Quick Tour of LLDB

You use the LLDB debugger to run programs step-by-step, set breakpoints, and inspect and modify program state.

You can get a basic understanding of the debugger's capabilities by interacting with a small example.

The following Swift code defines a `Greeter` type that is responsible for greeting individuals. The `Greeter` type keeps track of its acquaintances, and adjusts its greeting on subsequent encounters.

```
class Greeter {
    private var acquaintances: Set<String> = []

    func hasMet(personNamed name: String) -> Bool {
        return acquaintances.contains(name)
    }

    func greet(personNamed name: String) {
        if hasMet(personNamed: name) {
            print("Hello again, \(name)!")
        } else {
            acquaintances.insert(name)
            print("Hello, \(name). Nice to meet you!")
        }
    }
}

let greeter = Greeter()

greeter.greet(personNamed: "Anton")
greeter.greet(personNamed: "Mei")
greeter.greet(personNamed: "Anton")
```

If you create a file named `Greeter.swift` with the code above and run the `swiftc` command, passing the filename as a command-line argument along with the `-g` option to generate debug information, an executable named `Greeter` is created in the current directory.

```
$ swiftc -g Greeter.swift
$ ls
Greeter.dSYM
Greeter.swift
Greeter*
```

Running the `Greeter` executable produces the following output:

```
$ ./Greeter
Hello, Anton. Nice to meet you!
Hello, Mei. Nice to meet you!
Hello again, Anton!
```

To run the `Greeter` program through the LLDB debugger, pass it as a command-line argument to the `lldb` command.

```
$ lldb Greeter
(lldb) target create "Greeter"
Current executable set to 'Greeter' (x86_64).
```

This command starts an interactive console that allows you to run LLDB commands to interact with the program.

Note: Type `help` into an LLDB prompt to bring up extensive documentation about commands. See also Using Command-Line Help.

Related Chapter: Understanding LLDB Command Syntax

Set a breakpoint on line 18 with the `breakpoint set` (`b`) command, passing the `--line-number` (`-l`) option with the line number as its value, to have the debugger stop after the `Greeter` type declaration.

```
(lldb) breakpoint set --line 18
Breakpoint 1: where = Greeter`main + 70 at Greeter.swift:18, address =
0x0000000100001996
```

Set another breakpoint using the `breakpoint set` (`b`) command, passing the `--name` (`-n`) option with the function name `greet` as its value, to have the debugger stop anytime the `greet(personNamed:)` method is called.

```
breakpoint set --name greet
Breakpoint 2: where = Greeter`Greeter.Greeter.greet (personNamed : Swift.String) ->
() + 27 at Greeter.swift:9, address = 0x0000000100001bab
```

Related Chapter: Managing Breakpoints

If you run the process with the `process launch` (`run` or `r`) command, the process stops at the breakpoint on line 18.

Note: Starting a debugging session for a program doesn't automatically run that program. This allows you to set breakpoints that may trigger shortly after launch.

```
(lldb) process launch
Process 97209 launched: 'Greeter' (x86_64)
Process 97209 stopped
* thread #1: tid = 0x1288be3, 0x0000000100001996 Greeter`main + 70 at
Greeter.swift:18, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100001996 Greeter`main + 70 at Greeter.swift:18
   15          }
   16      }
   18
-> 18      let greeter = Greeter()
   19
   20      greeter.greet(personNamed: "Anton")
```

```
   21      greeter.greet(personNamed: "Mei")
```

Enter the `thread step-over` (`next` or `n`) command to advance the process to the the next function call on line 20.

```
(lldb) thread step-over
Process 97209 stopped
* thread #1: tid = 0x1288be3, 0x00000001000019bd Greeter`main + 109 at
Greeter.swift:20, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000019bd Greeter`main + 109 at Greeter.swift:20
   17
   18      let greeter = Greeter()
   19
-> 20      greeter.greet(personNamed: "Anton")
   21      greeter.greet(personNamed: "Mei")
   22      greeter.greet(personNamed: "Anton")
```

Use the `thread step-in` (`step` or `s`) command to have the debugger step into the `greet(personNamed:)` method.

```
(lldb) thread step-in
Process 97209 stopped
* thread #1: tid = 0x1288be3, 0x0000000100001bab Greeter`Greeter.greet(name="Anton",
self=0x0000000100606b10) -> () + 27 at Greeter.swift:9, queue = 'com.apple.main-
thread', stop reason = step in
    frame #0: 0x0000000100001bab Greeter`Greeter.greet(name="Anton",
self=0x0000000100606b10) -> () + 27 at Greeter.swift:9
   6           }
   7
   8           func greet(personNamed name: String) {
-> 9               if hasMet(personNamed: name) {
   10                  print("Hello again, \(name)!")
   11              } else {
   12                  acquaintances.insert(name)
```

Enter the `thread step-over` (`next` or `n`) command again, passing the `--count` (`-c`) option with the value `4`, to skip to the last line of the else branch in the `greet(personNamed:)` method.

```
(lldb) thread step-over --count 4
Process 97209 stopped
* thread #1: tid = 0x1288be3, 0x0000000100001e0c Greeter`Greeter.greet(name="Mei",
self=0x0000000100606b10) -> () + 636 at Greeter.swift:13, queue = 'com.apple.main-
thread', stop reason = step over
    frame #0: 0x0000000100001e0c Greeter`Greeter.greet(name="Mei",
self=0x0000000100606b10) -> () + 636 at Greeter.swift:13
   10                  print("Hello again, \(name)!")
   11              } else {
   12                  acquaintances.insert(name)
-> 13                  print("Hello, \(name). Nice to meet you!")
   14              }
   15          }
   16      }
```

Related Chapter: Controlling Process Execution

Use the `thread backtrace` (`backtrace` or `bt`) command to show the frames leading to the current function being called.

```
(lldb) thread backtrace

* thread #1: tid = 0x1288be3, 0x0000000100001a98
Greeter`Greeter.hasMet(name="Anton", self=0x0000000101200190) -> Bool + 24 at
Greeter.swift:5, queue = 'com.apple.main-thread', stop reason = step in

    frame #0: 0x0000000100001a98 Greeter`Greeter.hasMet(name="Anton",
self=0x0000000101200190) -> Bool + 24 at Greeter.swift:5

  * frame #1: 0x0000000100001be4 Greeter`Greeter.greet(name="Anton",
self=0x0000000101200190) -> () + 84 at Greeter.swift:9

    frame #2: 0x00000001000019eb Greeter`main + 155 at Greeter.swift:20

    frame #3: 0x00007fff949d05ad libdyld.dylib`start + 1

    frame #4: 0x00007fff949d05ad libdyld.dylib`start + 1
```

Use the `frame variable` (`f v`) command with no arguments to see all of the variables in the current stack frame.

```
(lldb) frame variable

(String) name = "Anton"

(Greeter.Greeter) self = 0x0000000100502920 {

  acquaintances = ([0] = "Anton")

}
```

Notice that the `acquaintances` property is populated with the name `Anton`, as it was inserted on the previous line.

Using the `expression` (`e`) command, you can change the state of the `acquaintances` property to alter the final output of the program.

```
(lldb) expression -- acquaintances.insert("Mei")

(lldb) expression -- acquaintances.remove("Anton")

(String?) $R1 = "Anton"
```

Disable the breakpoint on the `greet(personNamed:)` method using the `breakpoint disable command`, passing the breakpoint ID `2` as the argument. Then, enter the `process continue` (`continue` or `c`) command to resume execution of the process.

```
(lldb) breakpoint disable 2

1 breakpoints disabled.

(lldb) process continue

Resuming thread 0x12f1d19 in process 97209

Process 97209 resuming

Hello again, Mei!

Hello, Anton. Nice to meet you!

Process 97209 exited with status = 0 (0x00000000)
```

Notice that the output written to the console is different from the output of the program when run previously, without the debugger.

Related Chapter: Examining the Call Stack