Advanced Debugging with LLDB

Video 1 - Intro - On-Set

Hi, I'm [presenter] and I'm an iOS developer with T_____.

In this workshop we'll be exploring intermediate and advanced debugging techniques using Xcode and the LLDB debugger. This material is meant to supplement ideas and techniques you learned in the "Debugging for iOS" workshop. If you haven't completed that workshop and don't have at least some experience with LLDB, you might have trouble following the discussion. You'll get more out of this workshop if you go back and complete "Debugging for iOS" first.

We'll assume you're familiar with LLDB, but unless you're proficient with debuggers, there are probably parts of LLDB that still seem a bit cryptic: numerous ways examining data, details about *Stack frames,* more *threads* than you expected and the occasional peek at blocks of *assembly language instructions* can make LLDB a bit daunting.

In this workshop we'll be exploring debugging techniques, but the key lesson will be in understanding how LLDB works: how it sees your application and how to better interpret the information it provides.

Debuggers are complex tools; LLDB can tell you nearly *anything* about the execution state of your application which, in a modern multithreaded environment, is a *huge* number of details. Many of these details are not terribly useful for troubleshooting your application, and a proficient user of LLDB can quickly focus on the key pieces of information that are the most likely to be helpful in solving the puzzle of what is going wrong in your program.

In practice, debugging an application is often a matter of following a trail of "clues" that lead to a bad bit of logic or an odd case that isn't being correctly handled. The ability to successfully filter the information that LLDB is presenting and focus on the "clues" that matter is the difference between frustration and efficient debugging.

Being able to sort the *useful,* from the *not-so-useful* and *irrelevant* details partly depends on understanding how LLDB sees your program.

Learning to see your program through the "eyes" of LLDB is perhaps the most important lesson in this workshop, so we'll start with that.

One side note: throughout this workshop we'll be working with a simple iOS UITableview application. The Xcode project along with links to more information about LLDB and related topics are available in the teacher's notes. I encourage you to load the test project and experiment with LLDB commands and try the *challenges* I will be suggesting throughout the workshop.

Ok, on to execution through the "eyes" of LLDB

Video 2 - ScreenCast - See code from LLDB's perspective

Much of the time, developing software in Xcode means working from the perspective of the *compiler* anticipating the compilation of your high-level-language code. What matters when you're writing code are the syntactic rules of the language, whether that is Swift or Objective-C. Issues such as matching curly braces and variables being declared and used in ways that are consistent with their *types* are what Xcode is checking for.

In contrast, a debugger knows nothing about high-level language rules. LLDB's job is monitoring the hardware state of the processor as it executes instructions that *once were* Swift or Objective-C.

To illustrate this, let's look at the method -simpleStuff in ObjcDataSource.m in our tableview project. The method doesn't do much, and it actually it doesn't do anything useful at at all as far as the tableview is concerned, but it's extremely short and simple, so it's perfect for this example:

```
-(void) simpleStuff {
    int a,b,c;
    a = 3;
    b = 7;
    c = a + b;

    printf("sum of a+b+c = %d\n",(a+b+c));
}
```

As you might imagine, when I typed in this method, Xcode enforced all the syntactic rules of Objective-C. It complained when my braces and parens didn't match and it "kept me honest" about how I used the **int** variables: **a,b** and **c.** The rules it enforced were those that ensured that when time came to build the project, compilation of this module would be done successfully.

To see this method from LLDB's perspective we'll set a breakpoint at the assignment 'a=3;' by clicking in the sourcecode gutter. Then we'll run the application until we stop at the breakpoint, then enter "**disassemble"** or "**di"** at the LLDB command prompt in the console... which gives us:

```
(lldb) di
Tbl_MixedDebug`-[ObjcDataSource simpleStuff]:
    0x2f450 <+0>:  pushl  %ebp
    0x2f451 <+1>:  movl   %esp, %ebp
    0x2f453 <+3>:  subl   $0x28, %esp
    0x2f456 <+6>:  calll  0x2f45b                  ; <+11> at ObjcDataSource.m:48
    0x2f45b <+11>: popl   %eax
    0x2f45c <+12>: movl   0xc(%ebp), %ecx
    0x2f45f <+15>: movl   0x8(%ebp), %edx
    0x2f462 <+18>: leal   0x2df1(%eax), %eax
    0x2f468 <+24>: movl   %edx, -0x4(%ebp)
    0x2f46b <+27>: movl   %ecx, -0x8(%ebp)
->  0x2f46e <+30>: movl   $0x3, -0xc(%ebp)
    0x2f475 <+37>: movl   $0x7, -0x10(%ebp)
    0x2f47c <+44>: movl   -0xc(%ebp), %ecx
    0x2f47f <+47>: addl   -0x10(%ebp), %ecx
    0x2f482 <+50>: movl   %ecx, -0x14(%ebp)
    0x2f485 <+53>: movl   -0xc(%ebp), %ecx
    0x2f488 <+56>: addl   -0x10(%ebp), %ecx
    0x2f48b <+59>: addl   -0x14(%ebp), %ecx
    0x2f48e <+62>: movl   %eax, (%esp)
    0x2f491 <+65>: movl   %ecx, 0x4(%esp)
    0x2f495 <+69>: calll  0x31f68                  ; symbol stub for: printf
    0x2f49a <+74>: movl   %eax, -0x18(%ebp)
    0x2f49d <+77>: addl   $0x28, %esp
    0x2f4a0 <+80>: popl   %ebp
    0x2f4a1 <+81>: retl
    0x2f4a2 <+82>: nopw   %cs:(%eax,%eax)
(lldb)
```

If you're not familiar with x86 assembly, this might seem pretty cryptic, but don't worry; this workshop is *not* about learning assembly language and we won't be decoding all this.

The point is that *this* is how LLDB sees a debugging target, since the *reality* of execution is the sequence of machine instructions and how typeless binary values are moved between memory and CPU registers.

We won't be going into much detail about this block of instructions, but there are a few things that this disassembled code can tell us about how LLDB sees our method.

The arrow in the left margin is where the breakpoint has stopped execution. In this case it's **'a move**' instruction('movl') that's stores '0x3' hex (which is also '3' in decimal) at some memory location pointed to by a register.

Because of where we're stopped (where we had a breakpoint set) and the decimal value '3' we can assume that the move instruction is the assembly language generated for the Objective-C line, 'a=3;'. The instruction is *moving* an **int**-sized constant value to a memory offset calculated using the *%ebp* register. This offset is the location of the variable **a.**

Without decoding any more machine instructions we can see a few things about what compilation did to our Objective-C and what LLDB has to work with:

1 -- As I mentioned, LLDB sees instructions, *not* your source-code lines. The debugger's job is to tell you what's going on as your program executes as machine instructions on the processor.

When it runs your code it isn't Swift or Objective-C anymore: LLDB has some symbol information about the precompiled code and it does it's best map what it sees back to these lines of high-level language, but machine instructions are the reality of how your program does it's work.

2 -- There are a *many* more lines of machine instructions for this method than there were lines of Objective-C.

'a=3;' mapped to a single line of assembly, but typically there are multiple instructions for every line of high-level language code. For one thing, this means '*stepping*' through high-level language source-code can be a complicated process. "Stepping" a line of source code really means advancing, executing some number of instructions that correspond to the current line of high-level language.

One situation where the mapping between high-level language source code lines and machine instructions becomes especially problematic is *optimized code*. The nature of the "complications" associated with debugging optimized code (where **clang** has compiled under the -O -O2 flags) is that, in the interest of creating a more efficient sequence of instructions, **clang** is allowed to sacrifice the mapping between high-level language code and assembly. Overall the functionality is the same, but "single-stepping" may no longer be completely meaningful.

Optimized code is one of the few situations where debugging an iOS application at the machine instruction level might actually be necessary. For such situations, LLDB offers the '**thread step-inst**" and "**thread step-over-inst**" commands for single stepping through assembly instructions.

As a challenge, try these instruction stepping commands and see how the pointer to the current instruction in the disassemble output changes and how the Objective-C pointer does or doesn't change.

3-- Assembly contains no variable names. Machine instructions work with binary values in CPU registers and memory locations. Symbols, such as your variable names, don't exist during program execution. The debugger does it's best to *simulate* your Swift or Objective-C program executing natively, but in reality LLDB maintains a mapping between stored binary values and variable names.

The reason LLDB offers so much flexibility in terms of how you can examine data is because, even with type information, the most useful representation for the developer is not always obvious from the debugger's perspective.

That was a lot of detail, but hopefully, knowing a little more about how the debugger sees program execution will make LLDB's interface and behaviors a little less mysterious.

With that in mind, let's experiment a little with how LLDB displays data values using the '**frame variable**' command.

Video 3 - Screencast - 'fr v', frames & threads

To keep things simple, we'll stay in the -simpleStuff method for the moment and keep the same breakpoint at the line 'a=3'

The most basic command for inspecting values is '**frame variable**' which can be shortened to '**fr v'.**

A short aside: as you may have realized, all LLDB commands can be shortened to any unambiguous abbreviation, so 'frame' and be 'fr', 'fra' or 'fram' and 'break' can be expressed with 'br' or 'bre' etc..

To review technical terms a little: the word '*frame*' anywhere in LLDB refers to *stack frame* which roughly corresponds to the local variables, or *scope* created since the last function or method call.

Each time there is such a call, the function's(or method's) return address, along with space to store the local variables and input parameters are "*pushed*" onto a place in memory referred to as the *stack*, to be "*popped*" off again when the function or method returns.

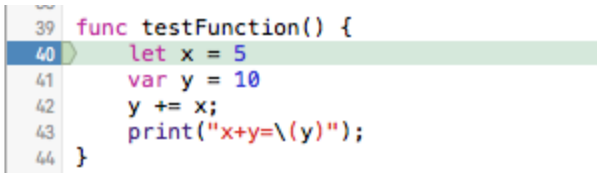
This is the mechanism by which local variables are created and destroyed as you make function calls and return.

There is more information about stacks and stack-frames in the teacher's notes.

If we enter ‘fr v’ at the LLDB command prompt while we’re stopped in -simpleStuff, we’ll see the local variables in the current scope along with a reference to the instance of ObjcDataSource that owns this instance method.

```
(lldb) fr v
(ObjcDataSource *) self = 0x00007fa32ad0ad20
(SEL) _cmd = "simpleStuff"
(int) a = 0
(int) b = 4
(int) c = 1
(lldb)
```

If we set a breakpoint in testFunction() in Menagerie.swift, the run the same LLDB command, we see:



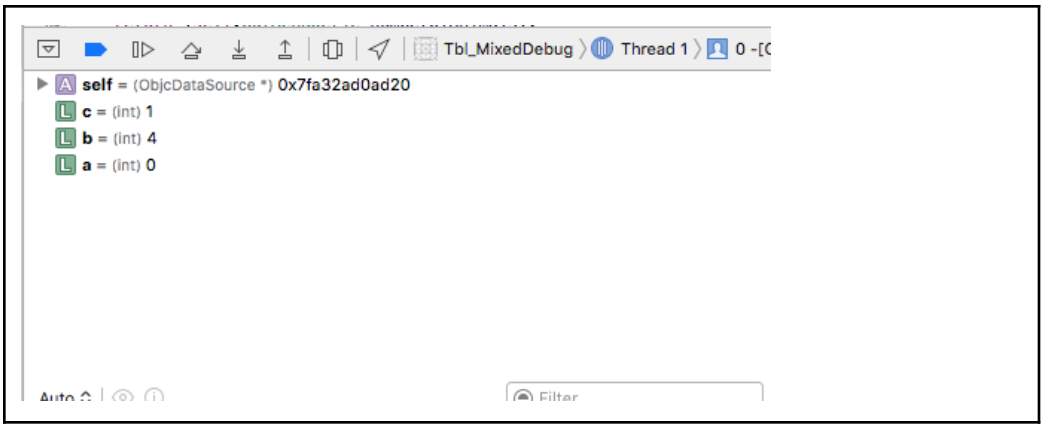

```
(lldb) fr variable
(Int) y = 140251406482336
(Int) x = 4512110200
```

The results are similar, except the method has no “self” reference since this is a pure(top-level) function, not an object method. Also note that the integers are Swift “Int” types rather than C primitive types.

As an experiment, try running ‘disassemble’ from this breakpoint in testFunction() and compare the output to that of the Objective-C method, -simpleStuff

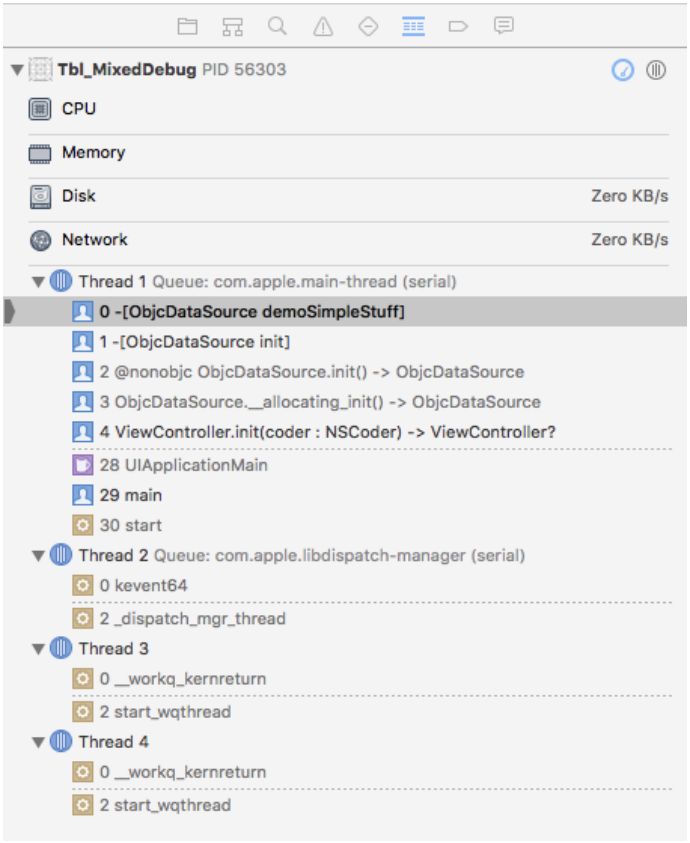
Let’s return to our breakpoint in -simpleStuff in the Objective-C module.

Having used LLDB before, you might already know that for both of these frames, the bottom-left pane in the Xcode UI displays the same output as the “frame variable” command



...and a stack trace of all frames is shown in the “Debug Navigator” pane on the left side of the Xcode interface.

This is called a ‘backtrace’ and shows the sequence of calls that lead to the current point in execution. The trace also indicates which frames of scope are currently being stored on the stack.



A more comprehensive backtrace is available by using the LLDB ‘**bt**’ command:

```
(lldb) bt
* thread #1: tid = 0x665da6, 0x00000001056a663c Tbl_MixedDebug`-[ObjcDataSource
simpleStuff](self=0x00007fd3787356d0, _cmd="simpleStuff") + 44 at ObjcDataSource.m:50, queue
= 'com.apple.main-thread', stop reason = breakpoint 1.1
  * frame #0: 0x00000001056a663c Tbl_MixedDebug`-[ObjcDataSource
simpleStuff](self=0x00007fd3787356d0, _cmd="simpleStuff") + 44 at ObjcDataSource.m:50
    frame #1: 0x00000001056a65dd Tbl_MixedDebug`-[ObjcDataSource
init](self=0x00007fd3787356d0, _cmd="init") + 237 at ObjcDataSource.m:34
    frame #2: 0x00000001056a7870 Tbl_MixedDebug`@nonobjc ObjcDataSource.init() ->
ObjcDataSource + 16 at ViewController.swift:0
    frame #3: 0x00000001056a7764 Tbl_MixedDebug`ObjcDataSource.__allocating_init() ->
ObjcDataSource + 68 at ViewController.swift:0
    frame #4: 0x00000001056a753c
Tbl_MixedDebug`ViewController.init(aDecoder=0x00007fd379823e00) -> ViewController? + 60 at
ViewController.swift:13
    frame #5: 0x00000001056a760d Tbl_MixedDebug`@objc ViewController.init(coder : NSCoder)
-> ViewController? + 45 at ViewController.swift:0
    frame #6: 0x00000001068c1095 UIKit`-[UIClassSwapper initWithCoder:] + 241
    frame #7: 0x0000000106a98e8a UIKit`UINibDecoderDecodeObjectForValue + 705
    frame #8: 0x0000000106a98bc0 UIKit`-[UINibDecoder decodeObjectForKey:] + 278
    frame #9: 0x00000001068c0d6b UIKit`-[UIRuntimeConnection initWithCoder:] + 180
    frame #10: 0x0000000106a98e8a UIKit`UINibDecoderDecodeObjectForValue + 705
    frame #11: 0x0000000106a9904b UIKit`UINibDecoderDecodeObjectForValue + 1154
    frame #12: 0x0000000106a98bc0 UIKit`-[UINibDecoder decodeObjectForKey:] + 278
    frame #13: 0x00000001068bff7d UIKit`-[UINib instantiateWithOwner:options:] + 1255
    frame #14: 0x0000000106c3a314 UIKit`-[UIStoryboard
instantiateViewControllerWithIdentifier:] + 181
    frame #15: 0x0000000106c3a467 UIKit`-[UIStoryboard instantiateInitialViewController] +
69
    frame #16: 0x00000001064f789f UIKit`-[UIApplication
_loadMainStoryboardFileNamed:bundle:] + 94
    frame #17: 0x00000001064f7bcf UIKit`-[UIApplication _loadMainInterfaceFile] + 260
    frame #18: 0x00000001064f63ef UIKit`-[UIApplication
_runWithMainScene:transitionContext:completion:] + 1392
    frame #19: 0x00000001064f3714 UIKit`-[UIApplication workspaceDidEndTransaction:] + 188
    frame #20: 0x00000001094c78c8
FrontBoardServices`__FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK__ + 24
    frame #21: 0x00000001094c7741 FrontBoardServices`-[FBSSerialQueue _performNext] + 178
    frame #22: 0x00000001094c7aca FrontBoardServices`-[FBSSerialQueue
_performNextFromRunLoopSource] + 45
    frame #23: 0x0000000106067301
CoreFoundation`__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ + 17
    frame #24: 0x000000010605d22c CoreFoundation`__CFRunLoopDoSources0 + 556
    frame #25: 0x000000010605c6e3 CoreFoundation`__CFRunLoopRun + 867
    frame #26: 0x000000010605c0f8 CoreFoundation`CFRunLoopRunSpecific + 488
    frame #27: 0x00000001064f2f21 UIKit`-[UIApplication _run] + 402
    frame #28: 0x00000001064f7f09 UIKit`UIApplicationMain + 171
    frame #29: 0x00000001056a7f62 Tbl_MixedDebug`main + 114 at AppDelegate.swift:12
    frame #30: 0x0000000108e8292d libdyld.dylib`start + 1
(lldb)
```

A quick look at the backtrace shows the thread executing in a dozen or so system support functions and methods in CoreFoundation and UIKit before finally beginning execution in our ‘Tbl_MixedDebug’ project. This is typical; a *lot* of system code operates on behalf of the application and most of it will be unfamiliar.

Except in very unusual cases, you can ignore everything except the very top of the stack where execution enters your application.

When the backtrace reaches your application source modules the frames will include line numbers which allow you to see exactly where method and function calls occurred. Using this information you can reconstruct exactly how execution reached the current breakpoint.

Times when you find yourself inexplicably looking at a screen of assembly instructions usually means you've somehow "popped" up to the frame for system code for which LLDB has no symbol information, so it does the best it can and displays instructions.

Being able to interpret thread backtraces is an important debugging skill, both for understanding your current debugging session, and because the information for "crash-reports" will almost always be in the form of a backtrace.

Crash reports about publicly released apps from services such as 'TestFlight' and 'Hockey App' will typically be in the form of an "Exception Backtrace" such as the following from Hockey:

```
Application Specific Information:
*** Terminating app due to uncaught exception 'NSRangeException', reason: '*** -[__NSArray0 objectAtIndex:]: index 0
beyond bounds for empty NSArray'

Last Exception Backtrace:
0   CoreFoundation                      0x0000000182a32e38 __exceptionPreprocess + 124
1   libobjc.A.dylib                     0x0000000182097f80 objc_exception_throw + 52
2   CoreFoundation                      0x00000001829a9a6c -[__NSArray0 objectAtIndex:] + 108
3   HuntStand                           0x0000000100062bd4 -[MapViewController loadLinesAndShapes]
(MapViewController.m:262)
4   HuntStand                           0x0000000100063898 -[MapViewController loadPinsOnMap] (MapViewController.m:349)
5   HuntStand                           0x0000000100066bc0 -[MapViewController viewDidLoad] (MapViewController.m:806)
6   UIKit                               0x0000000187b7cb40 -[UIViewController loadViewIfRequired] + 992
7   UIKit                               0x0000000187b94fd0 -[UIViewController __viewWillAppear:] + 128
8   UIKit                               0x0000000187d2fd10 -[UINavigationController _startCustomTransition:] + 1048
```

The key details of note for this stack trace is that the application is called 'HuntStand' and the last line of code that executed within the application was 'MapViewController.m" line 262. The frames above that one in the trace are system exception handling. Only the first one above our code (indicating a problem calling the -objectAtIndex instance method of NSArray) is useful for identifying the source of the crash.

Based on this report, you would expect to find an NSArray object on line 262. The most likely explanation is that the instance is being initialized incorrectly or that something is wrong with how objects are being indexed.

As a general rule, crashing bugs for which you have an exception-backtrace, either in Xcode or from a crash report, are the easiest to identify and fix.

Since what we're discussing are really 'thread backtraces' and 'thread stepping, I should probably go into a little detail about what is meant by "thread".

A thread is the current "*thread of execution*" through a set of instructions. Threads are lightweight units of concurrency within an executable that share state information.

In modern operating systems, such as iOS or MacOS, there is concurrency both between applications (though in iOS, application concurrency is somewhat limited) and within applications. The mechanism for intra-application concurrency is threads.

The output from the 'frame' command or a backtrace is actually the current stack-frame for a single *thread* of execution.

Though it's rare for a developer to be explicitly programming more than a handful of threads, the number of threads working on behalf of the user code, communicating with the OS and spawned in support of drawing operations might be dozens.

LLDB sees all these threads and makes "back-traces" for each of them available in the Xcode "Debug navigator":

Code that you have written will nearly always be on the main thread, 'Thread 1', which is the thread on which you must execute all code that draws or updates the UI.

Unless you are intentionally executing on another thread (by using one of the iOS concurrency libraries) or running as the result of an asynchronous callback (like a networking completion block) you will be executing on Thread 1.

One bit of confusion that often arises is the relation between *queues* and *threads.*

The most common iOS multithreading APIs (GCD and NSOperation) describe their capabilities in terms of queues. "Queues" are a slightly more abstract way of describing a mechanism that is functionally equivalent to threads.

The key points to remember are that "main_queue() = Thread 1 and submitting operations to a single queue is functionally equivalent (in terms of data dependency) to scheduling them on a single thread.

Understanding threads and their relationships to queues is important, but in practice, you're rarely concerned with a thread unless it's executing instructions you've written.

Returning to the "Debug Navigator" for some review: the list of function and method calls listed under 'Thread 1' are a 'stack back-trace' which is a list of the calls that have occurred on a single thread. This defines the current set of stack frames.

Frame '0' where you see "-[ObjcDataSource demoSimpleStuff]" is the current stack frame whose variables we saw when we entered the 'fr v' LLDB command.

The ‘frame’ command also allows examination of other stack frames. We’re currently in stack frame 0. Entering ‘frame select 1’ changes the frame to the -init() method from which -demoSimpleStruff was called.

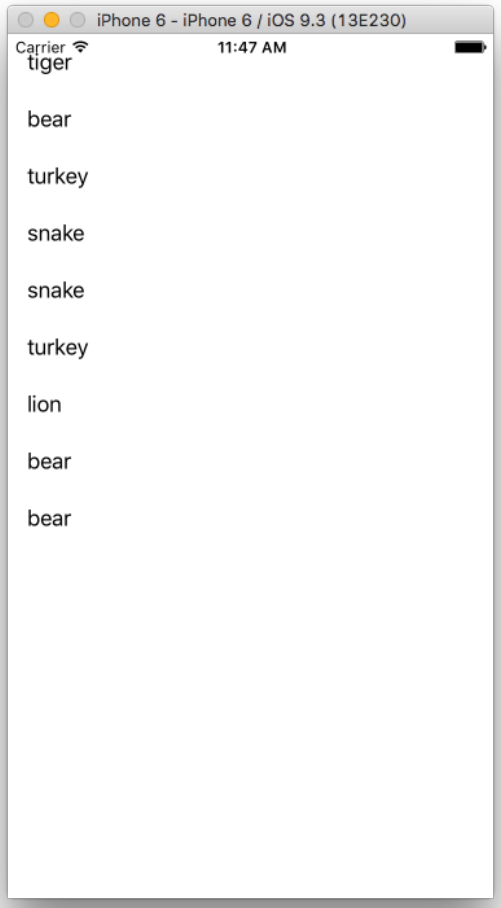
```
(lldb) frame select 1
frame #1: 0x000000010324875d Tbl_MixedDebug`-[ObjcDataSource init](self=0x00007fa59ad44f40,
_cmd="init") + 237 at ObjcDataSource.m:37
   34          //self.ourMenagerie.menagerie = @[@"ant",@"grass-hopper",@"beetle",@5]; //
xcode think its ok, but crashes
   35      }
   36      // demostrate 'di'
-> 37      [self demoSimpleStuff];
   38
   39      return self;
   40  }
(lldb) fr v
(ObjcDataSource *) self = 0x00007fa59ad44f40
(SEL) _cmd = "init"
(int) menagerieSize = 9
(lldb)
```

Note that changing the frame with ‘frame select’ also changed the frame indicator on the left in the debug navigator along with source code selection.

Now that we know a little more about how LLDB works, we can explore more of the capabilities of the tool.

But first we’ll do a quick overview of our demo project.

‘Tbl_MixedDebug’ generates a rudimentary table that displays a random number of randomly selected animals from our “Menagerie”. Running the application will produce something like:



The application was designed less to generate an impressive table than as a test-bed for exploring how types and values are “bridged” between Swift and Objective-C
To that end, the app delegate and tableview controller are Swift modules: AppDelegate.swift and ViewController.swift.
The table-view data-source is an Objective-C object, ObjcDataSource. The symbols in ObjcDataSource.h are made available in ViewController.swift through the bridging header Tbl_MixedDebug-Bridging-Header.h

One additional Swift module, Menagerie.swift provides data to the datasource, ObjcDataSource. The Swift symbols are made available to the objective-C through the auto-generated “Tbl_MixedDbug-Swift.h” file. This header does appear in the project files, but is generated in a temporary build directory which is in the project’s header include path.

Video 4 - The ‘Expression’ command

LLDB’s ‘Expression’ or ‘expr’ command allows you evaluate expressions within the context of the current debug target. This allows you to examine variables in a more flexible manner than ‘frame variable’ which passively examines values. ‘Expr’ allows you to execute descriptive methods, such as an Objective-C object’s implementation of -description. This execution includes participation of the Swift and Objective-C runtime systems.

The most common use of ‘expression’ is:
‘expression -O --’
The ‘-O’ option shows the “object description”. In the case of Objective-C, the -description method will be used. This ‘expression’ command is so useful that it’s been aliased to ‘po’ within LLDB.

‘Expression’ gives you nearly unlimited power to evaluate code within your program context including assignments. You therefore not only have the ability to execute commands with the context of the debugging target, but to alter the state of that target.

Modifying the current program state gives us the power to explore paths of execution that would not otherwise be followed.

I’ll show you a simple example using the animals array in our Menagerie class. This is the array that populates our tableview

We’ll set a breakpoint in the init method of Menagerie.swift after the call to super.init(). If we enter the command ‘fr v possibleAnimals’ we see:

```
(lldb) fr v self.possibleAnimals
([String]) self.possibleAnimals = 5 values {
  [0] = "lion"
  [1] = "tiger"
  [2] = "bear"
  [3] = "snake"
  [4] = "turkey"
}
```

If we then enter the command:

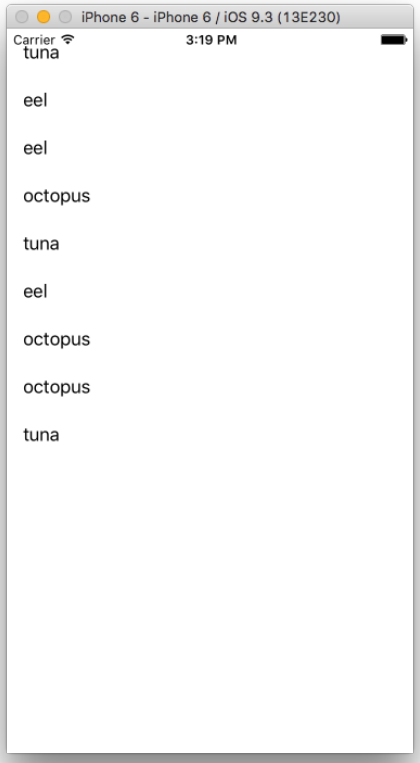
```
(lldb) expr self.possibleAnimals = ["shark","eel","octopus","tuna"]
```

...and inspect self.possibleAnimals again:

```
(lldb) fr v self.possibleAnimals
([String]) self.possibleAnimals = 4 values {
```

```
  [0] = "shark"
  [1] = "eel"
  [2] = "octopus"
  [3] = "tuna"
}
```

We've changed our selection of menagerie animals from land animals to sea creatures.
...then continuing, we see that our table ends up filled with marine life.

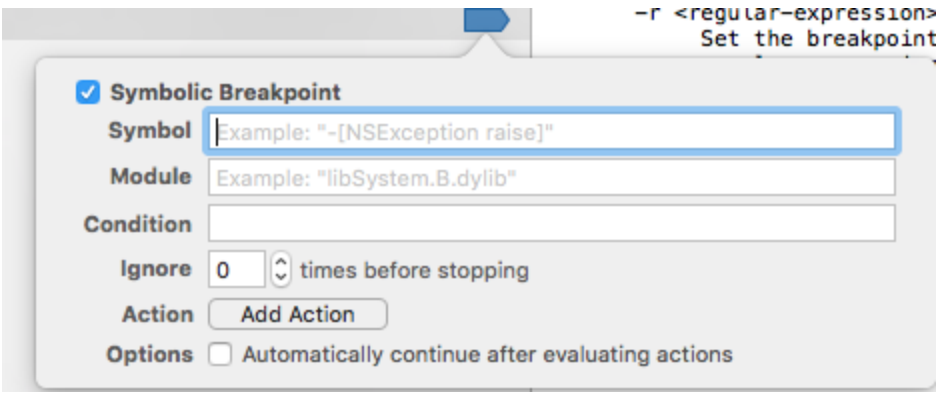


Video 5 - Breakpoints

Since you're already familiar with breakpoints we'll limit the discussion to some of the more advanced features.

Conditions and *Actions* can be added to an existing breakpoint by cntl-clicking or right clicking on the breakpoint within the breakpoint navigator.

Conditional and symbolic breakpoints can be added with a GUI interface by clicking the “plus” icon in the bottom left of the breakpoint navigator. This is the same “plus” button used to add an Objective-C exception breakpoint or a Swift Error breakpoint.



As an example we’ll enter ‘testFunction’ as our symbol (the function from Menagerie.swift we looked at earlier) and add ‘bt’ as our ‘added action’. Running our application, we see execution stops at ‘let x = 5’ within testFunction() and we see a backtrace from this point in the console window.

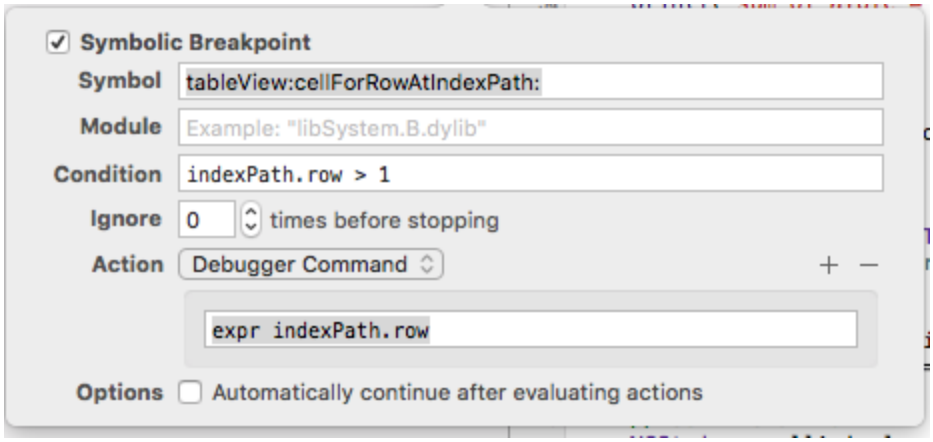
Symbols for breakpoints need not be unique; if we create a symbolic breakpoint for “init” the breakpoint navigator(after clicking the arrow to expand the detail) shows that hundreds of individual breakpoints matching this criterion have been created. Running with this breakpoint actually causes us to break in a number of modules for which LLDB has no symbol information...so we’re back to looking at assembly instructions. That isn’t very useful, so we’ll constrain the selection by editing the breakpoint (R-click, “Edit Breakpoint”) symbol to be “[ObjcDataSource init]” which creates a unique breakpoint.

For an example of a conditional breakpoint we’ll create one for “tableView:cellForRowAtIndexPath:” and add the condition “index.row > 1”.
We’ll also add an “Action” which will be a “Debugger Command”: “expr indexPath.row”

This breakpoint also creates a number of breakpoints since “cellForRowAtIndexPath:” appears many times in the UIKit frameworks. However since none of these are executed in our application, in practice, we have a single breakpoint.

This breakpoint should break each time the tableview requests a cell from the tableview datasource for each row in the table starting with the third (index 2). The breakpoint will then display the row number before stopping.

We need to use ‘expression’ rather than ‘frame variable’ to display the row since we need to call the “-row” method on an NSIndexPath object.



Running the application, we see a table row number (starting with 2) being displayed each execution stops at the breakpoint.

The GUI is convenient for creating breakpoints, but 'breakpoint' on the command-line offers additional options and functionality. For example:

breakpoint set -S someSelector - set break based on selector someSelector
breakpoint set --file somefile.m --line 11 - set on line 11 of somefile.m
breakpoint set -p "*blabla*" - set based on regular expression

Other interesting 'breakpoint set' options:
Thread id: **-t threadnum**
Queue name: **-q queuename**
Stop once only: **-o**
Stop only in the specified langauge: **-L language**
Ignore breakpoint the first N times: **-i N**

Setting breakpoints with the command-line interface requires more typing, but it offers the ability to script commands as we'll see in the next video.

Video 5 - .lldbinit and command files and scripting

Adding "Debugger Commands" to breakpoints through the GUI interface hinted at the ability to script actions using LLDB. This capability is actually quite extensive. The "cellForRowAtIndex" breakpoint we created in the GUI could be created as:

break set -n "tableView:cellForRowAtIndexPath:" -c "(indexPath.row > 1)"
break command add -o "expr indexPath.row"

These commands may be saved as a file, say “~/my_breakpoint”, then loaded whenever it is convenient using the command:

command source “~/my_breakpoint”

Loading LLDB commands from a file opens some interesting possibilities.
For example adding:

breakpoint set --name "Tbl_MixedDebug.Menagerie.init" -N menage_breaks
breakpoint set --name "Tbl_MixedDebug.Menagerie.numberOfAnimals" -N menage_breaks
breakpoint set --name "Tbl_MixedDebug.Menagerie.getAnimalAtIndex" -N menage_breaks
breakpoint disable menage_breaks
command alias menage_on breakpoint enable menage_breaks
command alias menage_off breakpoint disable menage_breaks

to your ~/.lldbinit will give you commands: **menage_on** and **menage_off** for toggling breakpoint for every method in Menagerie.swift

“~/.lldbinit” is a configuration file that LLDB reads on every invocation and will apply the commands to any target opened subsequently.

The “-N” in the script above adds a name or “tag” to each of the breakpoints so that it’s possible to enable and disable them en-masse. Then “command alias” creates the new commands that are then available from the LLDB command line.

Video 6 - More capabilities of LLDB

Watchpoints, sometimes referred to as ‘data breakpoints’ allow you to set an observer on a certain region of memory, such that any changes will trigger the watchpoint and the program will pause just as with a breakpoint.

An example of a watchpoint you might set while stopped in the init method of Menagerie is:

(lldb) watchpoint set variable self.menagerie

Adding animals to the menagerie will then trigger the watchpoint.

Try this out on your own

Since they are keyed on a memory address, application of watchpoints can be a bit tricky. For example, setting a watchpoint on a local variable only makes sense for a single creation of the variable scope since, the next time the variable is instantiated, it is likely to be at a new memory location.

The menagerie array works well as a watchpoint target since, as a property, the location in memory will remain fixed for the life of the Menagerie object.

In addition to customization made possible by aliasing and sourcing stored debugger commands, LLDB offers a Python scripting API.

'Chisel' is an example of what is possible using this API. It is a set of python scripts that help debug iOS applications. Full installation instructions are available in teacher's notes, but involves little more than downloading the scripts with homebrew and add a 'source' command to your ~/.initdb

'pcells' is a chisel command that prints the UItableview cells currently being displayed. If we pause our application after it has displayed the menagerie table view and enter the command at the lldb command prompt we see the cells visible on the screen.

This command can be quite useful for debugging applications where there are a large number of cells in the table that scroll on and off in response to user interaction.

```
(lldb) pcells
<__NSArrayI 0x7fb42b461000>(
<UITableViewCell: 0x7fb42b4455f0; frame = (0 0; 600 44); text = 'tiger'; autoresize = W;
layer = <CALayer: 0x7fb42b412620>>,
<UITableViewCell: 0x7fb42b44e250; frame = (0 44; 600 44); text = 'bear'; autoresize = W;
layer = <CALayer: 0x7fb42b44e830>>,
<UITableViewCell: 0x7fb42b4504a0; frame = (0 88; 600 44); text = 'bear'; autoresize = W;
layer = <CALayer: 0x7fb42b450870>>,
<UITableViewCell: 0x7fb42b7289d0; frame = (0 132; 600 44); text = 'tiger'; autoresize = W;
layer = <CALayer: 0x7fb42b728da0>>,
<UITableViewCell: 0x7fb42b72e0f0; frame = (0 176; 600 44); text = 'bear'; autoresize = W;
layer = <CALayer: 0x7fb42b706960>>
)

(lldb)
```

In addition to 'pcells' chisel contains numerous other commands, including pview

(do demo of pview)

Video 7 - “Poking around” our mixed-language Project/ Wrap-up

What debuggers like LLDB are really good at is poking around in code and figuring out how things really work. The slightly Rube-Goldburgesque structure of the tableview application allows investigation into how Objective-C and Swift symbols and types are *bridged* between the two languages.

Not all symbols and types can be shared between Swift and Objective-C modules, and many that are bridged can only *approximate* in terms of type constraints. As an example, we’ll look at the class variable “menagerie” in the Swift class Menagerie:

If we set and stop at a breakpoint in the ObjcDataSource init() method at the call to -simpleStuff and examine self.ourMenagerie.menagerie:

```
(lldb) po self.ourMenagerie.menagerie
<_TtCs21_SwiftDeferredNSArray 0x7f7ff8483c60>(
snake,
tiger,
turkey,
turkey,
turkey
)
```

..we see the contents of our table, but the most interesting detail is the bridged type:
`_TtCs21_SwiftDeferredNSArray`

This type is referenced as an NSArray* in ObjectDataSource.m, but something curious happens if we add the following assignment:

```
self.ourMenagerie.menagerie = @[@"ant",@"grass-hopper",@"beetle",@5];
```

Xcode has no complaints during compilation since having both NSString and NSNumber types in an NSArray is acceptable, but running the application now generates an EXC_BAD_INSTRUCTION exception.

The top 6 frames of the backtrace from the exception tells us what the problem is:

```
(lldb) bt
* thread #1: tid = 0x892a82, 0x000000010aa9b940 libswiftFoundation.dylib`Swift._arrayForceCast
<A, B> (Swift.Array<A>) -> Swift.Array<B> + 752, queue = 'com.apple.main-thread', stop reason
= EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)
```

```
    frame #0: 0x000000010aa9b940 libswiftFoundation.dylib`Swift._arrayForceCast <A, B>
(Swift.Array<A>) -> Swift.Array<B> + 752
    frame #1: 0x000000010aa873fa libswiftFoundation.dylib`static (extension in
Foundation):Swift.Array._forceBridgeFromObjectiveC (__ObjC.NSArray, result : inout
Swift.Optional<Swift.Array<A>>) -> () + 282
    frame #2: 0x000000010aa87267 libswiftFoundation.dylib`Foundation._convertNSArrayToArray
<A> (Swift.Optional<__ObjC.NSArray>) -> Swift.Array<A> + 55
    frame #3: 0x00000001081fe2dd Tbl_MixedDebug`@objc Menagerie.menagerie.setter + 61 at
Menagerie.swift:0
  * frame #4: 0x00000001081fc4c4 Tbl_MixedDebug`-[ObjcDataSource
init](self=0x00007fa983d25b10, _cmd="init") + 420 at ObjcDataSource.m:35
    frame #5: 0x00000001081fd810 Tbl_MixedDebug`@nonobjc ObjcDataSource.init() ->
ObjcDataSource + 16 at ViewController.swift:0
    frame #6: 0x00000001081fd6b4 Tbl_MixedDebug`ObjcDataSource.__allocating_init() ->
ObjcDataSource + 68 at ViewController.swift:0
```

A crash occurs as the Swift implementation of the Foundation framework attempts to cast our NSArray to a Swift Array that is constrained to have contents of a uniform type.

Objective-C is fine with an NSArray containing NSStrings and NSNumbers, but Swift Arrays are not so accepting.

Swift-Objective-C types bridging works fairly well, but there are some gotchas, and LLDB is an excellent tool for exploring them.

I encourage you to experiment more with bridged types and the rest of the demo project.

Hopefully this workshop provided some insights into how LLDB works and gave you a few new ideas for how to get the most out of the Xcode debugger.

Teacher's Notes

LLDB tutorial LLVM website
Dancing in the Debugger (objc.io)
Getting Started with LLDB (Apple)
LLDB Quick Start Guide (Apple)
About the Breakpoint Navigator (Apple)
How Debuggers Work
Call Stack Wikipedia
Understanding the Stack
Chisel - Additional LLDB commands implemented in Python
LLDB debugging tips - swifting.io

Swift and Objective-C Interoperability

WWDC 2015 - What's new in LLDB
WWDC 2014 - Introduction to LLDB and the Swift REPL
WWDC 2014 - Advanced Swift Debugging in LLDB
WWDC 2013 - Advanced Debugging With LLDB
WWDC 2012 - Debugging With LLDB