

デバッグ lldb の 使い方入門

東京工業大学 情報工学系 権藤克彦

まえがき

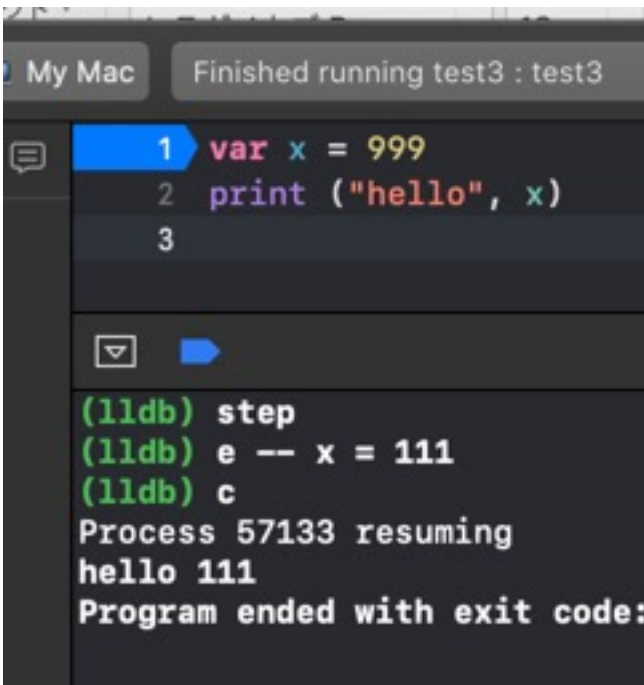
- 実行例を多く載せてます。 皆さん自身で実際に実行して確かめて下さい。
- デバッガの使い方はつまみ食いOK。 ちょっとずつ知識増やせばOK.
- 本書ではコマンドラインでの使い方を解説してます。
 - 統合開発環境 (IDE) でも大して変わらないけど
 - やさしめにしたつもり
- lldbの出力例は簡単のため一部省略しています.
- macOSなら, lldbのインストールは不要 (のはず)
 - 端末で lldb 初回起動時に Xcode インストールを促されたらやって下さい
- 本書で使用した環境
 - macOS Catalina (10.15.1)
 - Apple clang version 11.0.0 (clang-1100.0.33.12)
 - Target: x86_64-apple-darwin19.0.0
 - lldb-1100.0.30.6

デバッガとは

- デバッグを支援するツール（ソフトウェア）
- 代表的なデバッガの例：lldb, gdb, jdb
- EclipseやXcodeなどの統合開発環境(IDE)にもデバッガは入ってる
- 主な機能
 - **実行の一時停止**
 - ブレークポイントを設定して停止場所を指定
 - **ステップ実行**
 - ちょっとずつプログラムを実行して、実行を一時停止させる
 - **実行状態の表示**
 - 変数の値，現在の行番号，スタックトレースなどを表示
 - printfデバッグよりもずっと効率的！再コンパイル不要だから。
 - **実行状態の変更，実行途中での実行**
 - 変数への代入や関数呼び出しで「ここでこう実行させたら」を試せる

lldb とは

- いろいろ高性能なデバッガ
- サポート言語
 - C, C++, Objective-C, Swift
- macOS上のXcodeのデフォルトデバッガ
 - iOSアプリ開発でも役に立つ！
- サポート環境
 - macOS, iOS, Linux, FreeBSD, Windows
- オープンソース
 - Apache License, version 2.0 (llvmによる例外あり)
 - <https://llvm.org/docs/DeveloperPolicy.html#new-llvm-project-license-framework>



The screenshot shows the Xcode IDE interface. At the top, a status bar indicates 'My Mac' and 'Finished running test3 : test3'. Below this, a Swift code editor displays three lines of code: `1 var x = 999`, `2 print ("hello", x)`, and `3`. The first line is highlighted with a blue selection bar. Below the code editor, the lldb console is visible, showing the following output: `(lldb) step`, `(lldb) e -- x = 111`, `(lldb) c`, `Process 57133 resuming`, `hello 111`, and `Program ended with exit code:`.

一次情報

- lldb公式ページ
 - <http://lldb.llvm.org/>
- コマンド一覧（公式）
 - <https://lldb.llvm.org/use/map.html>
- チュートリアル（公式）
 - <https://lldb.llvm.org/use/tutorial.html>

実行例（C言語）

lldbの起動と終了

```
// hello.c
#include <stdio.h>
int main ()
{
    printf ("hello, world\n");
}
```

赤字がプログラマが入力する部分
青字は説明
黒字はlldbの出力

```
% gcc -g hello.c          デバッグオプション -g をつけてコンパイル
% lldb ./a.out            lldbデバッガの起動
(lldb) target create "./a.out"
Current executable set to './a.out' (x86_64).
(lldb) run                実行開始
Process 62148 launched: '/tmp/a.out' (x86_64)
hello, world              hello.c中のprintfの出力
Process 62148 exited with status = 0 (0x00000000)
(lldb) quit               lldbの終了
Quitting LLDB will kill one or more processes. Do you really
want to proceed: [Y/n] y  ほんとに終了？と聞かれるのでyを入力
%
```

コマンド引数argvを与えて実行

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    printf ("%d¥n", argc);
}
```

```
% gcc -g foo.c
% lldb ./a.out
(lldb) run 1 2 3 4      コマンド引数を与えて実行開始
Process 65815 launched: '/tmp/a.out' (x86_64)
5
Process 65815 exited with status = 0 (0x00000000)
(lldb) quit
(lldb)
```


標準入出力の切り替え

```
#include <stdio.h>
int main ()
{
    int c;
    while ((c = getchar ())!=EOF) {
        putchar (c);
    }
}
```

```
% lladb ./a.out
```

```
(lladb) process launch -i in.txt -o out.txt
```

-i で標準入力をファイル in.txt に切り替えた。

-o で標準出力をファイル out.txt に切り替えた

-e で標準エラー出力も切り替えられる

runは process launchの省略形だけど、ここはrunではダメ

```
(lladb) quit
```

実行例：Segmentation faultの原因を探る

細かいコマンドの意味は
気にせず実行を試してみて

```
#include <stdio.h>
int main ()
{
    int *p = (int *)0xFFFFFFFFFFFFFFFF;   アクセスダメそうなアドレスを代入
    printf ("%d¥n", *p);   ダメそうなアドレスの中身にアクセス（参照）
}
```

```
% lladb ./a.out
(lladb) run
* thread #1, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=1, address=0xffffffffffffffff) ←
    frame #0: 0x0000000100000f67 a.out`main at foo.c:5:22
    アドレス0xffffffffffffffffへの不正アクセスの実行時エラーで実行停止
    4         int *p = (int *)0xFFFFFFFFFFFFFFFF;
-> 5         printf ("%d¥n", *p);   実行停止した行はここ
    6     }
(lladb) print p
(int *) $0 = 0xffffffffffffffff
(lladb)
```

このアドレス
と一致

ポインタ変数pの値を表示
ポインタ変数pの値が原因でした

短縮コマンド

- lldbのコマンドには短縮形がある。覚えると入力が楽。

(lldb) breakpoint set --name main	main関数にブレークポイントをセット
(lldb) breakpoint set -n main	上の短縮形 (短縮形のオプション)
(lldb) break set -n main	上の短縮形
(lldb) b main	上の短縮形 (-nも不要)

- help で短縮形が分かることも (全部は載ってない)

```
(lldb) help breakpoint  
Commands for operating on breakpoints (see 'help b' for shorthand.)
```

- help で短縮コマンドから正規コマンドが分かる

```
(lldb) help b           bは実は_regexp-breakの短縮形だったw  
'b' is an abbreviation for '_regexp-break'
```

実行例：変数の値を表示

```
int main ()
{
    int x = 10;
    x += 3;
    x += 3;
    return x;
}
```

```
% lladb ./a.out
```

```
(lldb) b main
```

main関数にブレークポイントを設定

```
(lldb) run
```

実行開始

```
1      int main ()
```

```
2      {
```

```
-> 3          int x = 10;
```

3行目の実行直前で実行停止

```
4          x += 3;
```

```
(lldb) step
```

1行だけ実行（ステップ実行）

```
1      int main ()
```

```
2      {
```

```
3          int x = 10;
```

```
-> 4          x += 3;
```

この行の実行直前で実行停止

```
(lldb) print x
```

変数xの値を表示（結果は10）

```
(int) $0 = 10
```

```
(lldb) step
```

1行だけ実行（ステップ実行）

```
2      {
```

```
3          int x = 10;
```

```
4          x += 3;
```

```
-> 5          x += 3;
```

この行の実行直前で実行停止

```
(lldb) print x
```

変数xの値を表示（結果は13）

```
(int) $1 = 13
```

実行例

n の代わりに
\$arg1 でもOK

- 条件でブレーク，実行位置を表示

```
#include <stdio.h>
int fact (int n) {
    if (n <= 0) return 1;
    else return n * fact (n - 1);
}
int main ()    よくある階乗の計算
{
    printf ("%d\n", fact (5));
}
```

```
% lladb ./a.out
(lladb) break set -name fact -c "n==0"
(lladb) run    ↑ nが0の時に関数factを停止
      2      int fact (int n) {
-> 3          if (n <= 0) return 1;  nが0の時に関数factが実行停止された
      4          else return n * fact (n - 1);
(lladb) print n
(int) $6 = 0  nの値を確認すると確かに0になっている
(lladb) bt     バックトレース (main関数からここまでの関数呼び出し関係を表示)
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 4.1
  * frame #0: 0x0000000100000f1b a.out`fact(n=0) at fact.c:3:11
    frame #1: 0x0000000100000f44 a.out`fact(n=1) at fact.c:4:21
    frame #2: 0x0000000100000f44 a.out`fact(n=2) at fact.c:4:21
    frame #3: 0x0000000100000f44 a.out`fact(n=3) at fact.c:4:21
    frame #4: 0x0000000100000f44 a.out`fact(n=4) at fact.c:4:21
    frame #5: 0x0000000100000f44 a.out`fact(n=5) at fact.c:4:21
    frame #6: 0x0000000100000f72 a.out`main at fact.c:8:21
```

実行例

- 変数や式の値変更の監視 (watchpoint)

```
int main ()
{
    int x, y, z;
    x = 10;
    y = 20;
    z = 30;
}
```

```
% lladb ./a.out
```

```
(lldb) b main
```

```
(lldb) run
```

```
(lldb) watchpoint set variable z ← 変数zにウォッチポイントをセット
```

```
Watchpoint created: Watchpoint 1: addr = 0x7fffeefbfff774 size = 4 state =
enabled type = w
```

```
declare @ '/tmp/foo.c:4'
```

```
watchpoint spec = 'z'
```

```
new value: 32766
```

-w read の指定で値参照の監視も可能

```
(lldb) continue 実行再開
```

ハードウェア機能のため監視できる数に制限あり

```
old value: 32766
```

```
new value: 30
```

```
* thread #1, queue = 'com.apple.main-thread', stop reason = watchpoint 1
```

```
frame #0: 0x00000000100000fab a.out`main at foo.c:8:1
```

```
6         y = 20;
```

```
7         z = 30;
```

zへの代入直後で停止

```
-> 8     }
```

実行例

- 代入（実行途中での変数値の変更）

```
#include <stdio.h>
int main () {
    int x = 1, n = 0;
    while (x) { // 無限ループ
        n++;
    }
    printf ("hello, world¥n");
}
```

```
% lladb ./a.out
```

```
(lldb) run
```

```
Process 65107 launched: '/tmp/a.out' (x86_64)
```

CTRL-C 無限ループするのでコントロールキーを押しながらCを押し、強制中断

```
Process 65107 stopped
```

```
5         while (x) {
-> 6             n++;
7         }
```

```
(lldb) expr x=0 変数xに0を代入.
```

```
(int) $0 = 0
```

```
(lldb) continue 実行を再開
```

```
Process 65107 resuming
```

```
hello, world 無事にループを抜けて実行終了
```

```
Process 65107 exited with status = 0 (0x00000000)
```

```
(lldb)
```

ほぼ同じ

expr と print の違い

-- はオプションの終わりを表す特別なオプション.
他のオプション使用時に必要

- expr は expression の短縮コマンド. オプションを指定可能
- p と print は同じで, expression -- の短縮コマンド. オプション不可

```
(lldb) expression 16      オプションが無ければ -- は不要
(int) $1 = 16
(lldb) expression -format x -- 16  16進数形式の表示オプションを
(int) $2 = 0x00000010             指定したので -- が必要になる
(lldb) expression -format x 16    --を忘れると怒られる
error: use of undeclared identifier 'format'
(lldb) expr -format x 16          exprでも同じく, --を忘れると怒られる
error: use of undeclared identifier 'format'
(lldb) print 16
(int) $5 = 16
(lldb) expr -format x -- 16      expr はオプション (-format x) を指定可能
(int) $7 = 0x00000010
(lldb) print -format x -- 16    print はオプション (-format x) を指定不可
error: use of undeclared identifier 'format'
```


実行例

- 関数を呼び出す
(副作用があってもOK)

```
% lladb ./a.out
```

```
(lldb) b main
```

```
(lldb) run
```

```
6 int main ()
```

```
7 {
```

```
-> 8 printf ("%d¥n", fact (5));
```

```
9 }
```

```
(lldb) expr fact (0)
```

fact(0)を呼び出す。結果は1.

```
(int) $0 = 1
```

```
(lldb) expr fact (3)
```

fact(3)を呼び出す。結果は6.

```
(int) $1 = 6
```

```
(lldb) expr printf ("hello¥n")
```

ライブラリ関数の呼び出しも可能.

```
hello
```

```
(int) $2 = 6
```

```
(lldb)
```

```
#include <stdio.h>
int fact (int n) {
    if (n <= 0) return 1;
    else return n * fact (n - 1);
}
int main ()    よくある階乗の計算
{
    printf ("%d¥n", fact (5));
}
```

実行例（x86-64アセンブリ言語）

アセンブリコード例

foo.s

- printfで999を出力するだけ

```
% gcc foo.s  
% ./a.out  
999  
%
```

```
.text  
.globl _main  
.p2align 4, 0x90  
_main:  
    pushq %rbp  
    movq %rsp, %rbp  
    movl $999, -4(%rbp)  
    leaq L_.str(%rip), %rdi  
    movl -4(%rbp), %esi  
    movb $0, %al  
    callq _printf  
    popq %rbp  
    retq  
.cstring  
L_.str:  
.asciz "%d¥n"
```

実行例

- ステップ実行とレジスタ値の表示

```
(lldb) b main
(lldb) break set -a 0x0000000100000f60
(lldb) run
-> 0x100000f60 <+0>: pushq   %rbp
    0x100000f61 <+1>: movq    %rsp, %rbp
(lldb) stepi
-> 0x100000f61 <+1>:  movq    %rsp, %rbp
    0x100000f64 <+4>:  subq    $0x10, %rsp
(lldb) stepi
-> 0x100000f64 <+4>:  subq    $0x10, %rsp
    0x100000f68 <+8>:  movl    $0x3e7, -0x4(%rbp)
(lldb) register read $rbp
    rbp = 0x00007ffeefbfff780
(lldb) register read
    rax = 0x0000000100000f60   a.out`main
    rbx = 0x0000000000000000
    ...
```

アドレスを指定してブレークも可

次に実行する機械語命令

1命令実行するステップ実行

さらに1命令実行するステップ実行

レジスタ%rbpの値を表示

汎用レジスタの値をすべて表示

実行例

- メモリの値の表示（短縮ではない正規コマンド）

サイズを8バイト，表示形式は16進数で，
個数は4個で，というオプション指定

```
(lldb) register read $rbp
```

```
rbp = 0x00007ffeefbfff780 ▼ %rbpの現在の値
```

```
(lldb) memory read --size 8 --format x --count 4 0x7ffeefbfff780
```

アドレス0x7ffeefbfff780の値を表示

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) memory read --size 8 --format x --count 4 $rbp %rbpを指定してもOK
```

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) memory read --size 8 --format x --count 4 $rbp+16
```

16(%rbp)のメモリ値を表示

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
0x7ffeefbfff7a0: 0x0000000000000001 0x00007ffeefbfff948
```

```
(lldb)
```

実行例

- メモリの値の表示（全ページと内容は同じ，短縮形のオプションを使用）

サイズを8バイト，表示形式は16進数で，
個数は4個で，というオプション指定

```
(lldb) register read $rbp
```

```
rbp = 0x00007ffeefbfff780
```

▼ %rbpの現在の値

```
(lldb) memory read -s 8 -f x -c 4 0x7ffeefbfff780
```

アドレス0x7ffeefbfff780の値を表示

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) memory read -s 8 -f x -c 4 $rbp
```

%rbpを指定してもOK

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) memory read -s 8 -f x -c 4 $rbp+16
```

16(%rbp)のメモリ値を表示

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
0x7ffeefbfff7a0: 0x0000000000000001 0x00007ffeefbfff948
```

```
(lldb)
```

実行例

xはmemory readの短縮形

- メモリの値の表示（全ページと内容は同じ，短縮コマンドを使用）

サイズを8バイト，表示形式は16進数で，
個数は4個で，というオプション指定

```
(lldb) register read $rbp
```

```
rbp = 0x00007ffeefbfff780    %rbpの現在の値
```

```
(lldb) x -s 8 -f x -c 4 0x7ffeefbfff780    アドレス0x7ffeefbfff780の値を表示
```

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) x -s 8 -f x -c 4 $rbp                %rbpを指定してもOK
```

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) x -s 8 -f x -c 4 $rbp+16            16(%rbp)のメモリ値を表示
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
0x7ffeefbfff7a0: 0x0000000000000001 0x00007ffeefbfff948
```

```
(lldb)
```

実行例

xはmemory readの短縮形

- メモリの値の表示（全ページと内容は同じ，gdb由来の短縮オプションを使用）

```
(lldb) register read $rbp
```

```
rbp = 0x00007ffeefbfff780    %rbpの現在の値
```

```
(lldb) x/4gx 0x7ffeefbfff780    アドレス0x7ffeefbfff780の値を表示
```

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) x/4gx $rbp    %rbpを指定してもOK
```

```
0x7ffeefbfff780: 0x00007ffeefbfff798 0x00007fff678912e5
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
(lldb) x/4gx $rbp+16    16(%rbp)のメモリ値を表示
```

```
0x7ffeefbfff790: 0x00007fff678912e5 0x0000000000000000
```

```
0x7ffeefbfff7a0: 0x0000000000000001 0x00007ffeefbfff948
```

```
(lldb)
```

4 は表示する個数

g は8バイトごとにまとめて表示（4バイトごとなら w）

x は16進数表記，を意味するオプション

注：スラッシュ(/)の前にスペースを入れてはいけない

実行例

- 逆アセンブル

```
(lldb) disassemble -n main          main関数を逆アセンブル
```

```
a.out`main:
```

```
    0x100000f60 <+0>:  pushq   %rbp
    0x100000f61 <+1>:  movq    %rsp, %rbp
->  0x100000f64 <+4>:  subq    $0x10, %rsp
```

```
(lldb) dis -n main                  disはdisassembleの短縮コマンド
```

```
a.out`main:
```

```
    0x100000f60 <+0>:  pushq   %rbp
    0x100000f61 <+1>:  movq    %rsp, %rbp
->  0x100000f64 <+4>:  subq    $0x10, %rsp
```

```
(lldb)
```

```
% objdump -d a.out                 GNU Binutilsのobjdumpコマンドでも逆アセンブル可
```

```
__text:
```

```
100000f60:      55                pushq   %rbp
100000f61:      48 89 e5          movq    %rsp, %rbp
100000f64:      48 83 ec 10       subq    $16, %rsp
100000f68:      c7 45 fc e7 03 00 00  movl    $999, -4(%rbp)
```

お便利機能

ヘルプとアプロポス

- helpコマンド

```
(lldb) help      短縮コマンドを含めて、全コマンド一覧を表示
```

```
(lldb) help break  breakのサブコマンド一覧を表示
```

```
...
set    -- Sets a breakpoint or set of breakpoints in the executable.
...
```

```
(lldb) help break set  break setの引数やオプション一覧を表示
```

```
...
-n <function-name> ( --name <function-name> )
...
```

- aproposコマンド

```
(lldb) apropos break  breakに関連するコマンドの情報を表示
```

```
breakpoint
b
rbreak
tbreak
...
```

補完機能

- TABキー（あるいはCTRL-i）で候補列挙と自動補完が可能。
 - lldbコマンド名は常に補完可能。関数名や変数名は文脈が十分な時のみ。

```
% lldb ./a.out
(lldb) br TAB      TABキーを押すと（brとTABの間にはスペースを入れない）
(lldb) breakpoint  breakpointに自動補完
(lldb) breakpoint TAB さらにTABキーを押すと
Available completions: 指定可能なサブコマンド一覧が表示
clear -- Delete or disable breakpoints matching the specified source file
and line. 以下略
(lldb) breakpoint set -name mai TAB TABキーを押すと
(lldb) breakpoint set -name main  mainに自動補完
(lldb) run
(lldb) print a TAB 文脈不足でTABキーを押しても補完されない
(lldb) frame variable a TAB TABキーを押すと
(lldb) frame variable a      この場合は補完候補が表示される
Available completions:
a1
a2
```

```
int main ()
{
    int a1 = 10;
    int a2 = 20;
}
```

ヒストリ機能

- 入力したコマンド列は履歴として保存されている.
- 履歴は表示したり, 同じコマンドを再実行できる.

```
(lldb) b main
(lldb) run
(lldb) step                                CTRL-Fで一つ後のコマンドを表示
(lldb) CTRL-P                             CTRL-Pで一つ前のコマンドを表示
(lldb) step                                1つまえのstepが表示された
(lldb) command history                     コマンドの履歴一覧を表示
0: target create "./a.out"
1: b main
2: breakpoint set --name 'main'           b mainの長いコマンド版も入ってる
3: run
4: step
5: command history
(lldb) !4                                 履歴の番号を指定してコマンドを再実行
(lldb)                                    この場合は step を再実行
```

ブレイクポイント

ブレークポイントの設定（１）

- 関数名を指定してブレーク

```
% gcc -g hello.c
% lldb ./a.out
(lldb) break set -name main      main関数にブレークポイントを設定
(lldb) b main                   上と同じことをする短縮形のコマンド
(lldb) run
Process 62676 launched: '/tmp/a.out' (x86_64)
Process 62676 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
   frame #0: 0x0000000100000f68 a.out`main at hello.c:4:5
   1      #include <stdio.h>
   2      int main ()
   3      {
->  4          printf ("hello, world¥n");  この行を実行する直前で一時停止
   5      }
```

Target 0: (a.out) stopped.

```
(lldb) quit
```

ブレークポイントの設定（2）

- ファイル名と行番号を指定してブレーク

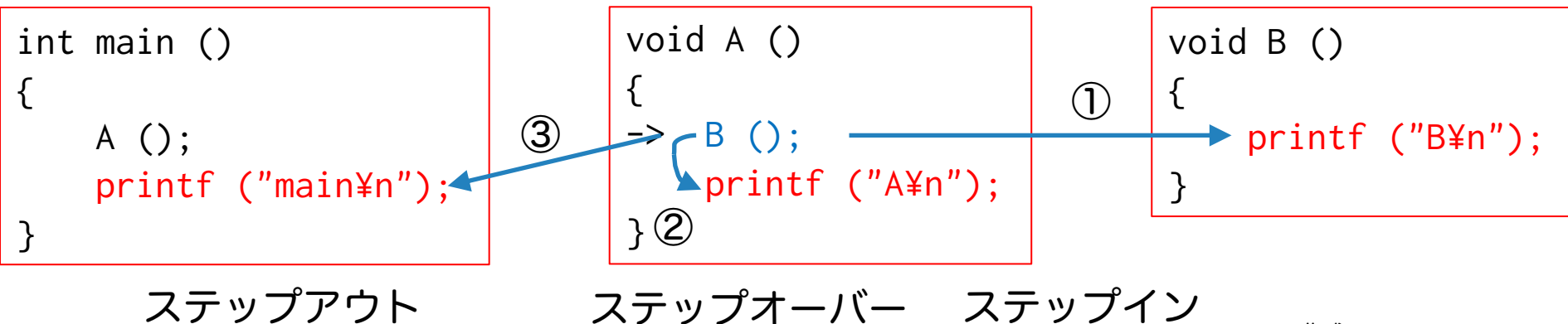
```
% llldb ./a.out
(llldb) break set --file hello.c --line 4
        ファイルhello.cの4行目にブレークポイントをセット
(llldb) b hello.c:4      上と同じことをする短縮形のコマンド
(llldb) run
Process 62692 launched: '/tmp/a.out' (x86_64)
Process 62692 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
2.1
    frame #0: 0x0000000100000f68 a.out`main at hello.c:4:5
    1      #include <stdio.h>
    2      int main ()
    3      {
-> 4          printf ("hello, world¥n");
    5      }
Target 0: (a.out) stopped.
(llldb) quit
```


ステップ実行

ちょっと実行して、すぐ実行を停止させる

ステップ実行の種類（まとめ）

- ① **ステップイン実行**：lldbコマンドは step （省略コマンドは s）
 - 今居る関数Aからの関数呼び出しBを**含めて**，次の行まで実行する
- ② **ステップオーバー実行**：lldbコマンドは next （省略コマンドは n）
 - 今居る関数Aからの関数呼び出しBを**含めず**，次の行まで実行する
- ③ **ステップアウト実行**：lldbコマンドは finish
 - 今いる関数Aからリターンするまで実行し，リターン直後に実行を停止する
- 次のブレークポイントまで実行（実行再開）
 - lldbコマンドは continue （省略コマンドは c）



ステップ実行の種類 (step)

- ステップイン実行 (step, s)

```
#include <stdio.h>
void B ()
{
    printf ("B¥n");
}
void A ()
{
    B ();
    printf ("A¥n");
}
int main ()
{
    A ();
    printf ("main¥n");
}
```

```
% llldb ./a.out
(llldb) b A      関数Aの先頭にブレークポイント設定
(llldb) run
6 void A ()
7 {
-> 8     B ();   この行の実行直前でブレーク
9     printf ("A¥n");
10 }
(llldb) step
2 void B ()
3 {
-> 4     printf ("B¥n"); Bの最初の実行で実行停止
5 }
```

ステップ実行の種類 (next)

- ステップオーバー実行 (next, n)

```
#include <stdio.h>
void B ()
{
    printf ("B¥n");
}
void A ()
{
    B ();
    printf ("A¥n");
}
int main ()
{
    A ();
    printf ("main¥n");
}
```

```
% llldb ./a.out
(llldb) b A      関数Aの先頭にブレークポイント設定
(llldb) run
6   void A ()
7   {
-> 8       B ();   この行の実行直前でブレーク
9       printf ("A¥n");
10  }
(llldb) next
6   void A ()
7   {
8       B ();
-> 9       printf ("A¥n"); B()リターン直後で実行停止
10  }
```

ステップ実行の種類 (finish)

- ステップアウト実行 (finish)

```
#include <stdio.h>
void B ()
{
    printf ("B¥n");
}
void A ()
{
    B ();
    printf ("A¥n");
}
int main ()
{
    A ();
    printf ("main¥n");
}
```

```
% lladb ./a.out
(lladb) b A      関数Aの先頭にブレークポイント設定
(lladb) run
      6 void A ()
      7 {
-> 8     B ();   この行の実行直前でブレーク
      9     printf ("A¥n");
     10 }
(lladb) finish
     11 int main ()
     12 {
     13     A ();
-> 14     printf ("main¥n");
        A()からのリターン直後で実行停止
     15 }
```

ステップ実行の種類 (continue)

- 次のブレークポイントに出会うまで実行 (continue, c)

```
#include <stdio.h>
void B ()
{
    printf ("B¥n");
}
void A ()
{
    B ();
    printf ("A¥n");
}
int main ()
{
    A ();
    printf ("main¥n");
}
```

```
% llldb ./a.out
(llldb) b A      関数Aの先頭にブレークポイント設定
(llldb) run
6    void A ()
7    {
-> 8        B ();   この行の実行直前でブレーク
9        printf ("A¥n");
10   }
(llldb) continue 次のブレークポイント
B
A
main
ブレークポイントに出会わなかったなので
最後まで実行して終了
(llldb)
```

実行状態の表示

変数の値の表示 (変数)

```
int main ()
{
    int i = 10;
    int a [5] = {1, 2, 3, 4, 5};
    a [3] = i;
    a [4] = a[3] + 5;
}
```

(lldb) **b -1 6** 6行目にブレークポイントを設定

(lldb) **run**

```
5          a [3] = i;
-> 6          a [4] = a[3] + 5;
7      }
```

(lldb) **expression i** 変数iの値を表示 (iの部分には任意の式を書ける)

(int) \$0 = 10

(lldb) **expr i** 上の短縮形

(int) \$1 = 10

(lldb) **print i** printは expression -- の短縮形

(int) \$2 = 10

(lldb) **p i** pはexpression -- の短縮形

(int) \$3 = 10

変数の値の表示 (配列)

```
int main ()
{
    int i = 10;
    int a [5] = {1, 2, 3, 4, 5};
    a [3] = i;
    a [4] = a[3] + 5;
}
```

```
(lldb) b -1 6
```

```
(lldb) run
```

```
5          a [3] = i;
-> 6          a [4] = a[3] + 5;
7      }
```

```
(lldb) p a[3]      配列a[3]の値を表示
```

```
(int) $2 = 10
```

```
(lldb) p a          配列aの値をすべて表示
```

```
(int [5]) $3 = ([0] = 1, [1] = 2, [2] = 3, [3] = 10, [4] = 5)
```

```
(lldb) p $2+$3      $で始まる変数はlldbの変数。lldbコマンドの引数で使って良い。
```

```
(int) $4 = 13
```

構造体の値の表示（１）

- コード例：簡単な線形リスト

```
#include <stdio.h>
struct node {
    int  data;
    struct node *next;
};
int main ()
{
    struct node n1 = {10, NULL};
    struct node n2 = {20, &n1};
    struct node n3 = {30, &n2};
    struct node *p = &n3;
}
```

構造体の値の表示 (1)

```
(lldb) b -1 12
```

```
(lldb) run
```

```
11      struct node *p = &n3;
```

```
-> 12 }
```

```
(lldb) print p      ポインタpの値の表示
```

```
(node *) $0 = 0x00007ffeefbfff750
```

```
(lldb) print *p      ポインタpが指す構造体の値を表示
```

```
(node) $1 = {
```

```
    data = 30
```

```
    next = 0x00007ffeefbfff760
```

```
}
```

```
(lldb) print *(p->next) 2番目のノードの値を表示
```

```
(node) $4 = {
```

```
    data = 20
```

```
    next = 0x00007ffeefbfff770
```

```
}
```

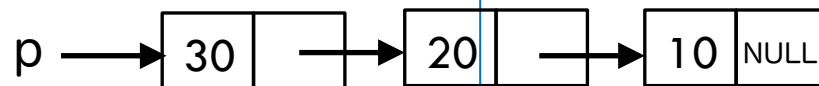
```
(lldb) print *(p->next->next) 3番目のノードの値を表示
```

```
(node) $5 = {
```

```
    data = 10
```

```
    next = 0x0000000000000000
```

```
}
```



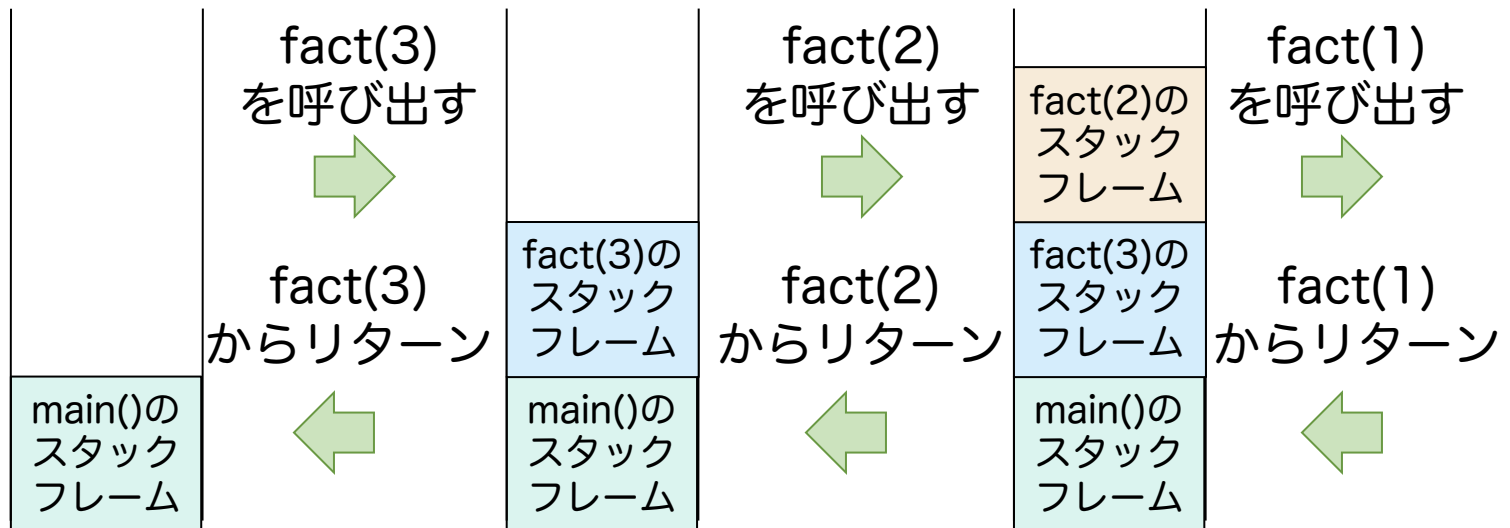
スタックトレース (1)

- コード例：階乗の計算

```
#include <stdio.h>
int fact (int n) {
    if (n <= 0) return 1;
    else return n * fact (n - 1);
}
int main ()
{
    printf ("%d¥n", fact (3));
}
```

スタックトレース (2)

- **スタックフレーム** = 関数呼び出し1回分のデータ。スタック上に配置。
 - 局所変数, 引数, 返り値, 戻り番地, 退避したレジスタの値などを含む。
- 関数を呼び出すとスタックフレームをスタックに積み,
リターンするとスタックから取り除く
- **スタックトレース (バックトレース)**
= 実行現時点でのスタック上の, 全てのスタックフレームを並べたもの



スタックトレース (3)

```
(lldb) breakpoint set -n fact -c n==0
```

```
(lldb) r
```

```
    2    int fact (int n) {  
-> 3        if (n <= 0) return 1;
```

```
(lldb) bt スタックトレース (バックトレース) を表示
```

```
* frame #0: 0x100000f1b a.out`fact(n=0) at fact.c:3:11 今居るスタックフレーム  
  frame #1: 0x100000f44 a.out`fact(n=1) at fact.c:4:21  
  frame #2: 0x100000f44 a.out`fact(n=2) at fact.c:4:21  
  frame #3: 0x100000f44 a.out`fact(n=3) at fact.c:4:21  
  frame #4: 0x100000f72 a.out`main at fact.c:8:21
```

```
(lldb) up 1つ上 (呼び出した関数) のスタックフレームに移動
```

```
frame #1: 0x0000000100000f44 a.out`fact(n=1) at fact.c:4:21
```

```
-> 4        else return n * fact (n - 1);
```

```
(lldb) print n    nの値は1
```

```
(int) $5 = 1
```

```
(lldb) down 1つ下 (呼び出された関数) のスタックフレームに移動
```

```
frame #0: 0x0000000100000f1b a.out`fact(n=0) at fact.c:3:11
```

```
-> 3        if (n <= 0) return 1;
```

```
(lldb) print n    nの値は0
```

```
(int) $6 = 0
```

```
(lldb)
```

ソースコードの表示

```
(lldb) breakpoint set -n fact -c n==0
(lldb) run
   2      int fact (int n) {
-> 3          if (n <= 0) return 1;
(lldb) list 2      2行目からソースコードを表示
   2      int fact (int n) {
   3          if (n <= 0) return 1;
   4          else return n * fact (n - 1);
   5      }
   6      int main ()
   7      {
   8          printf ("%d¥n", fact (3));
   9      }
(lldb) list main    関数mainのソースコードを表示
   6      int main ()
   7      {
   8          printf ("%d¥n", fact (3));
   9      }
```

演習問題

演習 (1)

- Segmentation Faultが起こる原因を調べよ

```
#include <stdio.h>
static int *p;
void foo () {
    p = NULL;
    return;
}
int main () {
    int i = 999;
    p = &i;
    foo ();
    printf ("%d¥n", *p);
}
```

```
% gcc foo.c
% ./a.out
Segmentation fault
%
```

答 (1)

- 関数foo中でポインタ変数pにNULLが代入されていた

```
% lladb ./a.out
(lldb) r
* thread #1, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=1, address=0x0)
-> 10      printf ("%d¥n", *p);    ここでSegmentation Fault発生
(lldb) print p
(int *) $0 = 0x0000000000000000    ポインタ変数pの値がNULLのせい
```

```
% lladb ./a.out    どこでpにNULLが代入されるか調べる
(lldb) b main
(lldb) r
(lldb) watchpoint set variable p    変数pにウォッチポイントを設定
(lldb) watchpoint modify -c p==0
    最後に作ったウォッチポイントに止まる条件 p==0 を追加
(lldb) c
    3      void foo () {
    4          p = NULL;
-> 5      }    ここでpにNULLが代入されていることが判明！
               これが最後じゃないなら、ぬるぽするまでcを繰り返す
```

演習（2）

- %rsp 16バイト境界違反の原因を調べよ
 - macOSではcall命令直前に、%rspの値は16の倍数でなければならない
 - この約束を破ると実行時エラーになる

```
.text
.globl _main
.p2align 4, 0x90
_main:
movl $999, -4(%rbp)
leaq L_.str(%rip), %rdi
movl -4(%rbp), %esi
movb $0, %al
# ここで%rspは16の倍数が必要
callq _printf
retq
.cstring
L_.str:
.asciz "%d¥n"
```

```
% gcc foo.c
% ./a.out
Segmentation fault
%
```

答 (2)

普通にデバッグすると、スタックフレームが壊れていて
どのcall命令で16バイト境界違反が起きたのかわからない。

```
% gcc -g foo.s
% lldb ./a.out
(lldb) r
* thread #1, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=EXC_I386_GPFLT)  一般保護例外で実行停止
    frame #0: 0x00007fff67882386
        スタックが16バイト境界じゃないエラーという関数が呼ばれてる
libdyld.dylib`stack_not_16_byte_aligned_error:
-> 0x7fff67882386 <+0>: movdqa %xmm0, (%rsp)
    0x7fff6788238b <+5>: int3
(lldb) bt  スタックトレースを表示させても
* thread #1, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=EXC_I386_GPFLT)
    * frame #0: 0x00007fff67882386  main関数のフレームが存在しない
libdyld.dylib`stack_not_16_byte_aligned_error
(lldb)
```

答 (2)

```
(lldb) dis -n main    main関数を逆アセンブル
a.out`main:
    0x100000f81 <+17>: movb    $0x0, %al
    0x100000f83 <+19>: callq   0x100000f8a
(lldb) breakpoint set -a 0x100000f81    call命令の一つ手前でブレーク
(lldb) r
(lldb) si                                call命令直前で実行を止める
-> 0x100000f83 <+19>: callq   0x100000f8a
(lldb) p/x $rsp                          %rspレジスタの値が16じゃない(ここが原因)
(unsigned long) $1 = 0x00007ffeefbfff788
```

16の倍数ならば、16進数の1桁目はゼロになる。

例：0x00007ffeefbfff78**0**

%rspの16バイト境界のチェックが、call命令の実行前に行われるので、call命令の一つ手前でブレークした

演習（3）

- 論理デバッグをせよ。dtohは10進数を16進数の文字（0～9，A～F）に変換する関数だが，10の時の結果がおかしい

```
#include <stdio.h>
// ASCIIコードを仮定
int dtoh (int d)
{
    if (d > 10) {
        return 'A' + d - 10;
    } else {
        return '0' + d;
    }
}

int main ()
{
    for (int i = 0; i < 16; i++) {
        printf ("%d:%c, ", i, dtoh(i));
    }
    puts ("¥n");
}
```

```
% gcc -g foo.c
% ./a.out
0:0, 1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7,
8:8, 9:9, 10::, 11:B, 12:C, 13:D, 14:E,
15:F,      ↑ この10の出力がおかしい
```

答 (3)

```
% gcc -g foo.c
% lldb ./a.out
(lldb) b -n dtoh -c d==10    d==10の時だけ、関数dtohの呼び出しをブレーク
(lldb) r
-> 4      if (d > 10) {    d==10の時、'A' + d - 10 を返さなくてはならないのに
5          return 'A' + d - 10;
6      } else {
7          return '0' + d;
(lldb) step
4      if (d > 10) {
5          return 'A' + d - 10;
6      } else {
-> 7          return '0' + d;    else節を実行して '0' + d を返していた
(lldb) finish
Return value: (int) $11 = 58    文字 ':' のASCIIコードは58
-> 13          printf ("%d:%c, ", i, dtoh(i));

4行目を if (d >= 10) { にすれば正しくなる
```

コマンド一覧表

「lldbチートシート」で検索すると
より詳細で網羅的な一覧表が出てくる

起動	lldb ./a.out	レジスタ値を読む	register read
終了	quit, q		reg read
実行開始	run, r	メモリ値を読む	memory read, x
	process launch		mem read
式の評価	expression, expr, e	逆アセンブル	disassemble, dis, d
	print, p	ヘルプ	help, h
ブレイクポイント の設定	breakpoint, p	アプロポス	apropos, a
		ソースコード表示	list, l
ステップイン実行	step, s	バックトレース	bt
ステップオーバー 実行	next, n	フレーム変数表示	frame variable, v
		フレーム上に移動	up
ステップオーバー 実行	finish	フレーム下に移動	down
		ウォッチポイント	watchpoint, w
実行再開	continue, c	コマンド履歴	command history
1命令実行	stepi, si	プロセスアタッチ	process attach

おまけ

コアダンプ・ファイルを使う

コアダンプ・ファイルとは？

- コアダンプ・ファイル（コアファイル）とは
 - 実行中やクラッシュ時の**プログラム実行状態**を格納したファイル
 - 実行状態＝その実行時点でのプログラムの**全メモリ・全レジスタの値**
 - 以前の実行履歴は含まない。例：呼び出しが終了した手続きの情報は含まない。
 - 全メモリの番地と値の情報を「**メモリイメージ**」と呼ぶ。
 - 通常、メモリの値はユーザ空間のみ。カーネル空間は含まない。
- **再現性のないバグ**のデバッグに非常に便利 post-mortem
 - コアダンプ・ファイルがあれば、何度でもデバッガで**検死解剖**が可能だから
- コアダンプ・ファイルの管理
 - 再現性のないバグのコアダンプ・ファイルは貴重 ← 消してはダメ
 - 不要なコアダンプ・ファイルはファイルシステムを圧迫 ← 不要なものは消す

コアダンプ・ファイルを使う準備

- 準備：Unix系OSでのコアダンプ・ファイル取得の設定
 - コアダンプ・ファイルのサイズをシェルで事前に要設定
 - sh系
 - `ulimit -c unlimited`
 - csh系
 - `limit coredumpsize unlimited`
- macOSでは、さらに以下が必要（要管理者権限）
 - `sudo sysctl kern.coredump=1` 再起動で設定はリセット
 - `sudo chmod o+w /cores` 再起動してもこの変更は有効

コアダンプ・ファイルを使う

- Segmentation faultが起きる原因をコアダンプ・ファイルを使って調べる (foo.cの内容は演習1と同じ)

```
% gcc -g foo.c
% ./a.out
Segmentation fault (core dumped)
% ls -l /cores/core.*
-r----- 1 gondow wheel 2046820352 /cores/core.57968
                                     ↑ コアファイル, 2GBとでかい
% lldb -c /cores/core.57968 ./a.out   コアダンプを使ってデバッガ起動
(lldb) bt
* thread #1, stop reason = signal SIGSTOP
  * frame #0: 0x000000010398af66 a.out`main at foo.c:11:17
(lldb) list
   10   foo ();
   11   printf ("%d¥n", *p);   ここ (11行目の17文字目) が原因と判明
   12   }
(lldb)
```

コアダンプファイルができる場所はシステムにより異なる

コアダンプ・ファイルを生成する

- lldb 中でコアダンプ・ファイルを生成できる

```
% gcc -g foo.c
% lldb ./a.out
(lldb) b main
(lldb) r
(lldb) process save-core mycorefile    mycorefileは出力先のファイル名
mach_header: 0xfeedfacf 0x01000007 0x00000003 0x00000004 0x000003dd
0x000116b0 0x00000000 0x00000000
0x00000019 0x00000048 [0x0000000100000000 - 0x0000000100001000)
[0x00000000000012000 0x0000000000001000) 0x00000005 0x00000005
0x00000000 0x00000000] ...      延々と長い出力
(lldb) quit
% ls -l mycorefile              コアダンプ・ファイルの存在を確認
-rw-----  1 gondow  wheel  1990180864  1  6 15:38  mycorefile
%                               ↑やはり約2GBと大きい
```

lldbの多言語対応

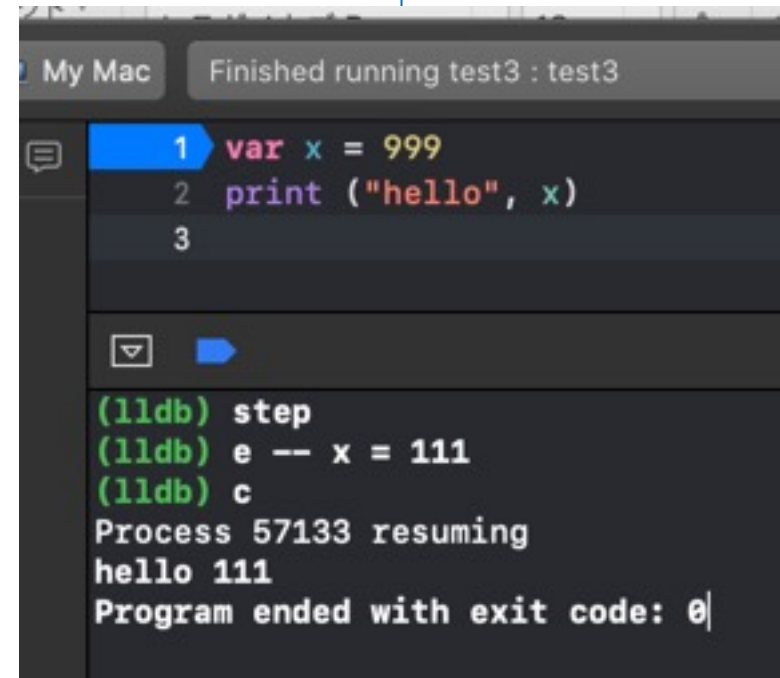
lldbは多言語対応

- サポートする言語
 - C, C++, Objective-C, Swift, アセンブリ言語
- サポートするOS
 - macOS, Linux, FreeBSD, Windows
- サポートする命令セット
 - i386, x86-64, ARM
- lldbはXcodeのデフォルトデバッガ
 - XcodeはiOSアプリとmacOSアプリの統合開発環境
- iOSアプリも, macOS上のデバッグ用バイナリは x86-64なので, x86-64の知識やLLVMツールチェーン (例: nm, objdump)を使用可能

Swiftコードのデバッグ例

```
% swiftc -g main.swift
% lladb ./main
(lladb) b main.swift:1
Breakpoint 1: where = main`main + 19 at main.swift:1:9,
address = 0x0000000100000df3
(lladb) r
-> 1      var x = 999
      2      print ("hello", x)
(lladb) s
      1      var x = 999
-> 2      print ("hello", x)
(lladb) e -- x = 111  xの値を111に変更
(lladb) c
hello 111          xの値が111に変わっている
(lladb) quit
%
```

```
var x = 999
print ("hello", x)
```



- コンパイラ swiftcのインストール方法は各自で調べてね
- Xcodeでも lldbを使用可能 (右図)
- Xcodeでビルドした実行可能ファイルをlldbコマンドでデバッグ可能

プロセスのアタッチ

プロセスのアタッチ

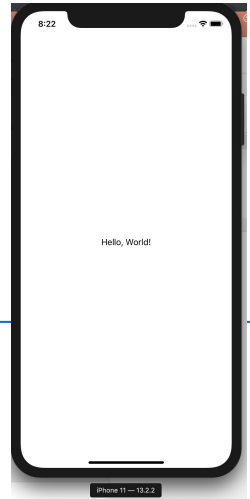
- 後付けで動作中のプロセスをデバッガ監視下に置く

```
#include <stdio.h>
int main () {
    int x = 1, n = 0;
    while (x) {
        n++;
    }
}
```

```
% gcc -g foo.c
% ./a.out
無限ループ
```

```
% ps | egrep a.out
66918 ttys004      0:00.00 egrep a.out
66913 ttys005      1:19.72 ./a.out
% lladb -p 66913 プロセス番号66913をlldb監視下に
(lldb) process attach --pid 66913
Process 66913 stopped
      4          while (x) {
-> 5              n++;
      6          }
(lldb) expr x=0 変数xに0を代入して
(int) $0 = 0
(lldb) continue 実行再開して無事に終了
Process 66913 resuming
Process 66913 exited with status = 0 (0x00000000)
(lldb)
```

iOSアプリにアタッチ



- iOSシミュレータ上で動作するiOSアプリにアタッチ

```
% xcrun simctl list devices      iOSシミュレータのID番号を確認
-- iOS 13.2 --
iPhone 11 (6AB60DA8-EC11-41C9-9E8A-D76B51AE3419) (Shutdown)
% open -a Simulator --args -CurrentDeviceUDID 6AB60DA8-EC11-41C9-9E8A-
D76B51AE3419                    ID番号を指定してiOSシミュレータを起動
% xcrun simctl install booted /Users/gondow/Library/Developer/Xcode/
DerivedData/HelloWorld-ergkwrbkxdttlbgtworicbosatwz/Build/Products/
Debug-iphonesimulator/HelloWorld.app iOSシミュレータにアプリをインストール
ここでiOSシミュレータ上で、インストールしたアプリを起動しておく
% lladb
(lladb) process attach -n HelloWorld --waitfor  iOSアプリにアタッチ
Process 67038 stopped
-> 0x114b0a82a <+10>: jae      0x114b0a834
      0x114b0a82c <+12>: movq    %rax, %rdi
Target 0: (HelloWorld) stopped.
(lladb) continue
Process 67038 resuming
(lladb)
```

メモ

- stepで停止後, displayで表示されない (stop-hookが機能しない)
- target stop-hook 中で continue を入れると, 実行時エラーが起きても continue し続ける. CTRL-i 連打で黙らせられるが, コマンド的に防ぐ方法は不明.
- ログのとり方例
 - log enable -f /tmp/log.txt lldb api