lldb cheat sheet

Execution Commands

start lld (prefix with xcrun on os x)
```
>lldb [program.app]
>lldb -- program.app arg1
```

load program
```
>file program.app
```

run program
```
>process launch [-- args]
>run [args]
```

set arguments
```
>settings set target.run-args 1
```

launch process in new terminal
```
>process launch --tty -- <args>
```

set env variables
```
>settings set target.env-vars DEBUG=1
```

remove env variables
```
>settings remove target.env-vars
DEBUG
```

show program arguments
```
>settings show target.run-args
```

set env variable and run
```
>process launch -v DEBUG=1
```

attach to process by PID
```
>process attach --pid 123
```

attach to process by name
```
>process attach --name a.out [--
waitfor]
```

attach to remote gdb on eorgadd
```
>gdb-remote eorgadd:8000
```

attach to gdb server on localhost
```
>gdb-remote 8000
```

attach to remote Darwin kernel in kdp mode
```
>kdp-remote eorgadd
```

source level single step
```
>thread step-in
>step
>s
```

source level setop over
```
>thread step-over
>next
>n
```

instruction level single step
```
>thread step-inst
>si
```

instruction level single step over
```
>thread step-inst-over
>ni
```

step out of the currently selected frame
```
>thread step-out
>finish
```

Return from currently frame, with return value
```
>thread return [RETURN EXPRESSION]
```

Backtrace and disassemble every time you stop
```
>target stop-hook add
>bt
>disassemble --pc
>DONE
```

run until line 12 or end of frame
```
>thread until 12
```

Breakpoint Commands

set breakpoint at all functions named main
```
>breakpoint set --name main
>br s -n main
>b main
```

set breakpoint in file test.c line 12
```
>breakpoint set --file test.c --line
12
>br s -f test.c -l 12
>b test.c:12
```

set breakpoint at all C++ methods with name main
```
>breakpoint set --method main
>br s -M main
```

set breakpoint at ObjC function
```
>breakpoint set --name "-[NSString
stringWithFormat:]"
>b -[NSString stringWithFormat:]
```

set breakpoint at all ObjC functions whose selector is count
```
>breakpoint set --selector count
>br s -S count
```

set breakpoint by regular expression function name
```
>breakpoint set --func-regex print.*
```

ensure that breakpoints by file and line work (c/cpp/objc)
```
>settings set target.inline-
breakpoint-strategy always
>br s -f foo.c -l 12
```

set a breakpoint by regular expression on source file contents
```
>breakpoint set --source-pattern
regular-expression --file SourceFile
>br s -p regular-expression -f file
```

set conditional breakpoint
```
>breakpoint set --name foo --
condition '(int)strcmp(y,"hello") ==
0'
>br s -n foo -c
'(int)strcmp(y,"hello") == 0'
```

list breakpoints
```
>breakpoint list
>br l
```

delete a breakpoint
```
>breakpoint delete 1
>br del 1
```

Watchpoint Commands

set watchpoint on variable when written to
```
>watchpoint set variable global_var
>wa s v global_var
```

set watchpoint on memory of pointer size
```
>watchpoint set expression --
0x123456
>wa s e -- 0x123456
```

set watchpoint on memory of custom size
```
> watchpoint set expression –x
byte_size -- 0x123456
> wa s e –x byte_size -- 0x123456
```

set a condition on a watchpoint
```
>watch set var global
>watchpoint modify -c '(global==5)'
```

list watchpoints
```
>watchpoint list
>watch l
```

delete a watchpoint
```
>watchpoint delete 1
>watch del 1
```

Examining Variables

show arguments and local variables
```
>frame variable
>fr v
```

show local variables
```
>frame variable --no-args
>fr v -a
```

show contents of variable bar
```
>frame variable bar
>fr v bar
>p bar
```

show contents of var bar formatted as hex
```
>frame variable --format x bar
>fr v -f x bar
```

show contents of global variable baz
```
>target variable baz
>ta v baz
```

show global/static variables in current file
```
>target variable
>ta v
```

show argc and argv every time you stop
```
>target stop-hook add --one-liner
"frame variable argc argv"
>ta st a -o "fr v argc argv"
>display argc
>display argv
```

display argc and argv when stopping in main
```
>target stop-hook add --name main --
one-liner "frame variable argc argv"
>ta st a -n main -o "fr v argc argv"
```

display *this when in class MyClass
```
>target stop-hook add --classname
MyClass --one-liner "frame variable
*this"
>ta st a -c MyClass -o "fr v *this"
```

Evaluating Expressions

evaluate expression (print alias possible as well)
```
>expr (int) printf ("Print nine:
%d.", 4 + 5)
>print (int) printf ("Print nine:
%d.", 4 + 5)
```

using a convenience variable
```
>expr unsigned int $foo = 5
```

print the ObjC description of an object
```
>expr -o -- [SomeClass
returnAnObject]
>po [SomeClass returnAnObject]
```

print dynamic type of expression result
```
>expr -d 1 -- [SomeClass
returnAnObject]
>expr -d 1 –
someCPPObjectPtrOrReference
```

set dynamic type printing as default

```
>settings set target.prefer-dynamic
run-target
```

calling a function with a breakpoint

```
>expr -i 0 --
function_with_a_breakpoint()
```

calling a function that crashes

```
expr -u 0 -- function_which_crashes()
```

Examining Thread State

show backtrace (current thread)

```
>thread backtrace
>bt
```

show backtrace for all threads

```
>thread backtrace all
>bt all
```

backtrace the first 5 frames of current thread

```
>thread backtrace -c 5
>bt 5 (lldb-169 and later)
>bt –c 5 (lldb-168 and later)
```

select a different stack frame by index

```
>frame select 12
>fr s 12
>f 12
```

show frame information

```
>frame info
```

select stack frame the called current frame

```
>up
>frame select --relative=1
```

select stack frame that is called by current frame

```
>down
>frame select --relative=-1
>fr s -r-1
```

select different frame using relative offset

```
>frame select --relative 2
>fr s -r2
>frame select --relative -3
>fr s -r-3
```

show general purpose registers

```
>register read
```

write 123 to register rax

```
>register write rax 123
```

skip 8 bytes using with program counter

```
>register write pc `$pc+8`
```

show general purpose registers as signed decimal

```
>register read --format i
>re r -f i
>register read/d
```

show all registers in all register threads

```
>register read --all
>re r -a
```

show registers rax, rsp, rbp

```
register read rax rsp rbp
```

show register rax with binary format

```
>register read --format binary rax
```

read memory from 0xbffff3c0 and show 4 hex uint32_t values

```
>memory read --size 4 --format x --
count 4 0xbffff3c0
>me r -s4 -fx -c4 0xbffff3c0
>x -s4 -fx -c4 0xbffff3c0
>memory read/4xw 0xbffff3c0
>x/4xw 0xbffff3c0
>memory read --gdb-format 4xw
0xbffff3c0
```

read memory starting at the expression "argv[0]"

```
>memory read `argv[0]`
>memory read --size `sizeof(int)`
`argv[0]`
```

read 512 bytes from address 0xbffff3c0 and save results to a local file

```
>memory read --outfile /tmp/mem.txt -
-count 512 0xbffff3c0
>me r -o/tmp/mem.txt -c512 0xbffff3c0
>x/512bx -o/tmp/mem.txt 0xbffff3c0
```

save binary memory data starting at 0x1000 and ending at 0x2000 to file

```
>memory read --outfile /tmp/mem.bin -
-binary 0x1000 0x2000
>me r -o /tmp/mem.bin -b 0x1000
0x2000
```

get information about specific heap allocation (Mac OS X only)

```
>command script import
lldb.macosx.heap
>process launch --environment
MallocStackLogging=1 -- [ARGS]
>malloc_info --stack-history
0x10010d680
```

get information about specific heap allocation and cast result to dynamic type that can be deduced (Mac OS X only)

```
>command script import
lldb.macosx.heap
>malloc_info --type 0x10010d680
```

find all heap blocks that contain pointer specified by an expression EXPR (Mac OS X only)

```
>command script import
lldb.macosx.heap
>ptr_refs EXPR
```

find all heap blocks that contain a C string anywhere in the block (Mac OS X only)

```
>command script import
lldb.macosx.heap
>cstr_refs CSTRING
```

disassemble current function for current frame

```
>disassemble –frame
>di -f
```

disassemble any functions named main

```
>disassemble --name main
>di -n main
```

disassemble address range

```
>disassemble --start-address 0x1eb8 -
-end-address 0x1ec3
>di -s 0x1eb8 -e 0x1ec3
```

disassemble 20 instructions from start address

```
>disassemble --start-address 0x1eb8 -
-count 20
>di -s 0x1eb8 -c 20
```

show mixed source and disassembly for the current function

```
>disassemble --frame –mixed
>di -f -m
```

disassemble the current function for the current frame and show the opcode bytes

```
>disassemble --frame –bytes
>di -f -b
```

disassemble the current source line for the current frame

```
>disassemble --line
>di -l
```

Executable and Shared Library Query Commands

list the main executable and all dependent shared libraries

```
>image list
```

look up information for a raw address in the executable or any shared libraries

```
>image lookup --address 0x1ec4
>im loo -a 0x1ec4
```

look up functions matching a regular expression in a binary

```
>image lookup -r -n <FUNC_REGEX>
(debug symbols)
>image lookup -r -s <FUNC_REGEX>
(non-debug syms)
```

find full source line information

```
>image lookup -v --address 0x1ec4
(look for entryline)
```

look up information for an address in a.out only

```
>image lookup --address 0x1ec4 a.out
>im loo -a 0x1ec4 a.out
```

look up information for a type Pointer by name

```
>image lookup --type Point
>im loo -t Point
```

dump all sections from the main executable and any shared libraries

```
>image dump sections
```

dump all sections in the a.out module

```
>image dump sections a.out
```

dump all symbols from the main executable and any shared libraries

```
>image dump symtab
```

dump all symbols in a.out and liba.so

```
>image dump symtab a.out liba.so
```

Miscellaneous

echo text to the screen

```
>script print "Here is some text"
```

remap source file pathnames for the debug session (e.g. if program was built on another PC)

```
>settings set target.source-map
/buildbot/path /my/path
```