensure that all Dogs and all Parakeets behave the same by encapsulating *all* their behavior in a struct and using a singleton behavior in one pointer. Now something IS a dog not because of its struct--it's all memory after all and structs go away after compile time--but because of the isa pointer it has.

The trick we learned to extend a struct works here for both Animals and Animal_klasses. We can create a new Dog struct with additional state, and we can create a new Dog_klass state with additional behavior.

Polymorphism!

```
// in C++: animal->speak();
// or just speak();
animal->isa->speak(animal);


// Can your C++ do this?
if (animal->isa == _dog)


// What about isKindOfClass:?
```

NOTE: in C++, classes are immutable, so there is one vtable per class.
The set of methods are fixed, so each virtual method call is (this->isa + offset)(this, ...);
As an optimization, non-virtual methods don't need to be part of the isa.

Objective-C is more dynamic. You can add or remove methods, so tables cannot just use a fixed offset.
Objective-C uses a "dictionary" and can interrogate up the inheritance hierarchy

Build Objective-C

- @YES, @“Parse”
- @protocol ?
- @interface/implementation?
- @selector

We understand literal syntax (@YES, @1, @{@”foo”: @”bar”}) recursively. We know it sets up an object or is shorthand or some object constructor, but we don’t know what objects are yet. Can we figure out a protocol? It’s simpler, but dependent on what a class is.

The first foothold we can get is @selector

Objective-C Method

- SEL is a string you can ==
- IMP is a C func:
- typedef void (*IMP)(id self, SEL sel, ...);
- Method is a struct with an IMP, SEL, and char * typeinfo

Pedants are careful to not use the word “method” and “function” interchangeably. A function is global, a method is an OOP concept.
In C++, a method is (loosely) a function with a secret this pointer.

If SEL is just a char *, then why make a new typedef? And what the heck does sel_registerName do? or sel_getName?!

SEL is an interned string. That means there’s many possible copies of the text “retain”, but in an entire process, there is only one @selector(retain). sel_register/getName work with the singleton versions. This means that you can check for equality of a SEL in O(1) instead of O(N). Just as important, you don’t dereference the pointer, so you never have a cache miss. Avoiding cache misses often means an order-of-magnitude improvement, so SEL is roughly two orders of magnitude faster than O(N)! Crazy!

Objective-C Object

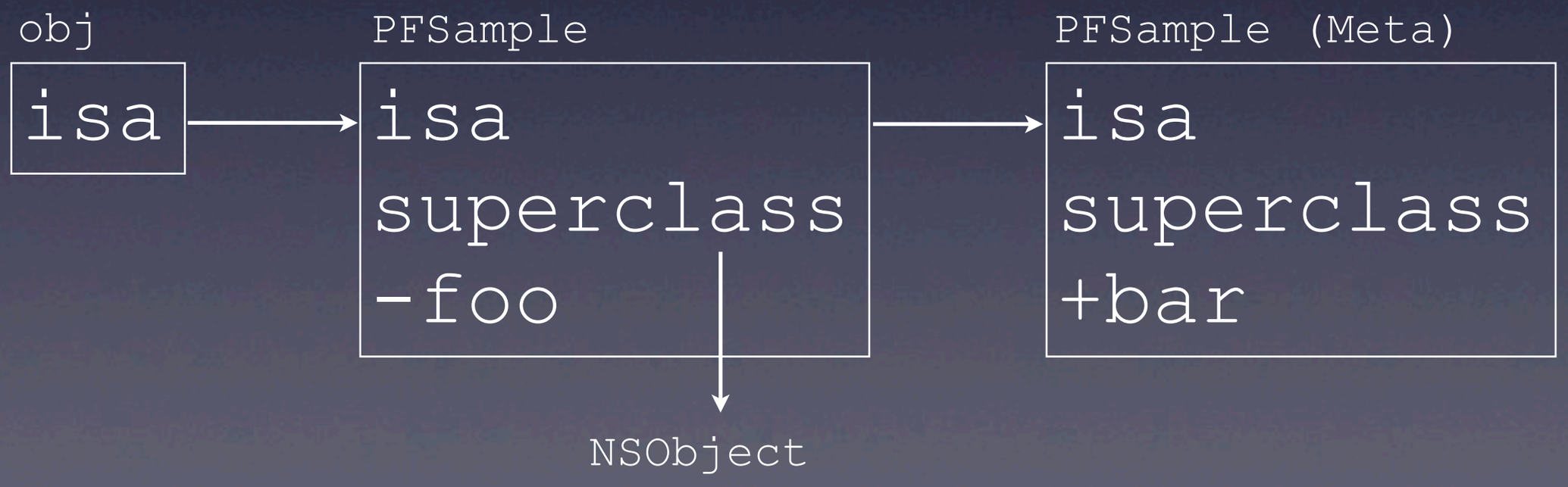
```
typedef struct objc_class *Class;
typedef struct objc_object {
    Class isa;
} *id;

// struct size can be dynamic
id object = (id)malloc(
     class->instance_size);
object->isa = class;
```

All objects are of type objc_object, but that doesn't matter. The "real" type is the isa pointer.

For those who care, objective-c has a _malloc_internal that worries about NSZones.

Objective-C Class

```
@interface PFSample : NSObject
- (void)foo;
+ (void)bar;
@end

id obj = [[PFSample alloc] init];
```



Classes are objects. You can tell, because they have an isa pointer! (So I guess I lied when I said all objects are from the objc_object struct. Sue me.)

Classes are objects and objects have classes. The class of a class is called a meta-class. There is no meta^2 class.

If a class' methods apply to an instance, a meta class' methods apply to a class. This ensures the same message resolution mechanics work on (non-class) objects and classes alike.

Omitted from this slide: A class' data is actually broken up into three different structs: objc_class, which contains a pointer to a class_rw_t, which contains a pointer to a class_ro_t

Sending Messages

```
[animal eat:food];

objc_msgSend(animal, “eat”, food);
```

```
-[PFKeet eat:](
      animal, “eat”, food);
```

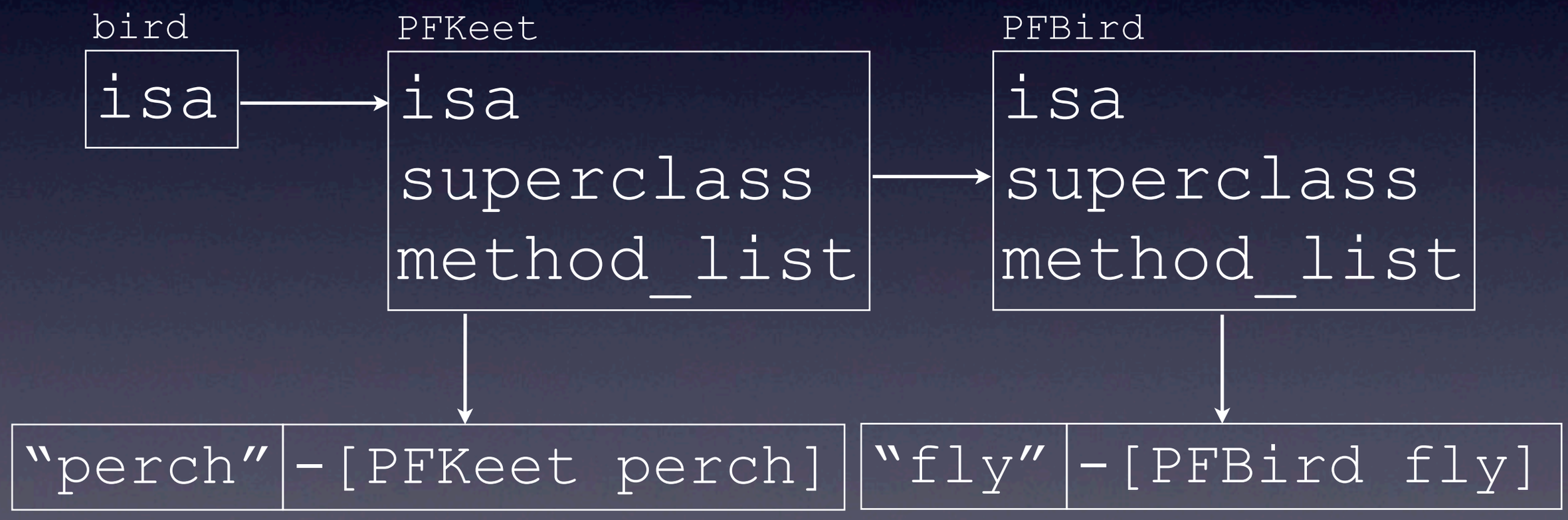
We’ve started unraveling the what the @ symbols do, what about the [] symbols?
The line [animal eat:food] sends the message (aka selector) eat to the animal object with the foo parameter. What does that mean?

Well first, objc_msgSend kicks off a Method lookup. (Notice that the method signature for objc_msgSend actually makes it an IMP too. More about that later)
Something happens, and profit! We suddenly have a new IMP that is the means for this particular animal to eat:

-[PFKeet :eat] is the name of a C function. In vanilla C, this would be an illegal name, so Objective-C uses this convention to avoid conflicts (the same way it uses @ symbols to introduce new keywords). Regardless, run `strings` on your binary if you don’t believe me, these are actual exported C functions.

Method Resolution

```
[bird fly];
```



What did the underpants gnomes do to find the correct IMP for that SEL?
Most of the work here is in look_up_method in objc-class.mm. Yes, that's .mm, as in Objective-C is written in Objective-C++. Did I just blow your mind?

follow the isa to PFKeet. PFKeet doesn't have a "fly", so go to superclass.
We find fly there, so we cache it in PFKeet. This is different from method_list. It's also faster than a linear traversal of method_list, so we would cache the method even if it were part of the original isa's method_list.

If we got all the way to NSObject (or another base class) and didn't find fly, we'd send [PFKeet resolveInstanceMethod:@selector(fly)] (assuming it responds to -resolveInstanceMethod). This is the way that dynamic methods are resolved.

If that doesn't work, we try forwardingTargetForSelector: This is fast because it still uses SEL and varargs. If that doesn't work, an NSInvocation is created (expensive) and passed to forwardInvocation: when method lookup fails and the runtime has to resort to message forwarding, the IMP forward:: is cached to avoid further lookup.

Method swizzling changes the key-value pair in this list. It also impacts all caching, so try to do it early before caches are built.

Et. al.

- Protocols
- Categories
- Properties
- Blocks
- ivars

a Protocol is an object with optional & required instance & class selectors. Classes have protocol_lists

Categories can be implemented as a bunch of IMPs and a static method that calls class_addInstanceMethod

Properties are just metadata on a Class. It's the synthesizer that does anything, and that's all compile-time.

Blocks are objects, so they have an isa. They copy every referenced member into the block with two caveats: referencing an ivar of the calling class captures self instead and __block captures actually capture a pointer to that variable. The block then has a function pointer as part of its definition which is like an IMP but without the SEL parameter

As of Objective-C 2.0, ivars are more dynamic. The class declares how much memory is needed to store all the ivars, and this is determined at runtime. There is a very compact binary structure in the class that says where to do a void * offset into the id to access an ivar. This is very compact and cacheable, so it's not a noticeable performance hit, and it solves the "fragile base class" problem. Apple can add more ivars to their base classes and your subclass will automatically adjust without recompilation.

VTable Optimization

- New to Objective C 2.0!
- Fixed set of IMPs, determined at runtime
- Classes can share with super
- Default IMP is objc_msgSend

“Determined at runtime” means “pick between the GC and non-GC list.”

The squashed vtable for the top 15 frequently called but seldomly modified methods, such as alloc, objectAtIndex:, objectForKey:

Swizzling, adding methods, etc. require a recreation of a vtable for a class & all its subclasses if the affected method is in the vtable. vtables get less churn by being picked wisely and by only getting created after +initialize. Before +initialize, the vtable is full of objc_msgSend. This means +initialize is the perfect spot to message swizzle because it won’t require vtable recomputation.

Wrap Up

```
PFUser *user = [[PFUser alloc] init];
user.username = @”thomas”;
user.password = @”NSMeetup”;
[user signUpInBackground:
   ^(BOOL success){
        if (success) { NSLog(@”Yey!”); }
   }];
```

Can you tell what the runtime is doing here?

PFUser is an object whose isa is a metaclass. alloc is in the vtable for NSObject; our vtable is the same as NSObject’s vtable & we skip lookup for PFUser & PFObject.

init is not in the vtable. Call objc_messageSend. We find the c function -[PFUser init] in the PFUser class’ Method list. Cache it for later.

username & password are dynamically synthesized. objc_messageSend calls class_getImplementation which fails lookup. It calls obc_msgSend(user, @selector(resolveInstanceMethod:), @selector(setUsername), which adds the Method to the PFUser Method list. Lookup is retried & succeeds. The new implementations are cached.

The block is an anonymous class. It has an isa pointer (blocks are objects!) a C-function that looks like C++ (void (*)(id self, ...)) and the rest of the block’s struct are captured variables.

If there’s sufficient time, explain objc_retainAutoreleasedReturnValue and objc_autoreleaseReturnValue

Questions?

Annotated slide deck available at
http://inlined.me/objc.pdf

Follow me at @inlined

Contact me at thomas@parse.com

Want to build the platform devs love?
Send a POST to https://parse.com/jobs/apply

This slide deck was re-published with permission on the Parse blog. Please get my permission before hosting my work elsewhere.

Details for Parse's job application API are available at https://parse.com/about/jobs#api
You may also apply via more traditional means at https://parse.com/about/jobs