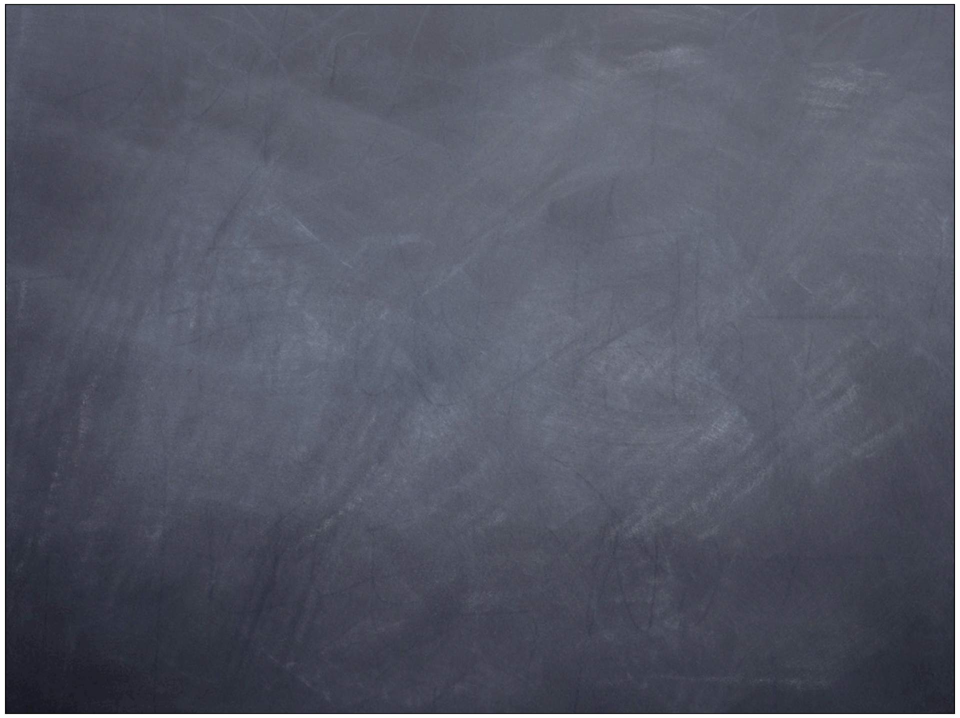
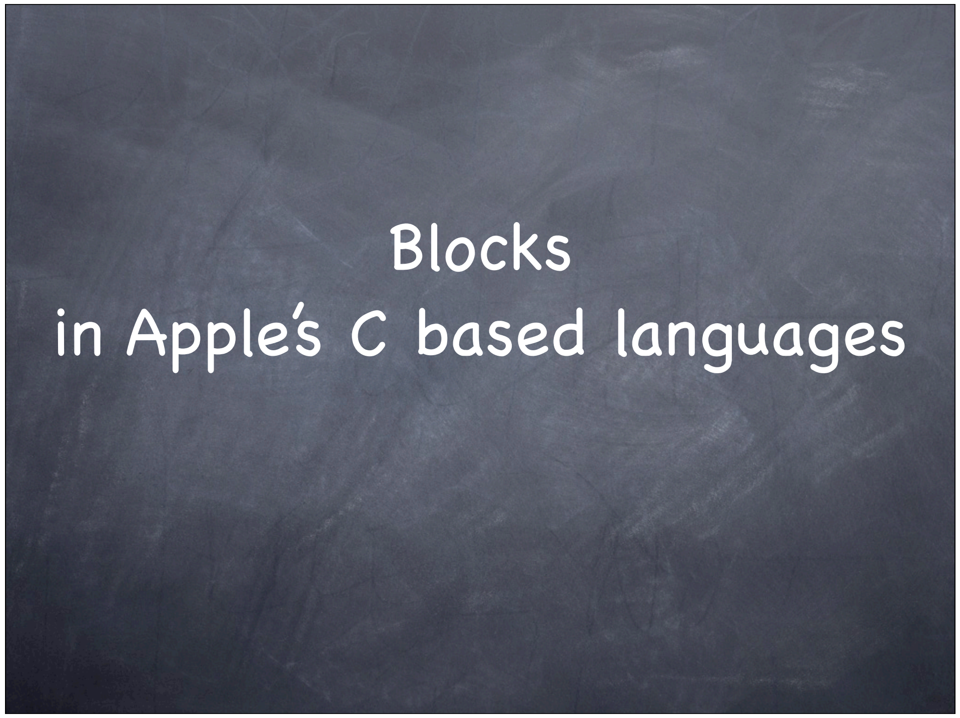
Blocks
in Apple's C based languages

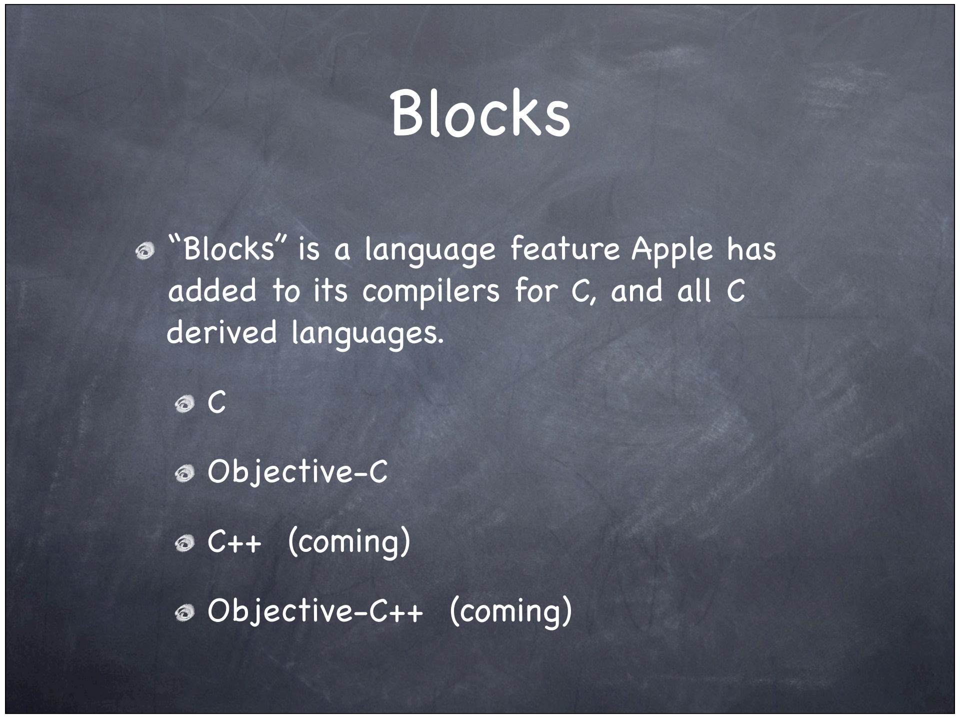




Blocks
in Apple’s C based languages

Assuming basic knowledge of at least one of the C based languages, and a little familiarity with Objective-C.

I’m not going to cover all the dark corners, but I’ll hint where some of them are, and suggest that you might want to stay away from them.



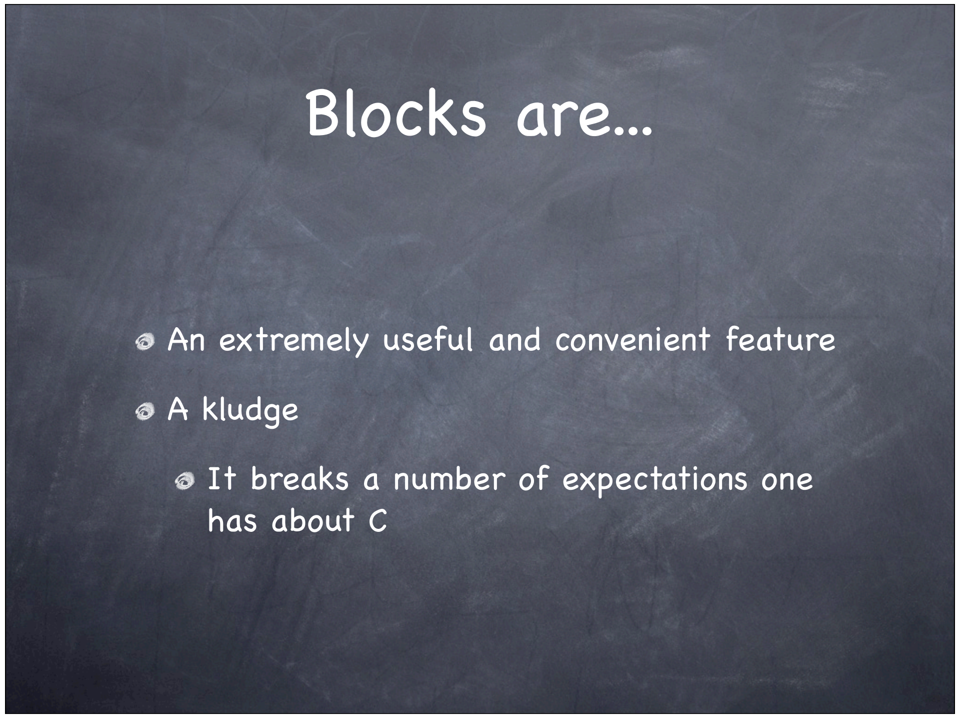
Blocks

- "Blocks" is a language feature Apple has added to its compilers for C, and all C derived languages.
  - C
  - Objective-C
  - C++ (coming)
  - Objective-C++ (coming)

1: Forgive me if I mess up my plural/singular distinctions.
4: I'll won't really touch on blocks in C++
5: or Objective-C++

Blocks

- Apple has submitted blocks to the standards committee for C.
  - Editorial break

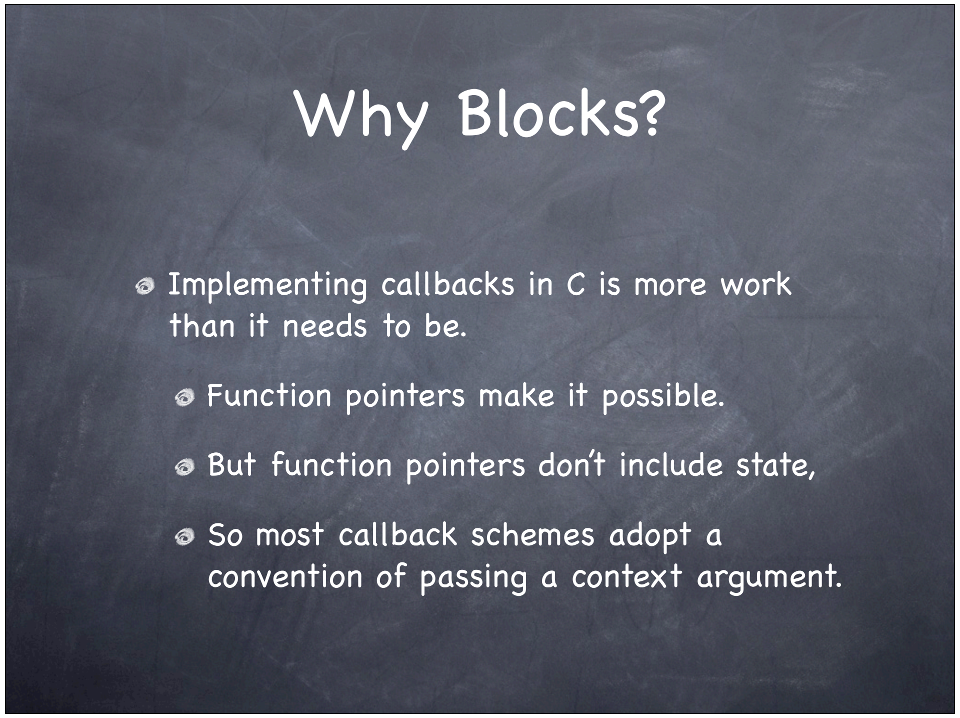


Blocks are...

- An extremely useful and convenient feature
- A kludge
  - It breaks a number of expectations one has about C

3: I'll point some of these out as we go along, but one overall observation is that C is traditionally a language without a big runtime. C statements map fairly obviously into the instructions required to implement them. That's definitely **not** the case with blocks. My completely uninformed opinion is that blocks are not likely to make their way into standard C, at least not without changes. That's **one** of the reasons I urge caution in using certain features of blocks. Apple would probably maintain compatibility even in the face of a changed standard, but I intend to be cautious.

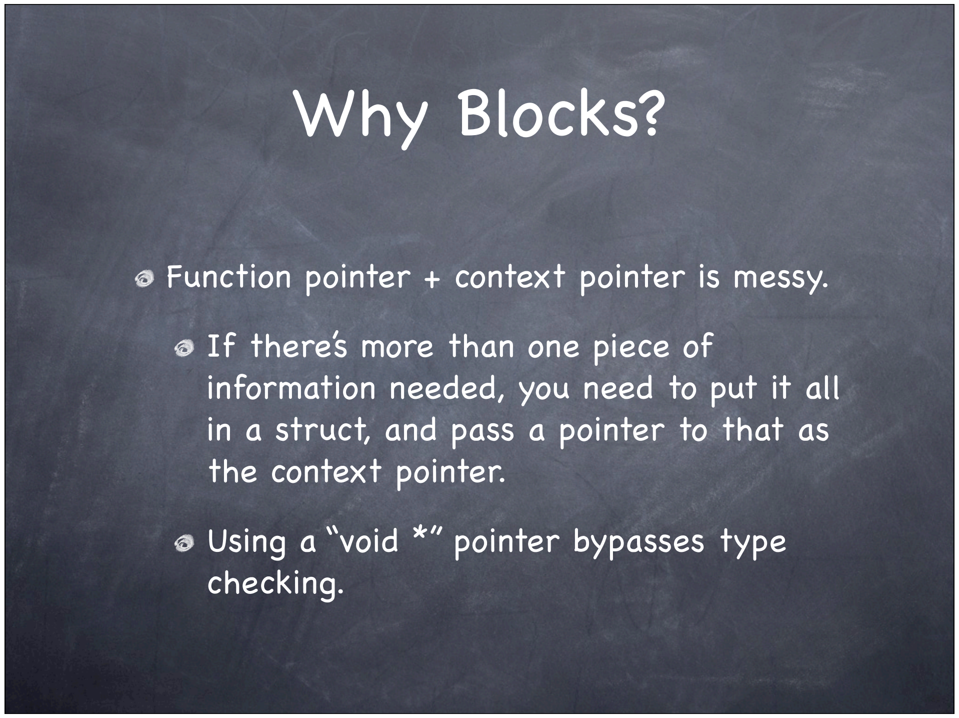
Objective-C, of course, **does** have a large runtime, so this observation isn't as relevant. But try looking at the assembly code for blocks (I did!). It's almost entirely unintelligible.



Why Blocks?

- Implementing callbacks in C is more work than it needs to be.
  - Function pointers make it possible.
  - But function pointers don't include state,
  - So most callback schemes adopt a convention of passing a context argument.

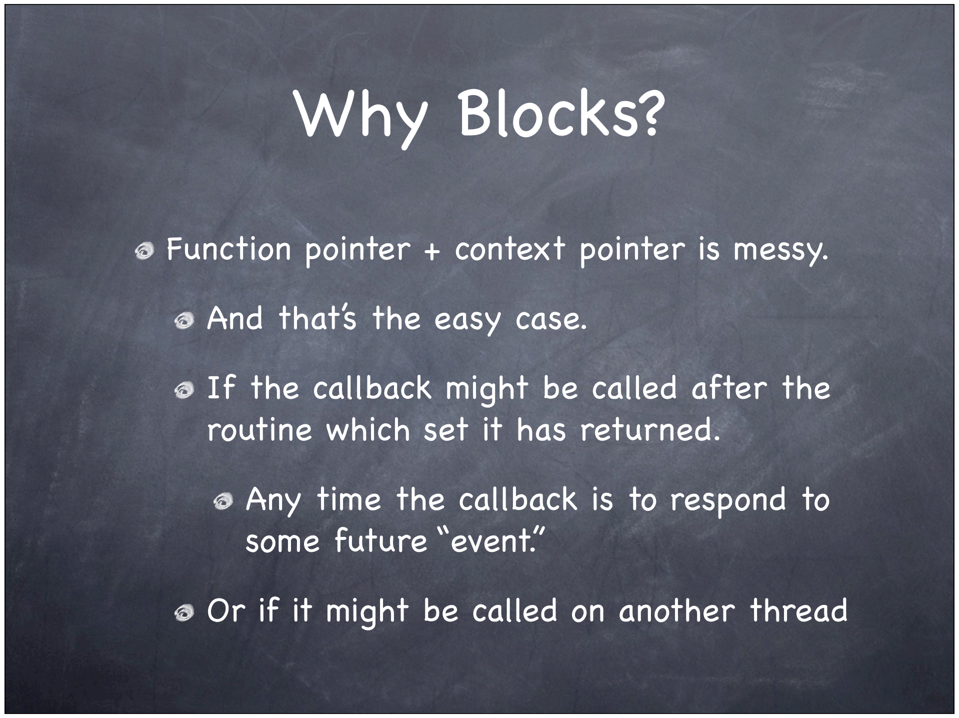
1: Callbacks and related uses are are good use for blocks. In this case I'm using the term to include not only traditional callbacks, but also dispatching work to another thread, handling "events", and simple synchronous cases where the block is called one or more times on the thread that started it, which the originating routine awaits the result.
4: If your callback needs to deal with a bunch of different "things" and they aren't all known until runtime, you couldn't write a separate function for each one, even if you wanted to. So the callback needs access to information that differs for each one.



Why Blocks?

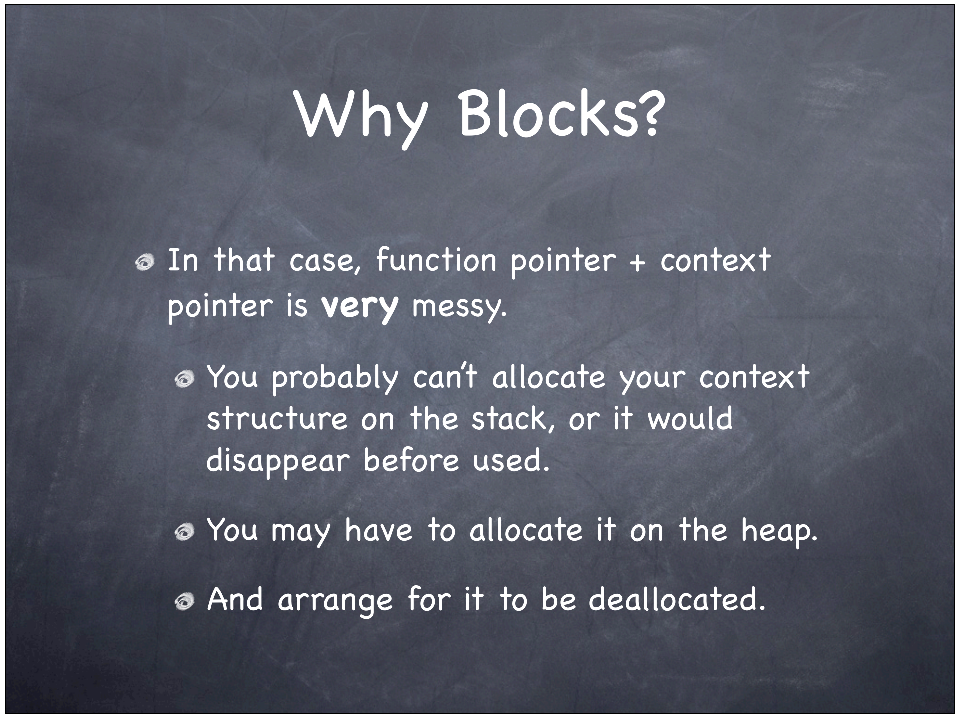
- Function pointer + context pointer is messy.
  - If there's more than one piece of information needed, you need to put it all in a struct, and pass a pointer to that as the context pointer.
  - Using a "void *" pointer bypasses type checking.

2: Even if you have a single piece of state information, you might do this anyway to avoid maintenance issues when that changes.
3: The routine that accepts the callback parameters doesn't know the type of the state information that the callback will need. It's just a middleman, passing data it doesn't need between two routines that do.



Why Blocks?

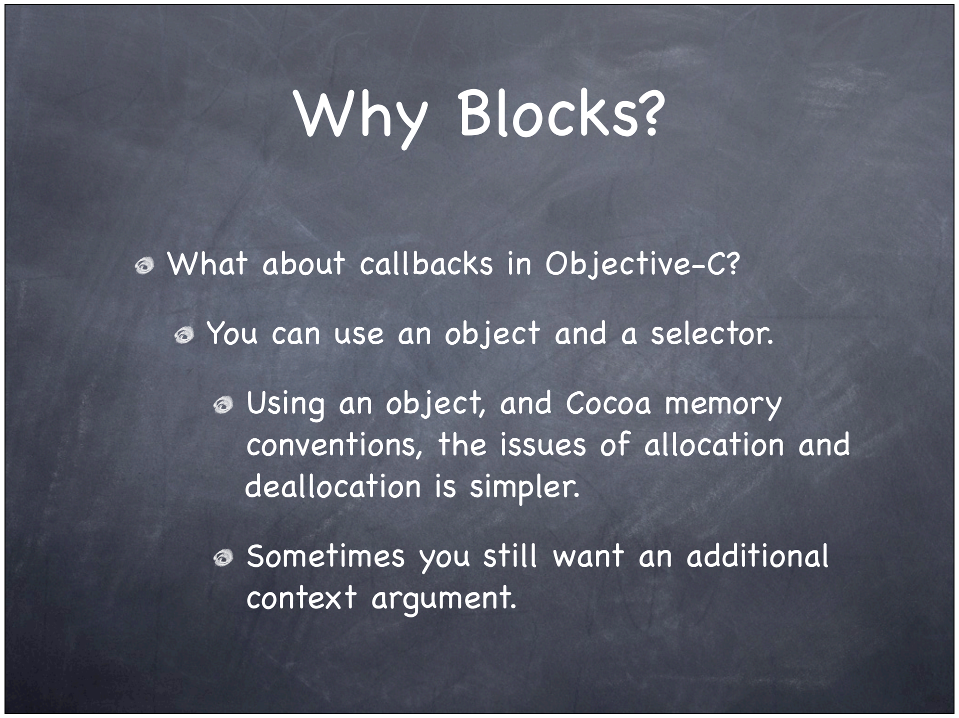
- Function pointer + context pointer is messy.
    - And that's the easy case.
    - If the callback might be called after the routine which set it has returned.
        - Any time the callback is to respond to some future "event."
    - Or if it might be called on another thread



Why Blocks?

- In that case, function pointer + context pointer is **very** messy.
  - You probably can't allocate your context structure on the stack, or it would disappear before used.
  - You may have to allocate it on the heap.
  - And arrange for it to be deallocated.

4: Not too soon, and not too late.



Why Blocks?

- What about callbacks in Objective-C?
  - You can use an object and a selector.
    - Using an object, and Cocoa memory conventions, the issues of allocation and deallocation is simpler.
    - Sometimes you still want an additional context argument.

1: That's C. What about Objective-C?
1: Since it includes C, you can use the same techniques, but there are some additional possibilities.
4: Often in Objective-C this has been done by putting your context data into a dictionary, passing that to the special purpose method created just for this call, and then taking the dictionary apart again in that method to get all the needed information.

Why Blocks?

- But, even in the best case...
- It's still very messy.
    - Simple callbacks, which are often logically related to the code that sets them, have to be put in separate functions or methods.
    - Harder to follow to code, and more boilerplate.

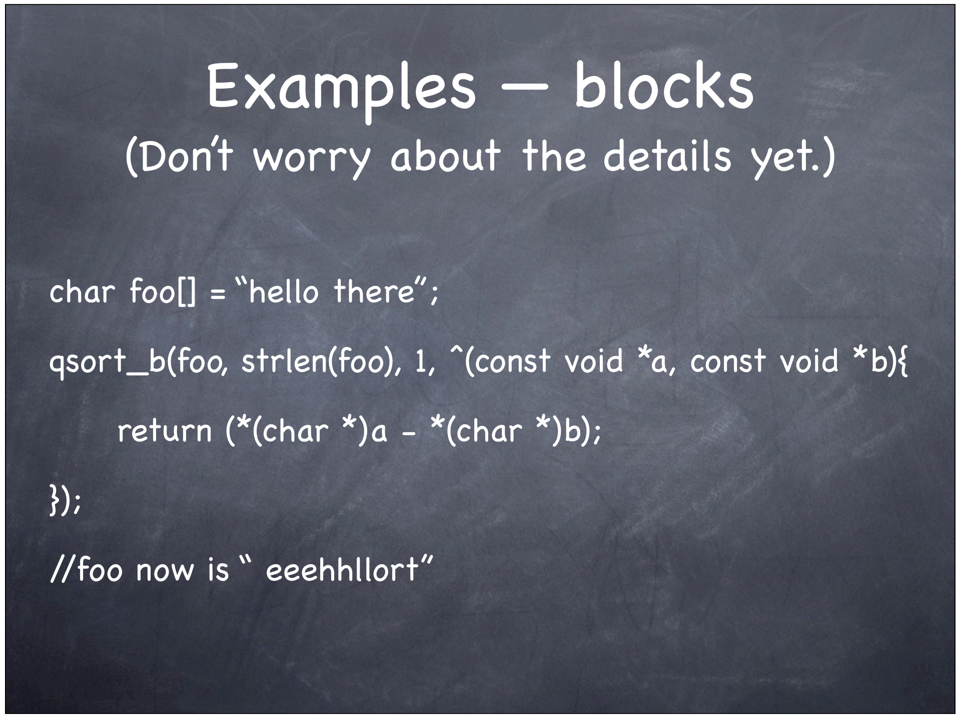
Why Blocks?

- Oversimplifying a bit, blocks are a better way to do callbacks.
  - That includes cases that aren't usually called "callbacks" — but are basically the same thing.

Examples — no blocks

(Don’t worry about the details yet.)

```
char foo[] = “hello there”;

int compare(const void *a, const void *b) {

   return (*(char *)a - *(char *)b);

}

qsort(foo, strlen(foo), 1, compare);

//foo now is “ eeehhllort”
```

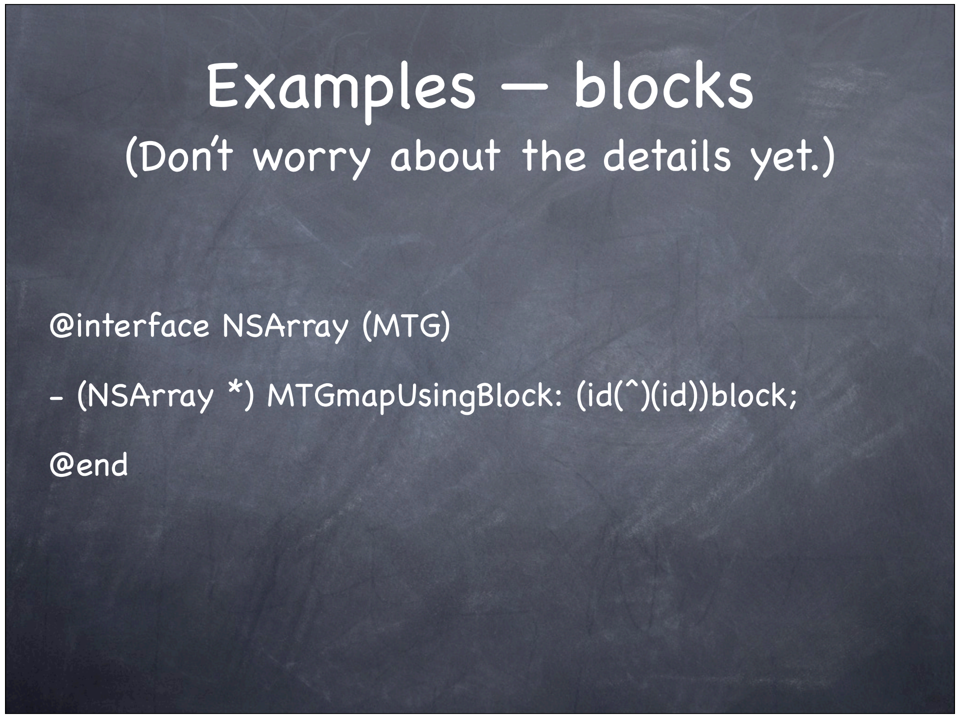


Examples — blocks

(Don't worry about the details yet.)

```
char foo[] = "hello there";
qsort_b(foo, strlen(foo), 1, ^(const void *a, const void *b){
	return (*(char *)a - *(char *)b);
});
//foo now is " eeehhllort"
```

3: The difference between these two examples isn't all that large. But I didn't even use the ability of blocks to capture state from the surrounding scope. In a case like that, you'd have had to use qsort_r for the pre-block case, and do all that structure mangling. Plus, in a larger program, it might not be obvious that the compare routine was subservient to the code calling qsort.

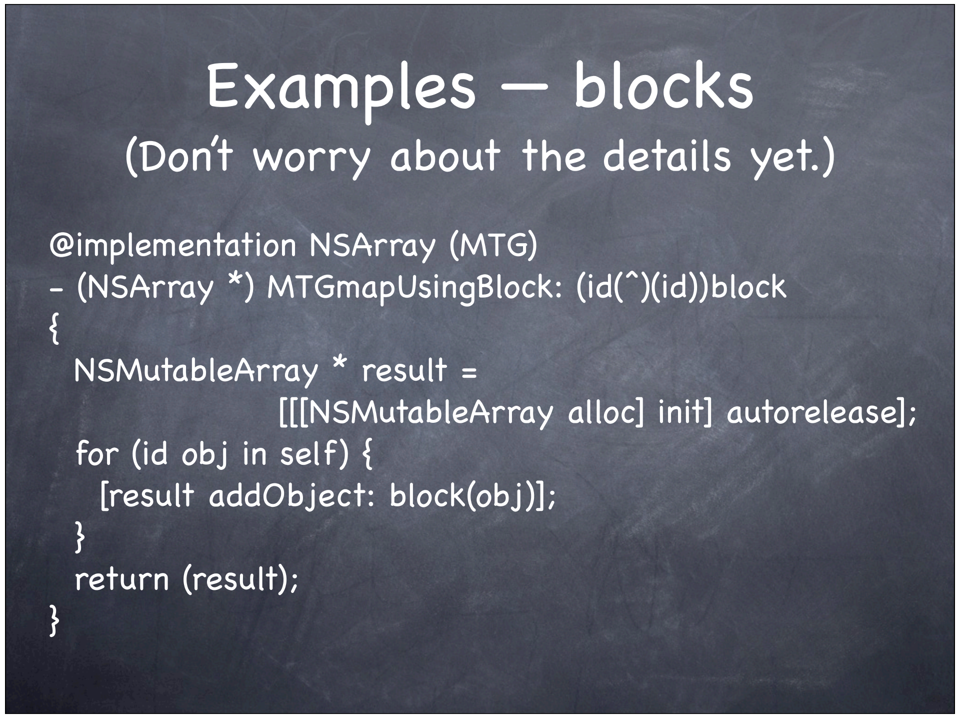


Examples — blocks

(Don't worry about the details yet.)

```
@interface NSArray (MTG)

- (NSArray *) MTGmapUsingBlock: (id(^)(id))block;

@end
```

1: Let's try an Objective-C example. Suppose we want to implement a map operation on arrays.
Here's the interface.
2: Yes, that syntax is ugly. In practice you could use a typedef to make it clearer

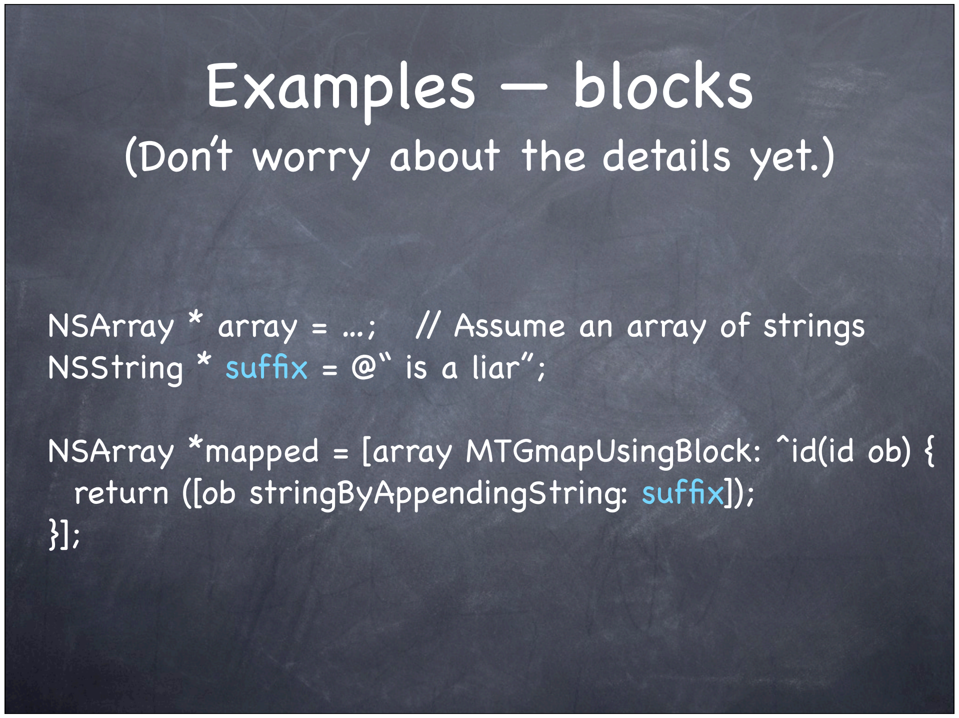


Examples — blocks

(Don't worry about the details yet.)

```
@implementation NSArray (MTG)
- (NSArray *) MTGmapUsingBlock: (id(^)(id))block
{
  NSMutableArray * result =
                  [[[NSMutableArray alloc] init] autorelease];
  for (id obj in self) {
    [result addObject: block(obj)];
  }
  return (result);
}
```

Here's the implementation.
The majority of the time, you'll be using Apple's routines which take block arguments, but you can see it's not too hard to define your own if you like.
*: A production quality routine might prefer to return an immutable array, but that's not part of the topic.



Examples — blocks

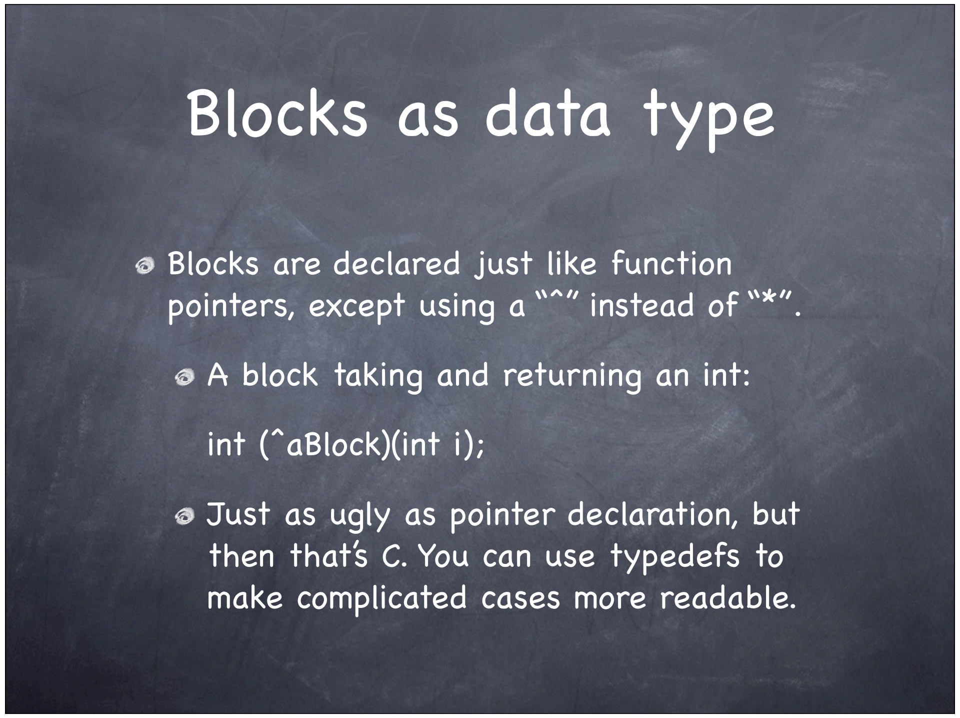
(Don't worry about the details yet.)

```
NSArray * array = …;    // Assume an array of strings
NSString * suffix = @" is a liar";

NSArray *mapped = [array MTGmapUsingBlock: ^id(id ob) {
  return ([ob stringByAppendingString: suffix]);
}];
```

And here we call it.
Most of the time, this is the sort of thing you'll end up doing — calling Apple's routines which accept a block.
Note how the block is able to use the "suffix" variable from the enclosing scope — something you'd have to arrange to pass through if you were defining map without blocks.
Simple, yes?

What Are Blocks?

- Blocks are inspired by similar features in other languages, such as blocks in Smalltalk, and closures in LISP.
- But in a C-like way they are:
  - a data type,
  - a set of operations you can perform, and
  - a syntax for providing a block as a literal



Blocks as data type

- Blocks are declared just like function pointers, except using a “^” instead of “*”.
  - A block taking and returning an int:

    ```
    int (^aBlock)(int i);
    ```

  - Just as ugly as pointer declaration, but then that’s C. You can use typedefs to make complicated cases more readable.

3: However, this syntax is mostly used by routines that accept blocks, or perhaps have them as instance variables. If you’re only calling existing APIs which use blocks, you may find things simpler.

Blocks as data type

- But what does a block value actually represent?
  - Apple doesn't specify, and you mostly don't need to know
  - But you can think of it as representing the address of a private structure

Blocks as data type

- Is it a pointer?
  - Apparently not quite. There are places in C that expect a pointer type, but won't accept a block value.
- They **are** objects, in the Objective-C sense.
  - But they are a bit weird. (More later.)

Blocks as data type

- What's **in** that pseudo-structure / object?
  - It's private. You don't need to know, and Apple doesn't say.
  - But if it helps to think of it, it obviously contains a pointer to the actual code of the block, plus the state captured from the enclosing scope.

Block Operations

- You can call a block, as you can call a function pointer
  - int (^aBlock)(int i) = ...;

    int a = aBlock(10);
  - More often you will pass a block as an argument to a function or method

Block Operations

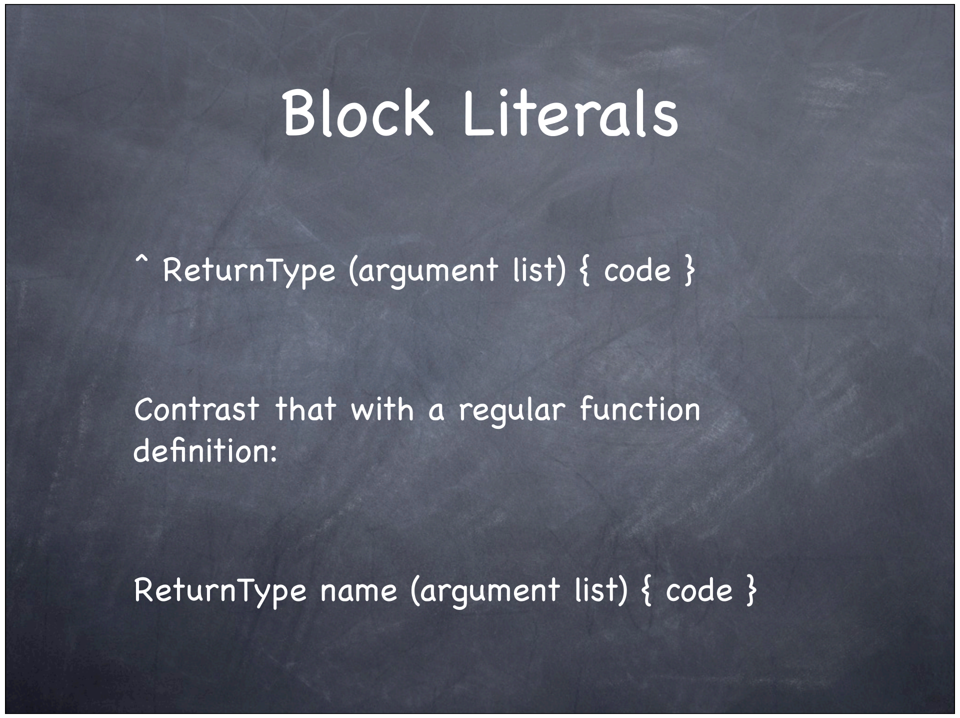
- There are also block operations related to memory management.
    - But there's a lot to say about blocks and memory management, so I'll postpone that until later.

Block Operations

- You can cast a block value to and from a pointer type like "void *"
  - You probably shouldn't
  - The only case I can think of when you'd want to do it is if you want some routine that takes a block argument of any type, and then you wouldn't be able to do anything useful with it. So don't.

Block Literals

- Block literals are how you actually write a block containing real code.
    - They're what you mean when you point to something and say "There's a block."



Block Literals

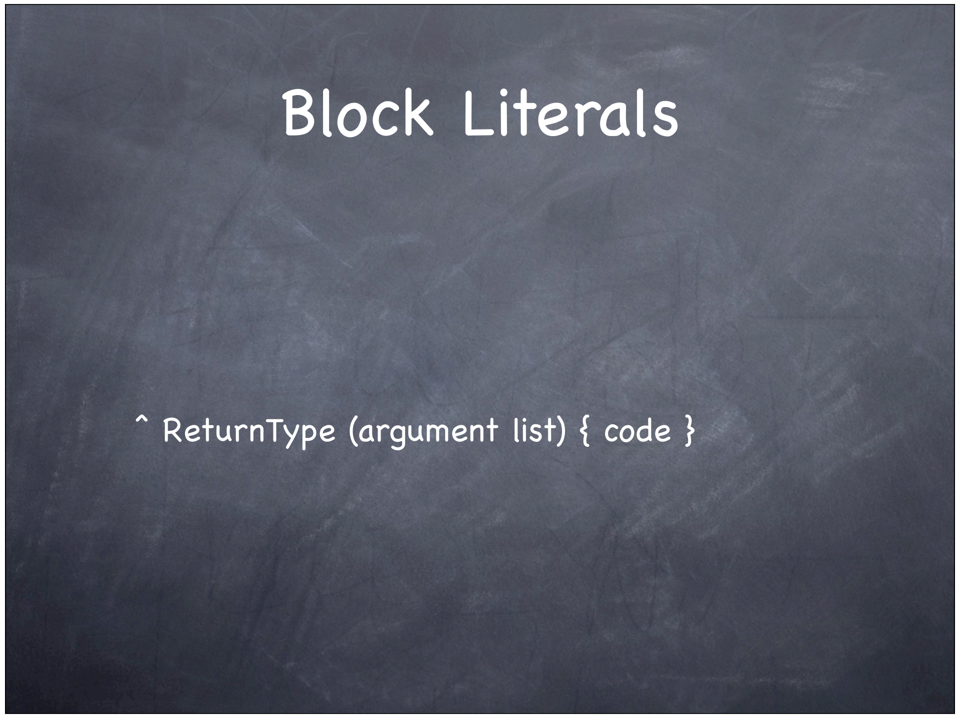
```
^ ReturnType (argument list) { code }
```

Contrast that with a regular function definition:

```
ReturnType name (argument list) { code }
```

1: It looks very much like a regular function definition, except for the caret, and the fact that there’s no function name.

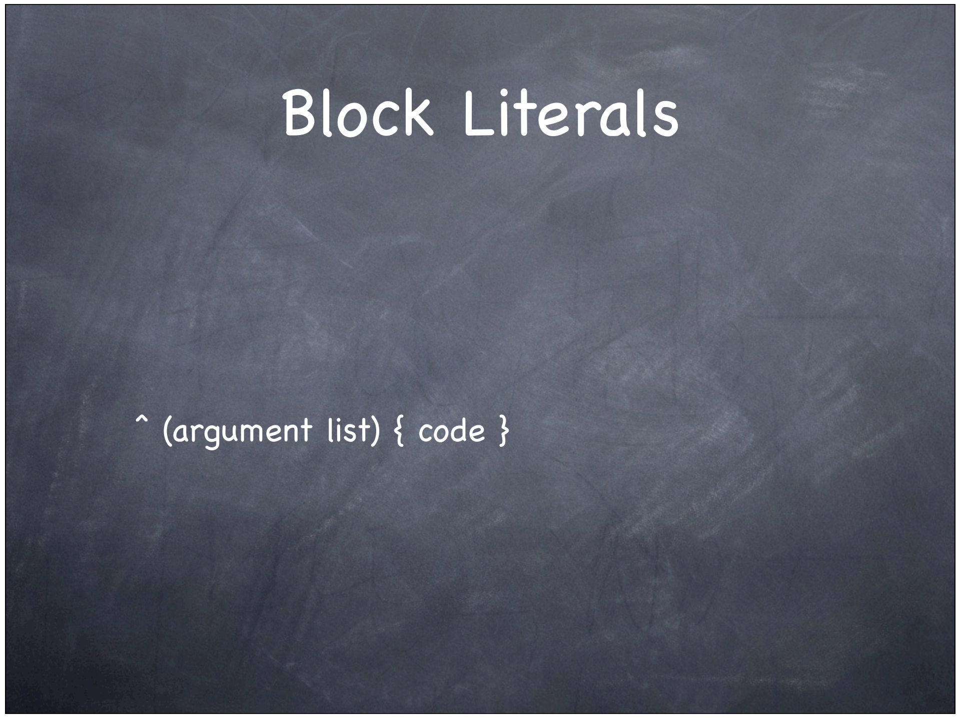
3: Note that this is a little simplified. In the case of complicated types the return type and function name are intermingled.



Block Literals

```
^ ReturnType (argument list) { code }
```

1: If the compiler can see that the code contains no return statement, meaning the return type would be void, you can omit it. If there is a return type, but the compiler can infer it from the actual return statements, you can still omit it. So in most cases you can write

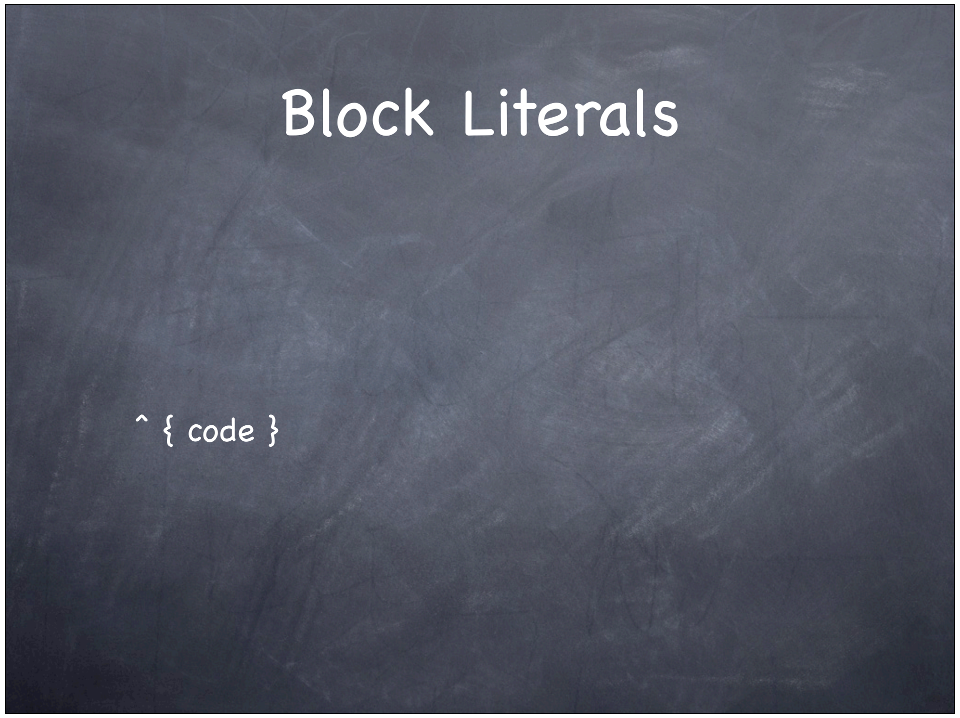


Block Literals

```
^ (argument list) { code }
```

In fact, you can almost always omit the return type. The only case where you really need it is where the compiler will infer the wrong return type. If you get an error about the return type, you can try putting it back in.

But suppose that the block takes no arguments. That’s actually fairly common, when the block is representing some anonymous piece of work. In that case, you can leave out the argument list as well, and you end up with



Block Literals

```
^ { code }
```

And that’s about as basic as you can get.

Block Literals

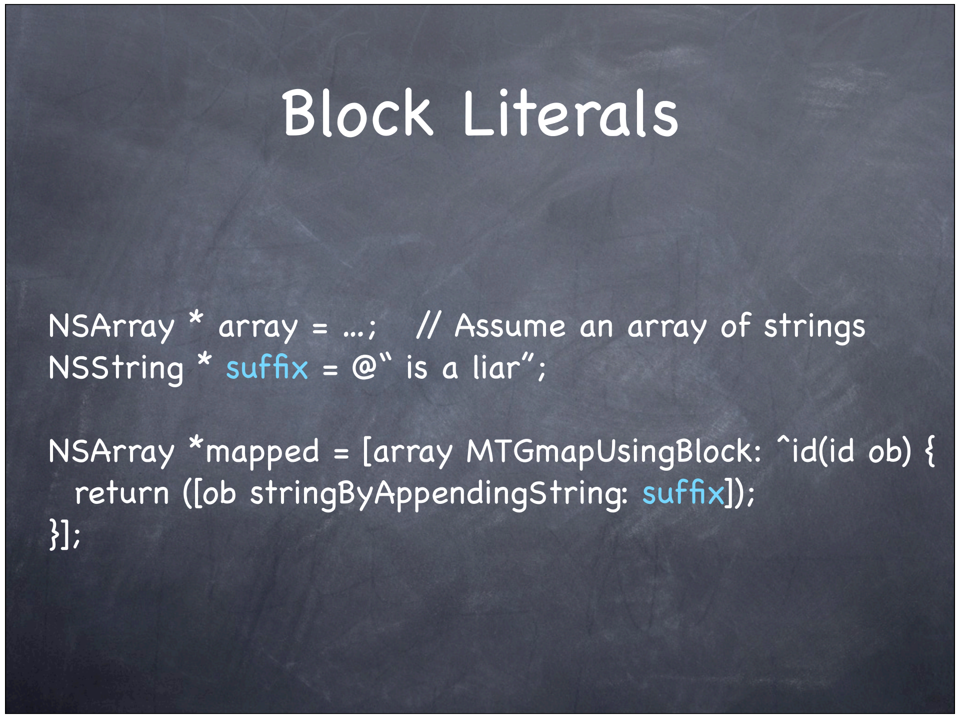
- What can you do with a block literal?
- You can assign it to a block variable, or instance variable
- But most of the time you just immediately pass it to some routine that will use it

Block Literals

- What's special about the code in a block literal?
- It can refer to variables defined in the enclosing scope
  - Two cases:
    - Read only access to a copy
    - Read / write access

Block Literals

- Read only access is the most common case
- Internally, the block makes a copy of the variable <u>at the time the block literal is evaluated</u>. Further changes to the variable will not be noticed by the block.



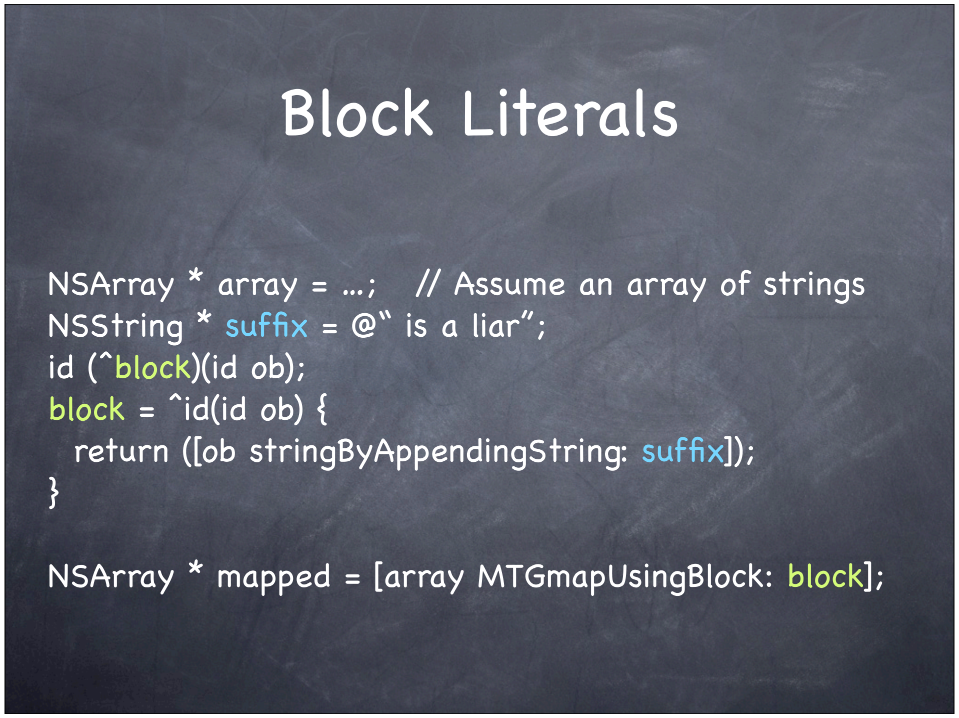
Block Literals

```
NSArray * array = ...;    // Assume an array of strings
NSString * suffix = @" is a liar";

NSArray *mapped = [array MTGmapUsingBlock: ^id(id ob) {
  return ([ob stringByAppendingString: suffix]);
}];
```

Let’s take a look at our example again.

See how the block uses the value of “suffix”.

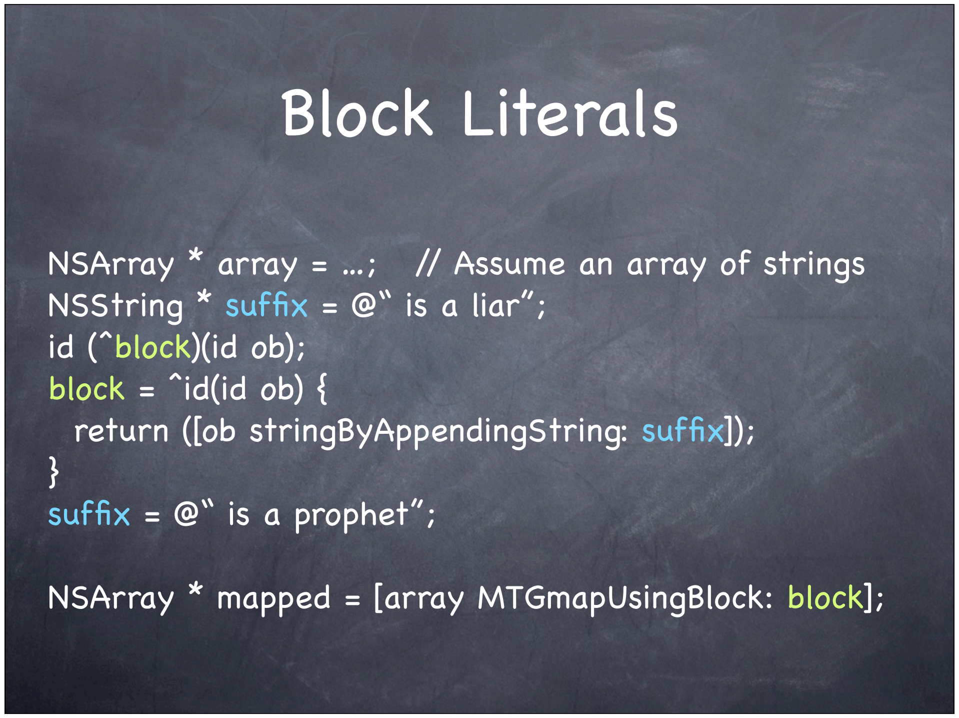
Let’s split it into two pieces.


Block Literals

```
NSArray * array = ...;    // Assume an array of strings
NSString * suffix = @" is a liar";
id (^block)(id ob);
block = ^id(id ob) {
  return ([ob stringByAppendingString: suffix]);
}

NSArray * mapped = [array MTGmapUsingBlock: block];
```

This is essentially the same thing, just separating the block literal from the call, and storing it in a variable named "block".

What happens if we alter "suffix" between those?

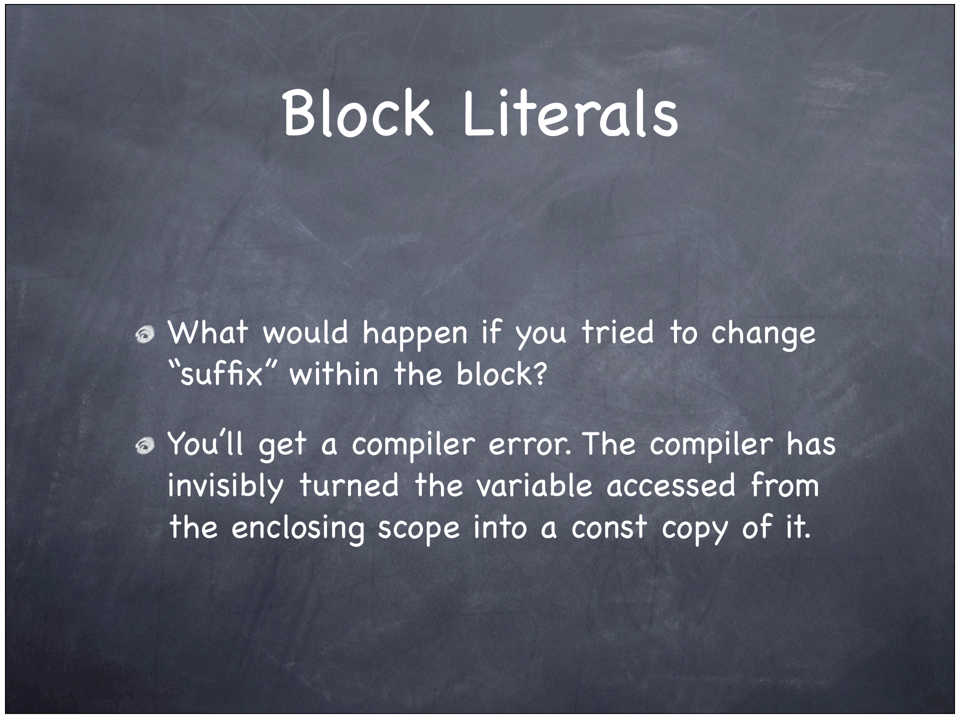


Block Literals

```
NSArray * array = …;    // Assume an array of strings
NSString * suffix = @“ is a liar”;
id (^block)(id ob);
block = ^id(id ob) {
  return ([ob stringByAppendingString: suffix]);
}
suffix = @“ is a prophet”;

NSArray * mapped = [array MTGmapUsingBlock: block];
```

The block does not see the new value of suffix, and produces the same result as before.

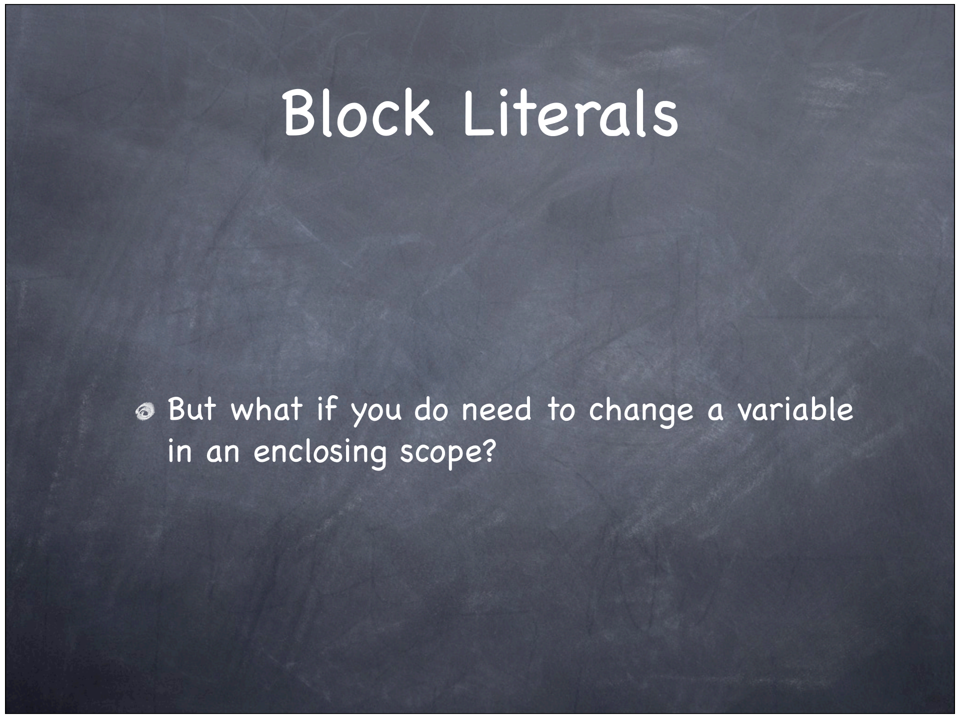
That’s because the copy of suffix is made when the block literal is evaluated, not when it’s executed.



Block Literals

- What would happen if you tried to change "suffix" within the block?
- You'll get a compiler error. The compiler has invisibly turned the variable accessed from the enclosing scope into a const copy of it.

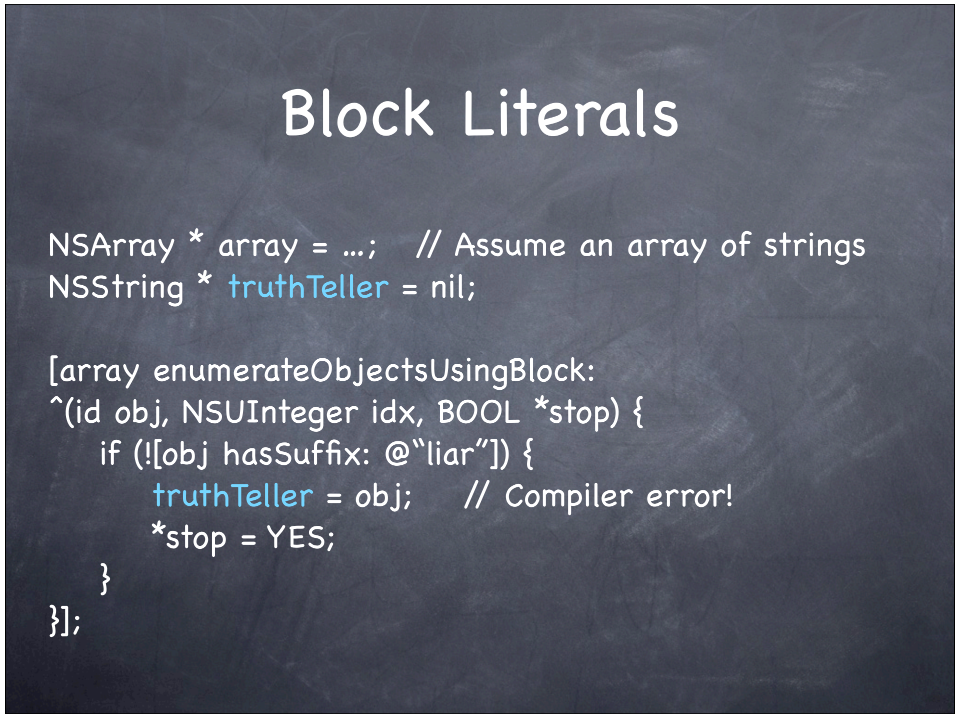
2: I find that to be another somewhat weird case. I can't think of anywhere else in C where something like that happens, though I might be missing another case somewhere. But weird or not, most of the time it is what you want.



Block Literals

- But what if you do need to change a variable in an enclosing scope?

Let’s look at another example.



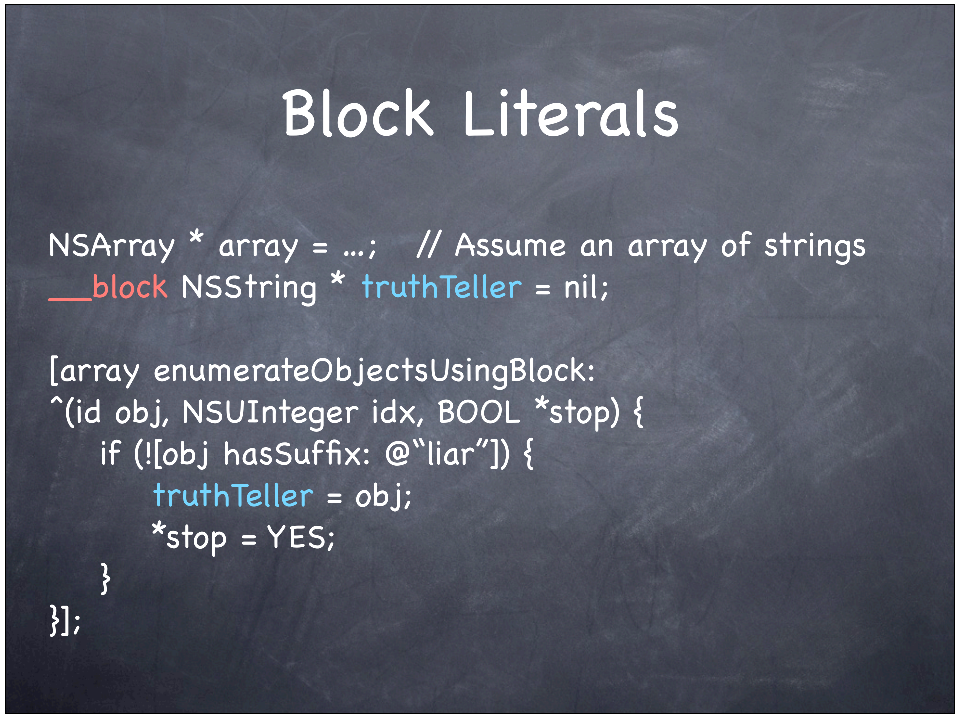
Block Literals

```
NSArray * array = ...;   // Assume an array of strings
NSString * truthTeller = nil;

[array enumerateObjectsUsingBlock:
^(id obj, NSUInteger idx, BOOL *stop) {
    if (![obj hasSuffix: @"liar"]) {
        truthTeller = obj;    // Compiler error!
        *stop = YES;
    }
}];
```

You could try this, but it would fail to compile. But note that it’s only the variable that’s read only. If “truthTeller” were a mutable string, you could change its value, even though you couldn’t change it to point to a different string object.

By the way, note that Apple has added yet another way to enumerate arrays and other container objects, and in some cases this is faster even than the recently introduced “fast enumeration”.

Don’t worry about this particular method on NSArray. The “stop” argument is a pointer to a boolean, allowing you to short circuit the iteration if desired.



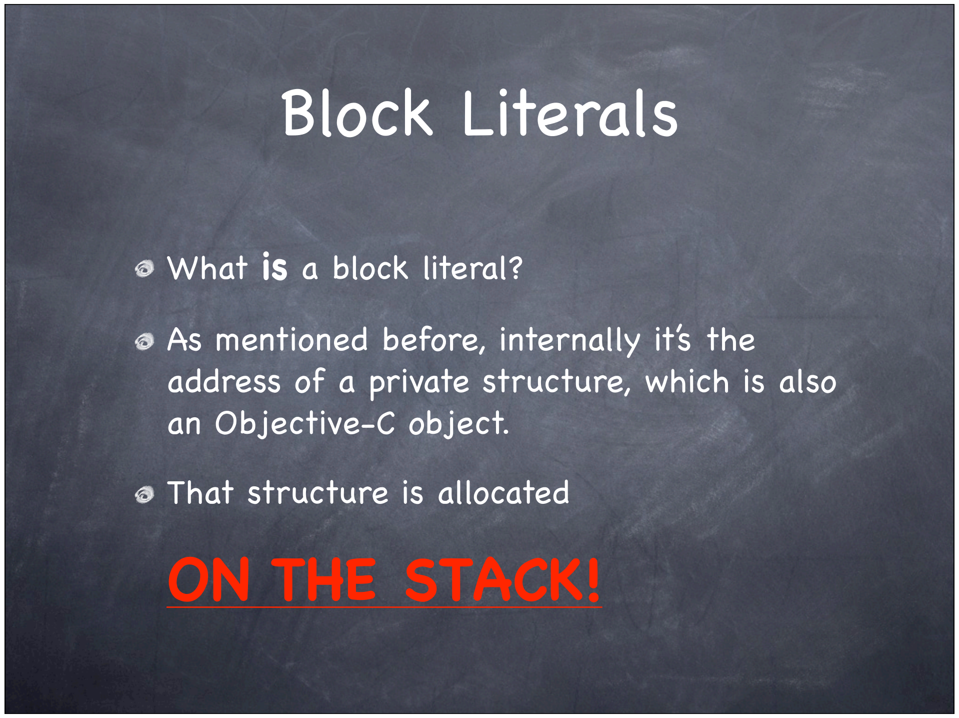
Block Literals

```
NSArray * array = ...;    // Assume an array of strings
__block NSString * truthTeller = nil;

[array enumerateObjectsUsingBlock:
^(id obj, NSUInteger idx, BOOL *stop) {
    if (![obj hasSuffix: @"liar"]) {
        truthTeller = obj;
        *stop = YES;
    }
}];
```

We can fix it by declaring “truthTeller” to be of the “__block” storage class. Note that there are two underbars in that symbol.

This is a new storage class, alongside the others like “auto” and “static”.

In simple cases like this it works well and is very useful. In more complicated cases it’s a little strange, and a lot less useful. We’ll get to that later.



Block Literals

- What **is** a block literal?
- As mentioned before, internally it's the address of a private structure, which is also an Objective-C object.
- That structure is allocated

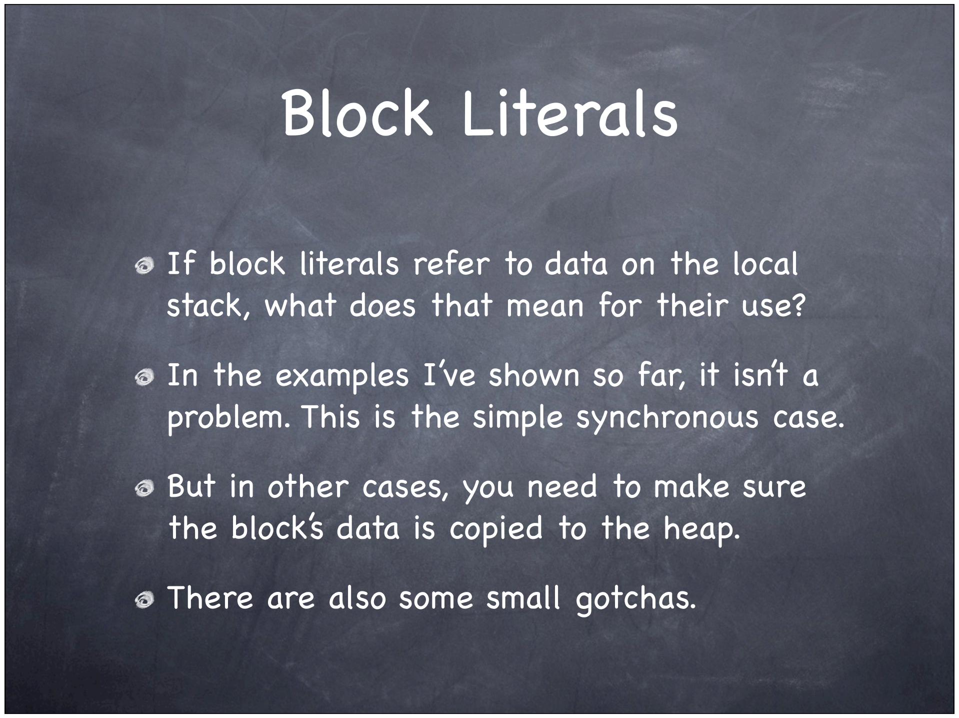
ON THE STACK!

3: This is important!

It's also weird. I can't think of another case where an Objective-C object is allocated on the stack.

It adds to the need to be careful about memory management in some cases, though thankfully in the most common cases it's all taken care of for you.

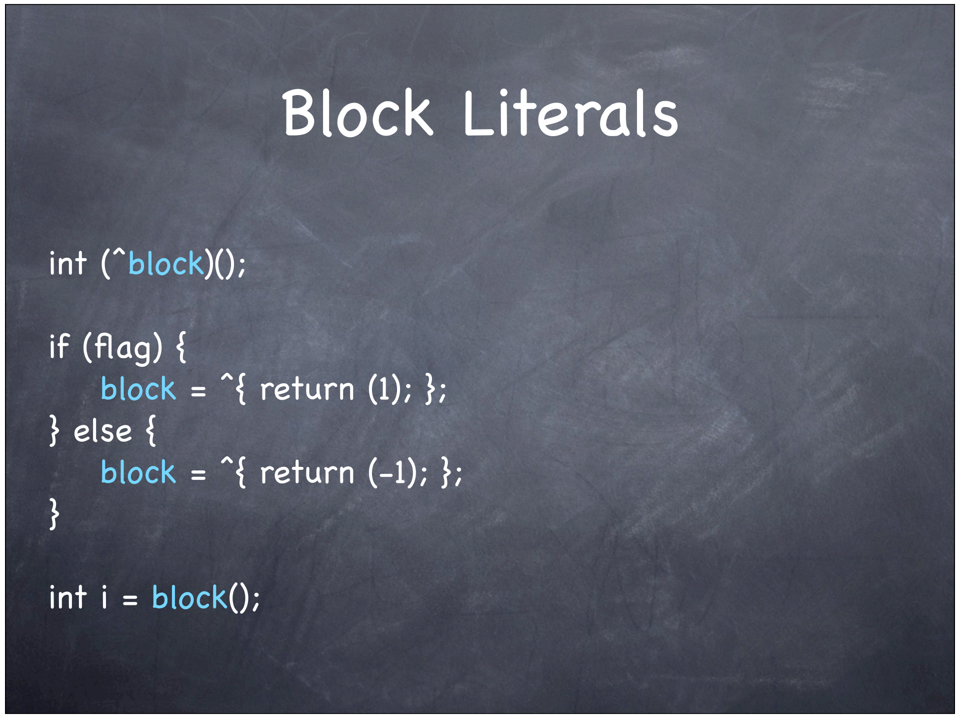
Why? It's purely about performance. Apple wants these to be fast, especially in the cases where stack allocation is OK.



Block Literals

- If block literals refer to data on the local stack, what does that mean for their use?
- In the examples I've shown so far, it isn't a problem. This is the simple synchronous case.
- But in other cases, you need to make sure the block's data is copied to the heap.
- There are also some small gotchas.

4: Let's look at those first, and then come back to the big gotcha.



Block Literals

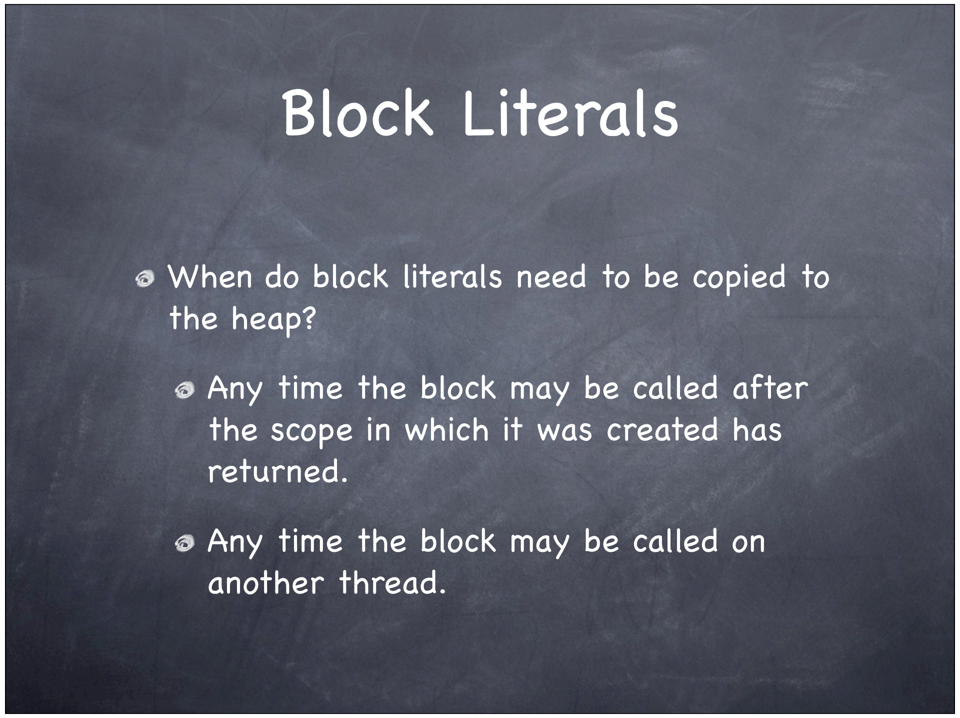
```
int (^block)();

if (flag) {
     block = ^{ return (1); };
} else {
     block = ^{ return (-1); };
}

int i = block();
```

This isn’t useful code, just something to show a point.
What’s wrong with it?

It might happen to work, but it might very well crash. It isn’t safe.

The if and else branches are C language scopes. The block literals are defined in those scopes, and may no longer exist once they’ve been exited. So the block being called may no longer exist. Of course you can get the same effect with any sort of nested code, including for and while loops.

In practice, this isn’t something you’ll run into very often, but you want to be aware of it just in case.

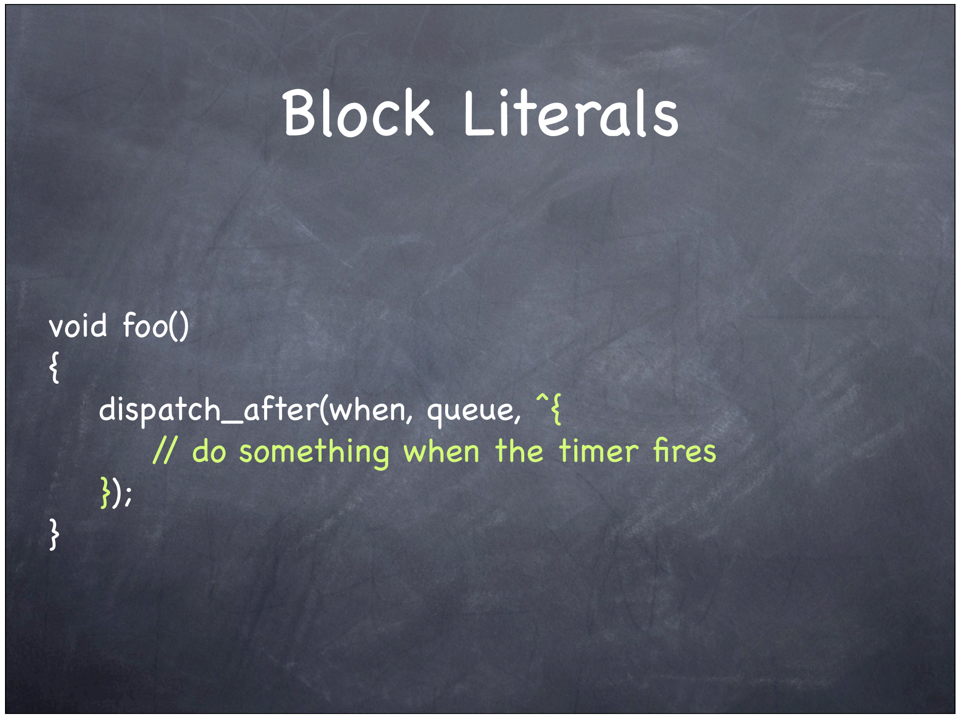


Block Literals

- When do block literals need to be copied to the heap?
  - Any time the block may be called after the scope in which it was created has returned.
  - Any time the block may be called on another thread.

2: This is the case when blocks are used for traditional callbacks. For example, Grand Central Dispatch allows you to specify a block to be called whenever a timer fires. That could happen long after the routine that defined the block has returned.

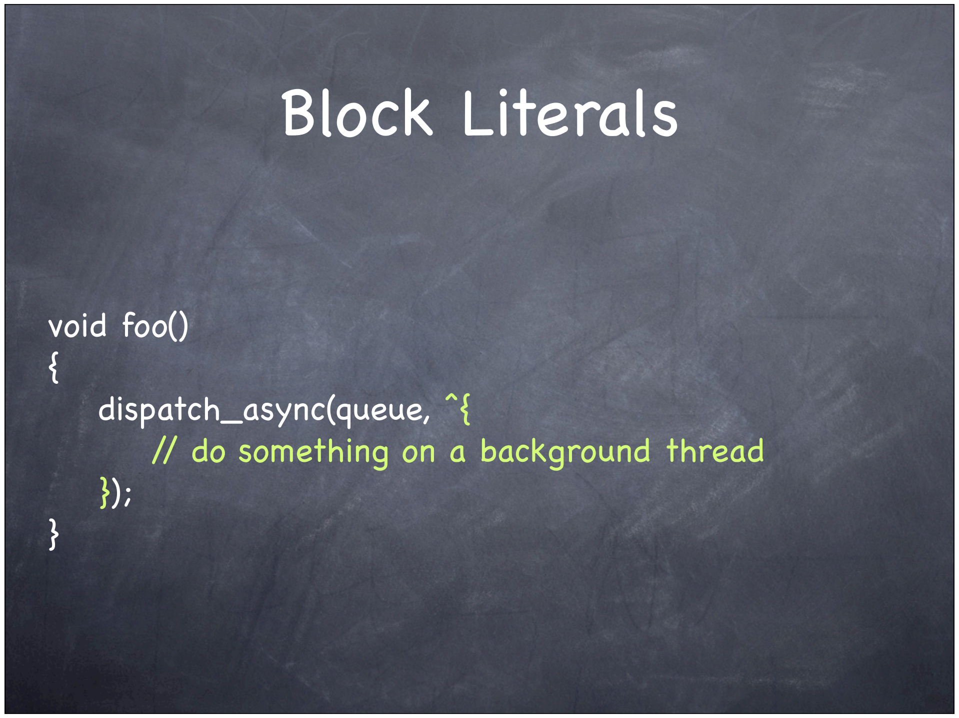
3: By Apple’s decree, even if the originating routine is waiting for that other thread to finsh. I think the main reason is that it can confuse the garbage collector, in which case it might work in a non-garbage collected environment. But I wouldn’t suggest trying it, as even if it works it could break in the future.



Block Literals

```
void foo()
{
    dispatch_after(when, queue, ^{
        // do something when the timer fires
    });
}
```

Here are some examples of these, using Grand Central Dispatch. There isn't time to talk about GCD tonight, but you should be able to see that this block, defined on the stack in the function “foo”, isn't going to still be there once the timer fires.



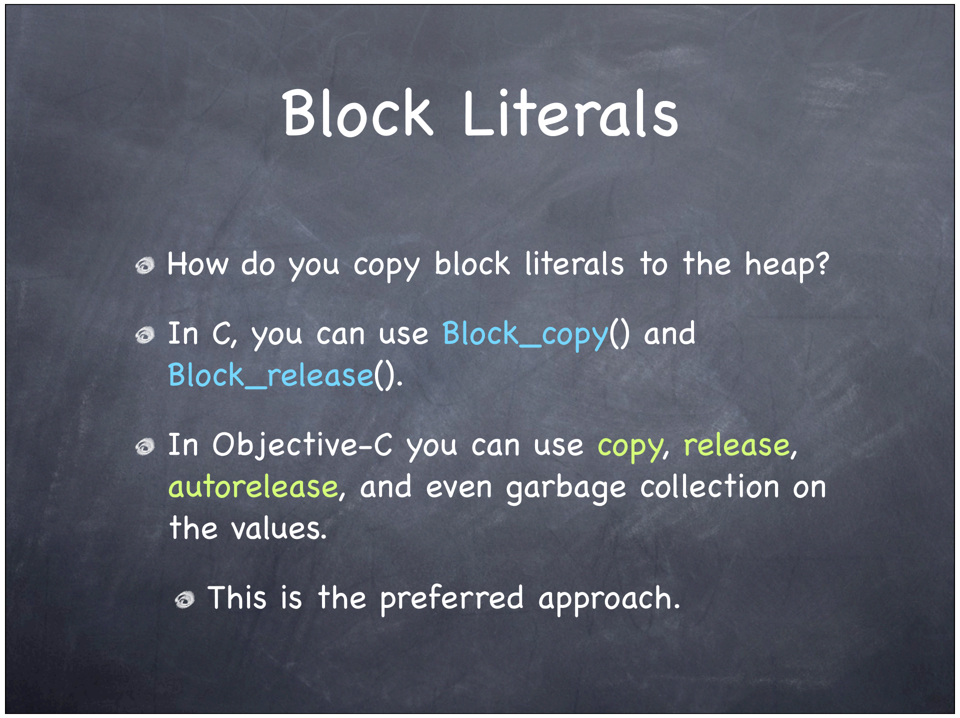
Block Literals

```
void foo()
{
    dispatch_async(queue, ^{
        // do something on a background thread
    });
}
```

Similarly, you can’t expect this block to still be there on the stack once that other thread runs.

However, both of these examples would work, but that’s because the GCD routines take care of copying the block to the heap for you.

In general, Apple’s API routines copy blocks to the heap any time the block’s execution might outlive the current stack frame. So if you’re only using blocks as arguments to Apple’s routines, you should be OK. But if you’re creating routines of your own that take blocks which might be executed non-synchronously, you should do this yourself, as a service to your callers.



Block Literals

- How do you copy block literals to the heap?
- In C, you can use Block_copy() and Block_release().
- In Objective-C you can use copy, release, autorelease, and even garbage collection on the values.
  - This is the preferred approach.

2: Block_copy copies its argument to the heap, and returns the new value. If it's already on the heap, it merely increments a reference count. Block_release decrements the reference count, and deletes the object if it's no longer referenced.

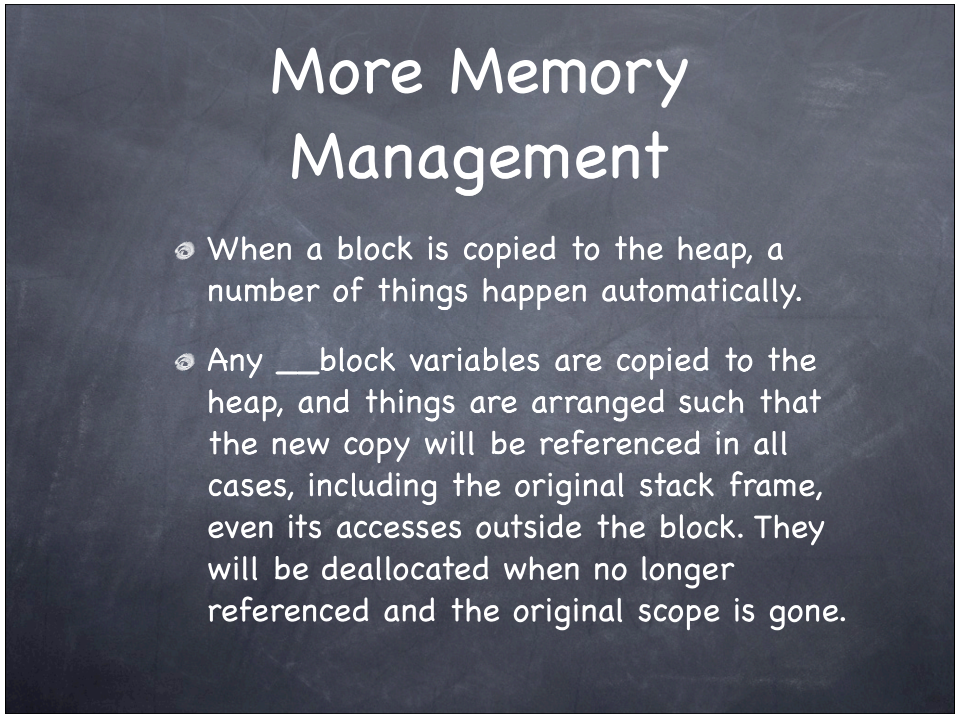
3: You could use retain, but it isn't really recommended.

More Memory Management

- OK, so most of the time the routines you pass a block to will take care of copying the block to the heap, and deleting it when it's no longer needed. What about the variables that are captured from the enclosing scope?

More Memory Management

- If the block might outlive its place of creation, you need to make sure the objects it accesses are still around.
- Some of that will be handled for you as well.

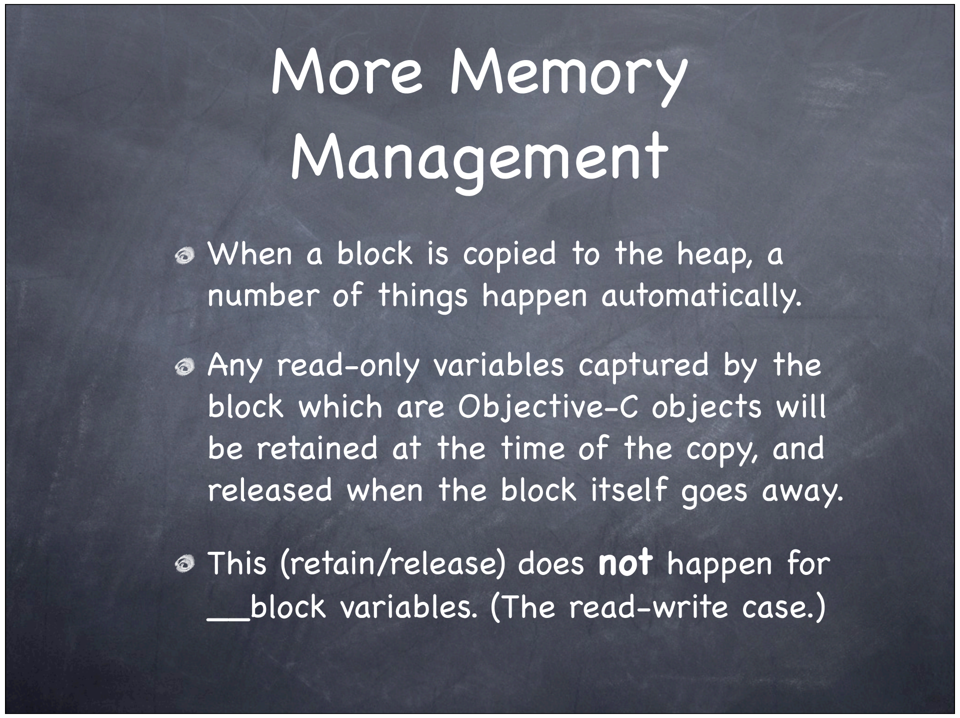

More Memory Management

- When a block is copied to the heap, a number of things happen automatically.
- Any __block variables are copied to the heap, and things are arranged such that the new copy will be referenced in all cases, including the original stack frame, even its accesses outside the block. They will be deallocated when no longer referenced and the original scope is gone.

2: There are a couple of gotchas here.

You can take the address of an __block variable using &, like any other variable. But it's address will change when it's copied to the heap. This is another one of those cases where blocks have a special case that doesn't occur anywhere else in the language.

Also, __block variables can not be certain types, in particular arrays (in the C sense). I believe that's due to C++ issues with copying, but it applies in C and Objective-C as well. (I did say I wasn't going to deal with C++, but for you C++ hackers note that all this copying will cause copy constructors to be executed. Just a word to the wise.)



More Memory Management

- When a block is copied to the heap, a number of things happen automatically.
- Any read-only variables captured by the block which are Objective-C objects will be retained at the time of the copy, and released when the block itself goes away.
- This (retain/release) does **not** happen for __block variables. (The read-write case.)

2: This handles a lot of the common cases of memory management for you, once again.

3: I'm not entirely sure why. An Apple engineer said that it was because in certain obscure cases it was impossible for it to do it correctly. I suspect it might be another C++ effect, though I'm not certain of that.

More Memory Management

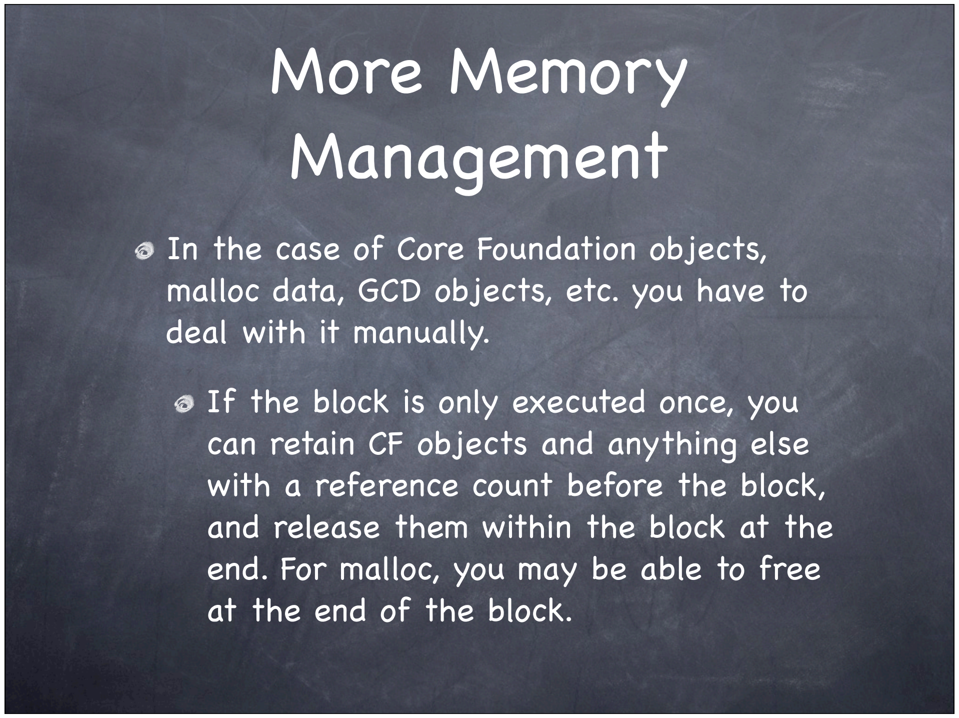
- I haven't mentioned this case so far, but if a block is defined within an Objective-C instance method, the block can reference its instance variables. In effect, it is capturing a read-only reference to "self". In these cases, it is "self" that is retained and released. (Direct references to "self" and "super" do the same thing, of course.)

More Memory Management

- A warning about all this automatic copying.
- If a variable is sometimes undefined, it will try to copy it anyway. If it's expecting it to be an Objective-C object... BOOM.
- It might look safe because you carefully arrange to only use it when it is defined, but it is not safe.

More Memory Management

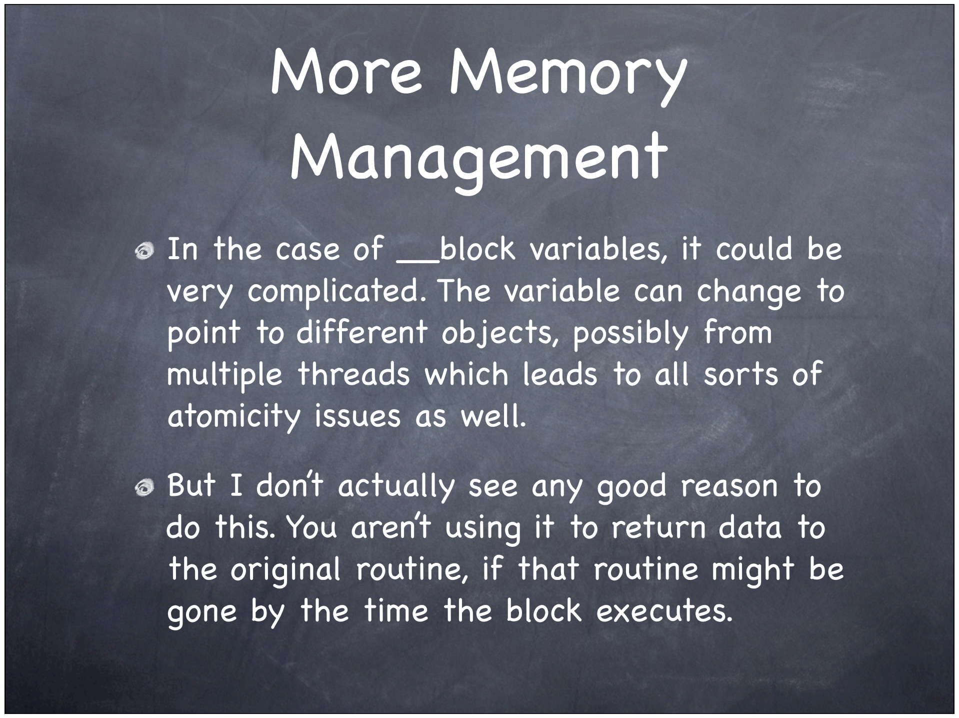
- What memory management isn't handled automatically?
- In the non-synchronous case:
  - Core Foundation objects, malloc data, GCD objects, etc.
  - Objects referenced by __block variables.



More Memory Management

- In the case of Core Foundation objects, malloc data, GCD objects, etc. you have to deal with it manually.
  - If the block is only executed once, you can retain CF objects and anything else with a reference count before the block, and release them within the block at the end. For malloc, you may be able to free at the end of the block.

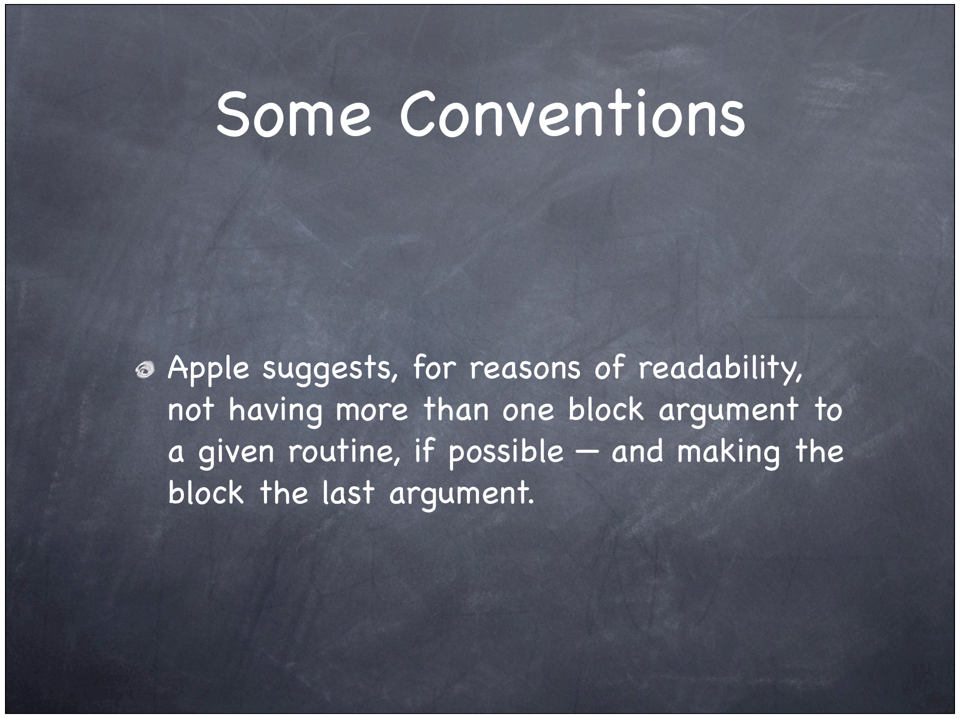
2: If the block will be executed multiple time, AND non-synchronously, it's going to be more complicated. That's what we get paid for, I guess.



More Memory Management

- In the case of __block variables, it could be very complicated. The variable can change to point to different objects, possibly from multiple threads which leads to all sorts of atomicity issues as well.
- But I don't actually see any good reason to do this. You aren't using it to return data to the original routine, if that routine might be gone by the time the block executes.

2: Apple describes that it's compilers do what's necessary to make this work "correctly" — but personally I would avoid using __block in anything but the simple case where the block is executed synchronously, and the variable is used only to return information from the block to the outer routine. Perhaps it will turn out that there are other good reasons to use the feature, but right now that's my feeling.



Some Conventions

- Apple suggests, for reasons of readability, not having more than one block argument to a given routine, if possible — and making the block the last argument.

All the examples I've shown have done this. The alternatives are mostly a lot harder to read.

Some Conventions

- In Objective-C, think about how you name your block arguments. Block is a little ambiguous. Apple suggests using it only for generic cases, and preferring something more descriptive when possible.

Block Literals

addOperationWithBlock:
  but
indexesOfObjectsPassingTest:
sortUsingComparator:
addLocalMonitorForEventsMatchingMask:handler:
beginBackgroundTaskWithExpirationHandler:
beginSheetModalForWindow:completionHandler:

Q & A

- Any questions?

The End

- Good night.