第 10 回 Unix ゼミ
C プログラム(デバッグ編)

高木 空

2024 年 07 月 01 日

目次

GDB

Chapter 1

1 概要

1.1 GDB とは

GDB は Gnu Project のデバッガです。

1.2 コマンド

GDB を起動すると先頭に(gdb)の表示が出て対話モードになります。

```
(gdb) <command> [<options>]
```

の形式でコマンドを実行し、様々の処理を行うことでデバッグを行います。

コマンド入力時は TAB による補完が可能です。また、曖昧さがない場合にはコマンドは先頭数文字だけの入力でも認識されます。

1.3 ヘルプ

コマンドのヘルプを help コマンドで確認できます。

```
(gdb) help [<class>|<command>|all]
```

argument name	description
class	特定のクラスのコマンドのリストを表示
command	特定のコマンドのヘルプを表示
all	全てのコマンドのリストを表示

class に指定できるもの:

class name	description
aliases	他のコマンドのエイリアス
breakpoints	特定の場所でプログラムを停止させる
data	データを検査する
files	ファイルを指定、検査する
internals	メンテナンスコマンド
obscure	ぼんやりした特徴
running	プログラムを走らせる
stack	スタックを検査する
status	ステータスの検査
support	サポート機能
tracepoint	プログラムを停止せずにトレースする
user-defined	ユーザー定義のコマンド

この一覧は引数無しで help コマンドを実行することで閲覧できます。

2 基本操作

2.1 GDB の起動

GDB を起動するには以下のコマンドをシェルで叩きます。

```
$ gdb [<options>] [<program> [<core-file>|<pid>]]
$ gdb [<options>] --args <program> [<args>...]
```

`options` に指定できるもの

option name	description
ファイル等の指定	
`--args`	実行ファイル以降の引数をプログラムに渡す
`--core <file>`	ダンプされたコアファイルを検査
`--exec <file>`	実行ファイルを検査
`--pid <pid>`	プロセス ID を指定してアタッチ
`--directory <dir>`	ソースファイル検索ディレクトリを指定
`--se <file>`	シンボルファイルおよび実行ファイルを指定
`--symbol <file>`	シンボルファイルを指定
`--readnow`	全てのシンボルを最初のアクセス時に読み込む
`--readnever`	シンボルファイルを読み込まない
`--write`	実行ファイルとコアファイルに書き込みを行う
初期化コマンドとコマンドファイル	
`-x, --command <file>`	ファイルからコマンドを実行
`-ix, --init-command <file>`	実行ファイル読み込み前に実行するコマンドファイル
`-ex, --eval-command <cmd>`	コマンドを実行
`-iex, --init-eval-command <cmd>`	実行ファイル読み込み前に実行するコマンド
`--nh`	`~/.gdbinit` を読み込まない
`--nx`	`.gdbinit` を全て読み込まない
出力と UI 制御	
`--fullname`	emacs-GDB で使用される出力情報
`--interpreter <interpreter>`	インタープリターを指定
`--tty <tty>`	デバッグするプログラムの入出力 TTY を指定
`-w`	GUI を使用
`--nw`	GUI を使用しない
`--tui`	TUI を使用
`--dbx`	DBX compatibility モード

`-q, --quiet, --silent`	GDB 開始時のメッセージを表示しない
モード	
`--batch`	バッチモードで実行
`--batch-silent`	バッチモードで実行し、出力を行わない
`--return-child-result`	デバッグするプログラムの終了コードで GDB も終了する
`--configuration`	GDB の詳細な設定を表示
`--help`	GDB 起動のヘルプを表示
`--version`	バージョン情報を表示
リモートデバッグオプション	
`-b <baudrate>`	帯域幅を指定
`-l <timeout>`	タイムアウト時間(秒)を指定
その他のオプション	
`--cd <dir>`	カレントディレクトリを変更
`-D, --data-directory <dir>`	データディレクトリを変更

2.2 プログラムの読み込み

デバッグするプログラムを GDB に読み込むには起動時の引数で指定するか、コマンドで指定することができます。

起動時には例えば

```
$ gdb ./a.out
```

のように指定し、コマンドでは

```
(gdb) file ./a.out
```

のように指定できます。

2.3 プログラムの起動

プログラムを起動するには `run` コマンドまたは `start` コマンドを使用します。 `run` コマンドは設定した停止場所まで継続実行し、`start` ではエントリポイントで停止します。

`run` コマンドの文法:

```
(gdb) run <args>
```

argument name	description
`args`	プログラムにコマンドライン引数として渡す

start コマンドの文法:

```
(gdb) start <args>
```

argument name	description
args	プログラムにコマンドライン引数として渡す

2.4 プログラムの終了
プログラムを終了するには ctrl+c を使用するか kill コマンドで終了できます。

2.5 GDB の終了
GDB を終了するには ctrl+d を使用するか quit または exit コマンドで終了できます。

3 プログラムの停止
デバッグ中のプログラムを停止させるにはあらかじめ停止場所や条件を設定しておく必要があります。停止箇所としてはブレークポイント、ウォッチポイント、キャッチポイントが設定できます。

なお、ブレークポイント、ウォッチポイント、キャッチポイントをまとめてブレークポイントと呼ぶこともあります。

3.1 ブレークポイント
プログラムが到達した際に停止する場所です。

3.1.1 ブレークポイントの設置
ブレークポイントを設置するには break コマンドを使用します。

```
(gdb) break [<location>] [thread <tnum>] [[-force-condition] if [<cond>]]
```

argument name	description
location	設置場所を指定します。詳細は Section 3.1.1.1 を参照
thread <tnum>	特定のスレッドでのみ停止。スレッド番号は info threads で確認
-force-condition	cond の式が現在の文脈で無効でも無視して設置
cond	条件式を満たすときのみ停止

3.1.1.1 位置指定
位置の指定方法はいくつかあります。代表的なものは
- [<file>:]<linenum>: 行番号を指定
- [<file>:]<funcname>: 関数名を指定
- <addr>: アドレスを指定

詳細は https://sourceware.org/gdb/current/onlinedocs/gdb.html/Location-Specifications.html#Location-Specifications をご覧ください。

3.2 ウォッチポイント

式の値を監視して変更があった際にプログラムを停止させます。

3.2.1 ウォッチポイントの設置

ウォッチポイントを設置するには `watch` コマンドを使用します。

```
(gdb) watch [-l <location>] <expr>
```

argument name	description
`location`	`expr` が指すアドレスのメモリを監視
`expr`	監視する式

3.3 キャッチポイント

例外やシステムコールなどの際に停止します。

3.3.1 キャッチポイントの設置

キャッチポイントを設置するには `catch` コマンドを使用します。

```
(gdb) catch <subcommand> [<args>]
```

subcommand name	description
`catch`	例外のキャッチ
`exec`	`exec` の呼び出し
`fork`	`fork` の呼び出し
`load`	共有ライブラリロード
`rethrow`	例外の再スロー
`signal`	シグナルキャッチ
`syscall`	システムコール発行
`throw`	例外スロー
`unload`	共有ライブラリアンロード
`vfork`	`vfork` 呼び出し

3.4 ブレークポイントの削除

ブレークポイントを削除するには `delete` コマンドを使用します。

```
(gdb) delete [breakpoint-num]
```

argument name	description

`breakpoint-num`	削除するブレークポイントを指定。ブレークポイント番号は `info breakpoints` で確認

3.5 ブレークポイントの無効化

ブレークポイントを無効化するには `disable breakpoints` コマンドを使用します。

```
(gdb) disable breakpoints [breakpoint-num]
```

argument name	description
`breakpoint-num`	無効化するブレークポイントを指定。ブレークポイント番号は `info breakpoints` で確認

3.6 ブレークポイントの有効化

ブレークポイントを有効化するには `enable breakpoints` コマンドを使用します。

```
(gdb) enable breakpoints [breakpoint-num]
```

argument name	description
`breakpoint-num`	有効化するブレークポイントを指定。ブレークポイント番号は `info breakpoints` で確認

3.7 ブレークポイント到達時実行コマンド

ブレークポイントに到達した際に実行するコマンドを `commands` コマンドで設定できます。

```
(gdb) commands <command> [<breakpoint-num> ...]
```

argument name	description
`command`	実行するコマンド
`breakpoint-num`	設定するブレークポイントを指定。ブレークポイント番号は `info breakpoints` で確認。指定しない場合、一番最後のブレークポイントが指定される

3.8 ブレークポイントの保存

`save breakpoints` コマンドで現在設定されているブレークポイントをファイルに保存できます。

```
(gdb) save breakpoints
```

3.9 ブレークポイントの読み込み

保存したブレークポイントは `source` コマンドで読み込めます。

```
(gdb) source <file>
```

4 プログラムの再開

ブレークポイント等で停止したプログラムを様々な方法で再開できます。再開方法は主にステップ実行と継続実行に分かれ、ステップ実行では一定分だけ進んで停止し、継続実行では次のブレークポイントまで進みます。

4.1.1 継続実行

継続実行をするには `continue` コマンドを使用します。

```
(gdb) continue [<count>]
```

argument name	description
`count`	ブレークポイントを無視する回数

4.1.2 関数から返る

関数から返るまで実行するには `finish` コマンドを使用します。 現在選択中のフレームが帰るまで実行します。フレームの選択方法は Section 5.2 を参照してください。

```
(gdb) finish
```

4.1.3 ジャンプ

特定の位置まで継続実行をするには `jump` コマンドを使用します。

```
(gdb) jump <loc>
```

argument name	description
`loc`	停止位置。行番号かアドレスで指定。

4.1.4 ソース行単位ステップオーバー

ソースコード行単位でステップし、関数呼び出しで中に入らないような実行(ステップオーバー)は `next` コマンドを使用します。

```
(gdb) next [<count>]
```

argument name	description
`count`	ステップ回数

4.1.5 命令単位ステップオーバー

機械語命令単位でのステップオーバーは `nexti` コマンドを使用します。

```
(gdb) nexti [<count>]
```

argument name	description
`count`	ステップ回数

4.1.6 ソース行単位ステップイン

ソースコード行単位でステップし、関数呼び出しで中に入るような実行(ステップイン)は `next` コマンドを使用します。

```
(gdb) step [<count>]
```

argument name	description
`count`	ステップ回数

4.1.7 命令単位ステップイン

機械語命令単位でのステップインは `nexti` コマンドを使用します。

```
(gdb) stepi [<count>]
```

argument name	description
`count`	ステップ回数

4.1.8 until

ソース行の上で実際に現在の次の行まで、または指定の場所まで実行するには `until` コマンドを使用します。

```
(gdb) until [<loc>]
```

argument name	description
`loc`	停止場所。指定方法は Section 3.1.1.1 を参照。指定しない場合、現在の行の次の行が指定される

5 スタックフレーム

5.1 バックトレース

コールスタックを表示するには `backtrace` コマンドを使用します。

```
(gdb) backtrace [<options>] [<count>]
```

option name	description
`-entry-values <v>`	関数の引数の表示方法を設定
`-frame-arguments <v>`	スカラ値でない引数の表示設定
`-raw-frame-argument on|off`	raw フォームで引数を表示
`-frame-info <v>`	フレーム情報の表示設定
`-past-main on|off`	`main` 関数以前のフレームを表示するか
`-past-entry on|off`	エントリポイント以前のフレームを表示するか
`-full`	局所変数を全て表示

5.2 フレームの選択

フレームを選択するには `frame` コマンドを使用します。

```
(gdb) frame [<num>]
```

argument name	description
`num`	選択するフレーム番号。フレーム番号は `backtrace` で確認。Section 5.1 を参照

6 変数の表示

6.1 プリント

変数や式の値を確認するには `print` コマンドを使用します。

```
(gdb) print [<options>...] [/<fmt>] [<expr>]
```

argument name	description
`options`	オプションを指定
`fmt`	フォーマットを指定
`expr`	表示する式

指定できるオプションおよびフォーマットは `help print` で確認できます。

6.2 ディスプレイ

ブレークポイント等で停止した際に自動的に変数や式の値を表示するには `display` コマンドを使用します。

```
(gdb) display[/<fmt>] <expr>
```

argument name	description
`fmt`	フォーマットを指定
`expr`	表示する式

LLDB

Chapter 2

1 概要

2 LLDB とは
LLDB は LLVM のデバッガです。

3 コマンド
LLDB を起動すると先頭に(lldb)の表示が出て対話モードになります。

```
(lldb) <noun> <verb> [<options>]
```

の形式でコマンドを実行し、様々の処理を行うことでデバッグを行います。

コマンド入力時は TAB による補完が可能です。また、曖昧さがない場合にはコマンドは先頭数文字だけの入力でも認識されます。

4 ヘルプ
コマンドのヘルプを help で確認できます。

```
(lldb) help [-ahu] [<command>...]
```

文法:

option name	description
-a, --hide-aliases	エイリアスを非表示
-h, --show-hidden-commands	-から始まる隠しコマンドを表示
-u, --hide-user-commands	ユーザー定義コマンドを非表示
command	ヘルプを表示するコマンドを指定

help -au でコマンドの一覧を閲覧できます。

5 基本操作

5.1 LLDB の起動
LLDB を起動するには以下のコマンドをシェルで叩きます。

```
$ lldb [<options>]
```

! TODO !
補足

5.2 プログラムの読み込み

デバッグするプログラムを GDB に読み込むには起動時の引数で指定するか、コマンドで指定することができます。

起動時には例えば

```
$ lldb ./a.out
```

のように指定し、コマンドでは

```
(lldb) target create ./a.out
```

のように指定できます。

5.3 プログラムの起動

プログラムを起動するには `process launch` を使用します。

```
(lldb) process launch [<options>]
```

文法:

option name	description
`-s, --stop-at-entry`	エントリポイントで停止
`-n, --no-stdio`	
`-A, --disable-aslr <bool>`	アドレスランダム化を行うか
`-e, --stderr <file>`	標準エラー出力を指定
`-i, --stdin <file>`	標準入力を指定
`-o, --stdout <file>`	標準出力を指定

全てのオプションは `help process launch` で確認できます。

5.4 プログラムの終了

プログラムを停止せずにトレース終了するには `ctrl+c` を使用するか `process kill` コマンドを使用します。

5.5 LLDB の終了

LLDB を終了するには `ctrl+d` を使用するか `quit` コマンドで終了できます。

6 プログラムの停止

デバッグ中のプログラムを停止させるにはあらかじめ停止場所や条件を設定しておく必要があります。停止箇所としてはブレークポイント、ウォッチポイントが設定できます。

6.1 ブレークポイント

プログラムが到達した際に停止する場所です。

6.1.1 ブレークポイントの設置

ブレークポイントを設置するには `breakpoint set` コマンドを使用します。

```
(lldb) breakpoint set [<options>]
```

文法:

option name	description
`-A, --all-files`	全てのファイルを検索
`-D, --dummy-breakpoints`	ダミーのブレークポイントを設置
`-H, --hardware`	ハードウェアブレークポイントを使用
`-d, --disable`	無効化されたブレークポイントを設置
`-l, --line <linenum>`	行番号 `linenum` を指定
`-a, --address <addr>`	アドレス `addr` を指定
`-n, --name <func>`	関数名 `func` を指定
`-F, --fullname <name>`	関数の完全修飾名を指定
`-S, --selector <selector>`	Objective-C のセレクタ名を指定
`-M, --method <method>`	C++のメソッド名を指定
`-r, --func-regex <reg>`	正規表現 `reg` にマッチする関数名を持つ関数を指定
`-b, --basename <func>`	関数の基本名が `func` の関数を指定(C++の名前空間や引数)を無視
`-p, --source-pattern-regex <reg>`	指定したファイル内のソースコードで正規表現にマッチする箇所を指定
`-E, --language-exceprion <lang>`	指定した言語の例外スローを指定
`-y, --joint-specifier <linespec>`	`filename:line[:column]`の形式でファイルと行を指定
`-k, --structured-data-key <none>`	スクリプトによるブレークポイントの実装に渡されるキーと値のペアのキー。 ペアは複数指定できます。
`-v, --structured-data-value <none>`	スクリプトによるブレークポイントの実装に渡されるキーと値のペアの値。 ペアは複数指定できます。
`-G --auto-continue <bool>`	コマンド実行後自動で再開

-C, --command <cmd>	停止時に自動実行するコマンド
-c, --condition <cond>	条件式 cond を満たすときだけ停止
-i, --ignore-count <n>	ブレークポイントを無視する回数
-o, --one-shot <bool>	一度停止したら削除
-q, --queue-name <name>	指定したキューに入っているスレッドのみ停止
-t, --thread-id <tid>	指定したスレッドのみ停止
-x, --thread-index <tidx>	指定したインデクスのスレッドのみ停止
-T, --thread-name <name>	指定したスレッドのみ停止
-R, --address-slide <addr>	指定されたオフセットを、ブレークポイントが解決するアドレスに追加します。現在のところ、これは指定されたオフセットをそのまま適用し、命令境界に整列させようとはしません。
-N, --breakpoint-name <name>	ブレークポイントの名前
-u, --column <col>	列を指定
-f, --file <filename>	検索するファイルを指定
-m, --move-to-nearest-code <bool>	一番近いコードへブレークポイントを移動
-s, --shlib <name>	共有ライブラリを指定
-K, --skip-prologue <bool>	プロローグをスキップ

6.1.2 ブレークポイントの削除

ブレークポイントを削除するには `breakpoint delete` コマンドを使用します。

```
(lldb) breakpoint delete [<breakpoint-id>]
```

`breakpoint-id` の値は `breakpoint list` で確認できます。 ブレークポイントを指定せずに実行すればすべてのブレークポイントを対象にできます。

6.1.3 ブレークポイントの無効化

ブレークポイントを無効化するには `breakpoint disable` コマンドを使用します。

```
(lldb) breakpoint disable [<breakpoint-id-list>]
```

ブレークポイントを指定せずに実行すればすべてのブレークポイントを対象にできます。

6.1.4 ブレークポイントの有効化

ブレークポイントを有効化するには `breakpoint enable` コマンドを使用します。

```
(lldb) breakpoint enable [<breakpoint-id-list>]
```

ブレークポイントを指定せずに実行すればすべてのブレークポイントを対象にできます。

6.1.5 ブレークポイント到達時実行コマンド

ブレークポイントに到達した際に実行するコマンドを `breakpoint command` コマンドで管理できます。

```
(lldb) breakpoint command <subcommand> [<options>]
```

`subcommand` には `add`, `delete`, `list` を指定できます。

6.1.5.1 add

コマンドを追加します。

```
(lldb) breakpoint command add [<options>] <breakpoint-id>
```

文法:

option name	description
`-o, --one-liner <command>`	コマンドを指定

6.1.5.2 delete

コマンドを削除します。

```
(lldb) breakpoint command delete <breakpoint-id>
```

6.1.5.3 list

設定されたコマンドの一覧を表示します。

```
(lldb) breakpoint command list <breakpoint-id>
```

6.1.6 ブレークポイントの保存

ブレークポイントをファイルに保存するには `breakpoint write` コマンドを使用します。

```
(lldb) breakpoint write [<options>] [<breakpoint-id-list>]
```

文法:

option name	description
`-a, --append`	保存先ファイルが既存の場合、書き加える
`-f, --file <file>`	保存先のファイルを指定

ブレークポイントを指定しない場合、すべてのブレークポイントを対象とします。

6.1.7 ブレークポイントの読み込み

`write` で保存したブレークポイントを読み込むには `breakpoint read` コマンドを使用します。

```
(lldb) break read [<options>]
```

文法:

option name	description
`-N, --breakpoint-name <name>`	名前を指定してその名前付きのブレークポイントのみ読み込む
`-f, --file <file>`	読み込み元のファイルを指定

ブレークポイントを指定しない場合、すべてのブレークポイントを対象とします。

6.2 ウォッチポイント

式の値を監視して変更があった際にプログラムを停止させます。

6.2.1 ウォッチポイントの設置

ウォッチポイントの設置には `watchpoint set` コマンドを使用します。

```
(lldb) watchpoint set <subcommand> [<options>] -- <varname>|<expr>
```

`subcommand` には `variable` と `expression` が指定でき、それぞれ変数、式の監視を行います。 文法:

option name	description
`-s, --byte-size <size>`	監視する値のメモリサイズを指定
`-w, --watch-type <type>`	ウォッチタイプを指定。`read|write|read_write` が指定可能

6.2.2 ウォッチポイントの削除

ウォッチポイントを削除するには `watchpoint delete` コマンドを使用します。

```
(lldb) watchpoint delete [<watchpoint-id-list>]
```

`watchpoint-id` の値は `watchpoint list` で確認できます。 ウォッチポイントを指定せずに実行すればすべてのウォッチポイントを対象にできます。

6.2.3 ウォッチポイントの無効化

ウォッチポイントを無効化するには `watchpoint disable` コマンドを使用します。

```
(lldb) watchpoint disable [<watchpoint-id-list>]
```

ウォッチポイントを指定せずに実行すればすべてのウォッチポイントを対象にできます。

6.2.4 ウォッチポイントの有効化

ウォッチポイントを有効化するには `watchpoint disable` コマンドを使用します。

```
(lldb) watchpoint enable [<watchpoint-id-list>]
```

ウォッチポイントを指定せずに実行すればすべてのウォッチポイントを対象にできます。

6.2.5 ウォッチポイント到達時実行コマンド

ウォッチポイントに到達した際に実行するコマンドを `watchpoint command` コマンドで管理できます。

```
(lldb) watchpoint command <subcommand> [<options>]
```

`subcommand` には `add`, `delete`, `list` を指定できます。

6.2.5.1 add

コマンドを追加します。

```
(lldb) watchpoint command add [<options>] <watchpoint-id>
```

文法:

option name	description
`-o, --one-liner <command>`	コマンドを指定

6.2.5.2 delete

コマンドを削除します。

```
(lldb) watchpoint command delete <watchpoint-id>
```

6.2.5.3 list

設定されたコマンドの一覧を表示します。

```
(lldb) watchpoint command list <watchpoint-id>
```

7 プログラムの再開

ブレークポイント等で停止したプログラムを様々な方法で再開できます。再開方法は主にステップ実行と継続実行に分かれ、ステップ実行では一定分だけ進んで停止し、継続実行では次のブレークポイントまで進みます。

7.1.1 継続実行

継続実行をするには `thread continue` コマンドを使用します。

```
(lldb) thread continue [<thread-index> ...]
```

`thread-index` を指定するとそのスレッドのみ継続実行できます。指定しない場合全てのスレッドが対象となります。`thread-index` の値は `thread list` コマンドで確認できます。

7.1.2 関数から帰る

関数から帰るには `thread step-out` コマンドを使用します。 現在選択中のフレームが帰るまで実行します。フレームの選択方法は Section 8.2 を参照してください。

```
(lldb) thread step-out [<thread-index>]
```

7.1.3 ソース行単位ステップオーバー

ソース行単位ステップオーバーをするには `thread step-over` コマンドを使用します。

```
(lldb) thread step-over [<thread-index>]
```

7.1.4 命令単位ステップオーバー

命令単位ステップオーバーをするには `thread step-inst-over` コマンドを使用します。

```
(lldb) thread step-inst-over [<thread-index>]
```

7.1.5 ソース行単位ステップイン

ソース行単位ステップインをするには `thread step-in` コマンドを使用します。

```
(lldb) thread step-in [<thread-index>]
```

ソース行単位ステップインをするには `thread step-in` コマンドを使用します。

```
(lldb) thread step-in [<thread-index>]
```

7.1.6 命令単位ステップイン

命令単位ステップインをするには `thread step-inst` コマンドを使用します。

```
(lldb) thread step-inst [<thread-index>]
```

7.1.7 until

特定の箇所まで実行したい場合には `thread until` コマンドを使用します。

```
(lldb) thread until [<options>] <linenum>
```

文法:

option name	description
`-a, --address <addr>`	停止箇所のアドレスを指定
`-f, --frame <frame-index>`	`until` の実行フレーム番号を指定

8 スタックフレーム

8.1 バックトレース

コールスタックを表示するには `thread backtrace` コマンドを使用します。

```
(lldb) thread backtrace
```

8.2 フレームの選択

フレームを選択するには `frame select` コマンドを使用します。

```
(lldb) frame select [<options>] [<frame-index>]
```

文法:

option name	description
`-r, --relative <offset>`	選択中のフレームからのオフセットで選択

8.3 フレーム情報の表示

フレーム情報を表示するには `frame info` コマンドを使用します。

```
(lldb) frame info
```

9 変数の表示

変数を表示するには `frame variable`

```
(lldb) frame variable [<options>]
```

文法:

option name	description
`-A, --show-all-children`	上限を無視して子を表示
`-D, --depth <count>`	表示する最大深さ
`-F, --flat`	フラットフォーマットで表示
`-G, --gdb-format`	GDB フォーマットで表示
`-L, --locationn`	位置情報を表示
`-O, --object-description`	言語ごとの説明 API を使用して表示
`-P, --ptr-depth <count>`	値をダンプする際のポインタの深さ
`-R, --raw-output`	フォーマットを行わない
`-S, --synthetic-type <bool>`	synthetic provider に従うかを表示
`-T, --show-types`	型を表示
`-V, --validate <bool>`	型チェックの結果を表示
`-Y [<count>], --no-summary-depth=[<count>]`	概要情報を省略する深さ
`-Z, --element-count <count>`	型が配列あるかのように表示
`-a, --no-args`	関数の引数を省略
`-c, --show-declaration`	変数の宣言情報を表示
`-d, --dynamic-type <none>`	オブジェクトの完全な動的型を表示
`-f, --format <fmt>`	フォーマット
`-g, --show-globals`	グローバル変数を表示
`-l, --no-locals`	局所変数を省略
`-r, --regex`	変数名を正規表現として解釈
`-s, --scope`	変数のスコープを表示
`-t, --no-recognized-args`	recognized function arguments を省略
`-y, --summary <name>`	変数の出力が使用すべきサマリーを指定
`-z, --summary-string <name>`	フォーマットに使用するサマリー文

9.1 ディスプレイの代用

LLDB には `display` の直接的なコマンドはありません(エイリアスとしては存在します)。 代わりに `target stop-hook` コマンドを使用して、停止時に実行するコマンドを設定できます。 `target stop-hook` で `frame variable` を追加することで `display` と同様の動作を行えます。

```
(lldb) target stop-hook add --one-liner "frame variable argc argv"
```

Perf

Chapter 3

1 導入

Perf は Linux 用のプロファイラツールです。

1.1 コマンド

perf は git のように `perf <command>`の形式で各種ツールを使用します。 サポートされるコマンドの一覧は `perf` で閲覧できます。

```
  $ perf

  usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

  The most commonly used perf commands are:
    annotate        Read perf.data (created by perf record) and display
annotated code
    archive         Create archive with object files with build-ids found
in perf.data file
    bench           General framework for benchmark suites
    buildid-cache   Manage build-id cache.
    buildid-list    List the buildids in a perf.data file
    c2c             Shared Data C2C/HITM Analyzer.
    config          Get and set variables in a configuration file.
    daemon          Run record sessions on background
    data            Data file related processing
    diff            Read perf.data files and display the differential
profile
    evlist          List the event names in a perf.data file
    ftrace          simple wrapper for kernel's ftrace functionality
    inject          Filter to augment the events stream with additional
information
    iostat          Show I/O performance metrics
    kallsyms        Searches running kernel for symbols
    kvm             Tool to trace/measure kvm guest os
    list            List all symbolic event types
    mem             Profile memory accesses
    record          Run a command and record its profile into perf.data
    report          Read perf.data (created by perf record) and display the
profile
    script          Read perf.data (created by perf record) and display
trace output
    stat            Run a command and gather performance counter statistics
    test            Runs sanity tests.
    top             System profiling tool.
    version         display the version of perf binary
    probe           Define new dynamic tracepoints
```

一部のコマンドはカーネルで特殊なサポートを必要とするため使用できない場合があります。各コマンドのオプションの一覧を`-h` で出力することができます。

例:

```
$ perf stat -h

Usage: perf stat [<options>] [<command>]

  -a, --all-cpus        system-wide collection from all CPUs
  -A, --no-aggr         disable CPU count aggregation
  -B, --big-num         print large numbers with thousands' separators
```

1.2 イベント

perf は測定可能なイベントのリストを表示することができます。イベントは複数のソースからなり、一つはコンテキストスイッチやマイナーフォルトなどのカーネルカウンタです。これをソフトウェアイベントと呼びます。

もう一つは Performance Monitoring Unit(PMU)と呼ばれるハードウェアです。PMU はサイクル数、リタイアした命令、L1 キャッシュミスなどのマイクロアーキテクチャイベントを測定するためのイベントリストを提供します。これらのイベントをハードウェアイベントと呼びます。

イベントの一覧は

```
$ perf list
```

で閲覧できます。

2 stat によるカウント

`perf stat` を使用することでプログラム実行時のイベントを集計できます。

```
$ perf stat -e <event>[,<event>]... <command>
```

で `command` 実行時の `event` の集計を行えます。

3 record によるサンプリング

`perf record` を使用することでプロファイル情報を収集して、`perf stat` よりも細かいソースコード、命令単位レベルで情報を見ることができます。

```
$ perf record [<options>] <command>
```

文法:

option name	description
`-g`	コールグラフをレコード
`-o, --output <file>`	出力ファイルを指定

4 report によるサンプルの解析

`perf record` で収集したサンプルを `report` コマンドで閲覧します。

```
$ perf report
```

文法:

option name	description
`-g, --call-graph`	コールグラフを表示
`-i, --input <file>`	読み込むファイルを指定
`--stdio`	TUI ではなく標準出力に表示