

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

1 デバッガ	4
2 GDB	5
2.1 GDB の起動	5
2.1.1 ファイルオプション	5
2.1.2 モードオプション	6
2.1.3 起動に GDB が行う動作	7
2.1.4 初期化ファイル	8
2.1.4.1 ホームディレクトリの初期初期化ファイル	8
2.1.4.2 システム全体の初期化ファイル	8
2.1.4.3 ホームディレクトリ初期化ファイル	9
2.1.4.4 ローカルディレクトリ初期化ファイル	9
2.2 GDB の終了	9
2.3 シェルコマンド	9
2.4 ロギング出力	9
3 GDB コマンド	10
3.1 コマンド構文	10
3.2 コマンド設定	10
3.3 コマンド補完	10
3.4 ファイル名引数	10
3.5 コマンドオプション	11
3.6 ヘルプ	11
4 GDB でプログラムを実行する	12
4.1 デバッグのためのコンパイル	12
4.2 プログラムの開始	12
4.3 プログラムの引数	13
4.4 プログラムの環境	13
4.5 プログラムの作業ディレクトリ	13
4.6 プログラムの入出力	14
4.7 すでに実行中のプロセスのデバッグ	14
4.8 子プロセスの終了	14
4.9 複数の下位接続とプログラムのデバッグ	15
4.9.1 inferior 固有のブレークポイント	15
4.10 複数スレッドのプログラムのデバッグ	15
4.11 フォークのデバッグ	16
4.12 チェックポイント、再起動	17
5 停止と継続	17
5.1 ブレークポイント、ウォッチポイント、キャッチポイント	17
5.1.1 ブレークポイントの設定	17
5.1.2 ウォッチポイントの設定	18
5.1.3 キャッチポイントの設定	19
5.1.4 ブレークポイントの削除	19

5.1.5 ブレークポイントを無効にする	19
5.1.6 ブレーク条件	20
5.1.7 ブレークポイントコマンドリスト	20
5.1.8 動的 printf	20
5.1.9 ブレークポイントをファイルに保存する方法	21
5.1.10 静的プロブポイントの一覧表示	21
5.1.11 ブレークポイントを挿入できません。	21
5.2 継続とステップ	21
5.3 関数とファイルのスキップ	22
5.4 シグナル	23
5.5 スレッドストップ	24
5.5.1 オールストップモード	24
5.5.2 ノンストップモード	24
5.5.3 プログラムを非同期実行する	24
5.5.4 スレッド固有のブレークポイント	25
5.5.5 システムコール割り込み	25
5.5.6 オブザーバモード	25
6 後ろ向きのプログラム実行	26
7 Inferior の実行の記録と再生	26
8 スタックの検証	28
8.1 スタックフレーム	28
8.2 バックトレース	28
8.3 フレームの選択	28
8.4 フレーム情報	29
8.5 それぞれのフレームにコマンドを適用する	29
9 ソースファイルの検証	29
9.1 リスト	29
9.2 位置指定	30
9.2.1 linespec	30
9.2.2 明示的位置	30
9.2.3 アドレス位置	31
9.3 編集	31
9.3.1 エディタを変更する	31
9.4 検索	31
9.5 ソースパス	31
9.6 機械語	32
9.7 ソース読み込み無効化	32
10 データの検証	32
10.1 式	33
10.2 あいまいな式	33
10.3 プログラム変数	33
10.4 人工配列	33

10.5 出力フォーマット	33
10.6 メモリ検査	34
10.7 メモリタグ	34
10.8 自動表示	34
10.9 表示設定	35
10.10 きれいな表示	37
10.10.1 pretty-printer 導入	37
10.10.2 pretty-printer の例	37
10.10.3 pretty-printer のコマンド	38
10.11 値の履歴	38
10.12 コンビニエンス変数	38
10.13 コンビニエンス関数	39
10.14 レジスタ	40
10.15 浮動小数点ハードウェア	41
10.16 ベクタユニット	41
10.17 OS 情報	41
10.18 メモリ領域属性	41
10.18.1 属性	42
10.18.1.1 メモリアクセスモード	42
10.18.1.2 メモリアクセスサイズ	42
10.18.1.3 データキャッシュ	42
10.18.1.4 メモリアクセスチェック	42
10.19 メモリーファイル間のコピー	42
10.20 コアファイルの生成	43
10.21 文字セット	43
10.22 ターゲットのデータをキャッシュする	43
10.23 メモリ検索	44
10.24 値のサイズ	44
11 最適化されたコードのデバッグ	44
11.1 インライン関数	44
11.2 テールコールフレーム	44

1 デバッグ

プログラムのバグ(bug)を取り除く(de-)ことをデバッグといいます。デバッグを行う手法はいくつかあり、例えばプログラム中に標準出力を行う命令を追加してデバッグを行う print デバッグと呼ばれる方法があります。デバッグはデバッグを支援するツールで、プログラムの任意箇所での停止や、変数の値の表示や変更、スタックトレースやメモリ内容の監視など高度な機能によりデバッグを支援します。

C 言語で書かれたプログラムに対応するデバッグはいくつか存在しており、有名なものに GDB と LLDB が存在します。このドキュメントではこの二つのデバッグについて基本的な使用方法の解説を行います。

2 GDB

GDB は Gnu Project のデバッガです。

2.1 GDB の起動

GDB を起動するには以下のいずれかのコマンドを使用します。起動後はコマンドを受け付けます。

```
gdb [options] [executable-file [core-file or process-id]]  
gfb [options] --args <executable-file> [inferior-arguments ...]
```

--args を指定する場合、実行可能ファイルの後の引数(inferior-arguments) が実行時に渡されます。例えば `gdb --args gcc -O2 -c foo.c` は `gcc -O2 -c foo.c` の実行にデバッガをアタッチします。

options に指定できるオプションは `gdb -h` で確認できます。

2.1.1 ファイルオプション

GDB が起動すると、options 以外の引数は実行ファイルとコアファイル(またはプロセス ID)を指定するものとして読まれます。つまり `-se`、`-c` としてです。

-symbols <file>, -s <file>

file からシンボルテーブルを読み取ります。

-exec <file>, -e <file>

file を実行ファイルとして読み込みます。

-se <file>

file からシンボルテーブルを読み取り、実行ファイルとして使用します。

-core <file>, -c <file>

file をコアダンプとして検査します。

-pid <number>, -p <number>

プロセス ID が number のプロセスにアタッチします。

-command <file>, -x <file>

file からコマンドを実行します。

-eval-command <command>, -ex <command>:

単一の GDB コマンドを実行します。複数回指定可能です。

-init-command <file>, -ix <file>

下位ファイルをロードする前かつ gdbinit ロード後に file からコマンドを実行します。

-init-eval-command <command>, -iex <command>

下位ファイルをロードする前か gdbinit ロード後に GDB コマンド command を実行します。

-early-init-command <file>, -eix <file>

出力生成前にファイルからコマンドを実行します。

-early-init-eval-command <command>, -eiex <command>

出力生成前に GDB コマンド command を実行します。

-directory <directory>, -d <directory>

directory をソースファイルとスクリプトファイルを検索するパスに追加します。

-readnow, -r

各シンボルファイルのシンボルテーブル全体を起動時に読み取ります。デフォルトではこの機能はオフになっています。

--readnever

各シンボルファイルのシンボルテーブルを読み取らないようにします。このオプションをつけるとシンボリックデバッグが実行できなくなります。

2.1.2 モードオプション

GDB はさまざまなモードで実行できます。

-nx, -n

初期化ファイルにあるコマンドを実行しません。

-nh

ホームディレクトリ初期化ファイルにあるコマンドを実行しません。システム全体及びカレントディレクトリの初期化ファイルは実行されます。

-quiet, -silent, -q

起動時のメッセージを表示しません。これらメッセージはバッチモードでも表示されません。コマンドによりこのオプションを有向化することもできます。

-batch

バッチモードで実行します。-x で指定したコマンドファイルのコマンドがすべて実行された後、終了コード 0 を返して終了します(-n が指定されていない場合は初期化ファイルのコマンドも実行されます)。ファイル内のコマンド実行中にエラーが発生した場合は 0 以外のステータスコードを返して終了します。

-batch-silent

バッチモードで実行し、かつ全く標準出力への出力を行いません。

-return-child-result

GDB の終了ステータスをデバッグ中のプロセスの終了コードにします。ただし (1)GDB が異常終了した場合、(2)ユーザが明示的に終了ステータスを指定した場合、(3)子プロセスが実行されないか終了しない場合(終了ステータスは-1 になる)の三つの場合を除きます。

-nowindows, -nw

GDB に GUI インターフェースがある場合、CUI のみを使用するように指定します。

windows, -w

GDB に GUI インターフェースがある場合、GUI インターフェースを使用します。

-cd <directory>

作業ディレクトリを directory に移動して実行します。

-data-directory <directory>, -D <directory>

directory をデータディレクトリ(GDB が補助ファイルを検索する場所)として実行します。

-fullname, -f

スタックフレーム表示時およびプロセス停止時に完全なファイル名と行番号を出力します。

-annotate <level>

GDB 内の注釈レベルを設定します。これはプロンプト、式の値、ソース行、その他の出力とともに GDB が出力する情報の量を制御します。レベル 0 が通常、1 が Gnu Emacs で使用され非推奨、レベル 3 は最大の注釈です。

--args

実行ファイル以降の引数をすべて下位のコマンドライン引数として渡します。

--baud <bps>, -b <bps>

GDB がリモートデバッグに使用するシリアルインターフェースの回線速度を設定します。

-l <timeout>

GDB がリモートデバッグに使用する通信のタイムアウト(単位:秒)を設定します。

-tty <device>, -t <device>

プログラムの標準入力と出力に device を使用して実行します。

-tui

TUI(Text User Interface) モードをアクティブにします。TUI はターミナル上の複数のテキストウィンドウを管理し、ソース、アセンブリ、レジスタ、およびコマンド出力を表示します。

-interpreter <interp>

制御プログラムまたはデバイスとのインターフェイスにインタープリター interp を使用します。このオプションは GDB をバックエンドとして GDB と連携によって設定されることを目的としています。

-write

実行ファイルとコアファイルを読み取り書き取りの両方で開きます。

-statistics

GDB は各コマンドを完了してプロンプトに戻った後、時間とメモリ使用量に関する統計情報を表示します。

-configuration

GDB はビルド時の構成パラメータの詳細を出力し、終了します。

2.1.3 起動に GDB が行う動作

セッション起動時に GDB が行う処理を以下に示します。

1. 基本的な内部状態を初期化します。
2. ホームディレクトリにある初期初期化ファイルが存在する場合、コマンドを読み取ります。

3. `-eiex` と `-eix` で指定されたコマンドとコマンドファイルを指定された順番に実行します。
4. コマンドラインで指定されたコマンドインタプリターを設定します。
5. システム全体の初期化ファイルと初期化ディレクトリからファイルを読み取ります。
6. ホームディレクトリ内の初期化ファイルを読み取り、ファイル内のすべてのコマンドを実行します。
7. `-iex` および `-ix` で指定されたコマンドとコマンドファイルを指定された順番に実行します。通常 `-ex` および `-x` を代わりに使用します。この方法では GDB 初期化ファイルが実行される前および `inferior` がロードされる前に設定を適用できます。
8. コマンドラインオプションとオペランドを処理します。
9. 現在の作業ディレクトリにある初期化ファイルを読み込んで実行します。
10. デバッグするまたはアタッチするプログラムまたはコアファイルが指定されている場合、GDB はプログラムまたはそのロードされた共有ライブラリ用に提供された自動ロードスクリプトをロードします。
11. `-ex` および `-x` で指定されたコマンド及びコマンドファイルを読み込んで実行します。
12. *history file* に記録されたコマンド履歴を読み取ります。

2.1.4 初期化ファイル

GDB 起動時に GDB はいくつかの初期化ファイルからコマンドを実行します。これらの初期化ファイルはコマンドファイルと同じ構文を使用し、同様に処理されます。

起動時にロードされる初期化ファイルのリストをロードされる順番で表示するには `gdb --help` が使用できます。

初期初期化ファイルは初期化プロセスの非常に速い段階でロードされます。ここでは `set` または `source` コマンドのみを配置できます。

ほかの一般の初期化ファイルは任意のコマンドを実行できます。

2.1.4.1 ホームディレクトリの初期初期化ファイル

GDB は最初にこれを探します。GDB がホームディレクトリ内を検索する場所はいくつかあり、これらの場所は順番に検索され、最初に見つかったファイルのみをロードします。MacOS 以外では以下の場所が検索されます。

- `$XDG_CONFIG_HOME/gdb/gdbealynit`
- `$HOME/.config/gdb/gdbealynit`
- `$HOME/.gdbalynit`

`-nx`, `-n` オプションでこれらの初期初期化ファイルを読むことを阻止できます。

2.1.4.2 システム全体の初期化ファイル

以下の二か所が検索され、これらは常にチェックされます。

system.gdbinit

単一のシステム全体初期化ファイルです。 `--with-system-gdbinit` オプションで設定できます。

system.gdbinit.d

ディレクトリです。

2.1.4.3 ホームディレクトリ初期化ファイル

システム全体初期化ファイルを読んだ後、これを探します。以下の場所を検索し、最初に見つかったファイルのみをロードします。MacOS 以外では以下の場所が検索されます。

- `$XDG_CONFIG_HOME/gdb/gdbinit`
- `$HOME/.config/gdb/gdbinit`
- `$HOME/.gdbinit`

2.1.4.4 ローカルディレクトリ初期化ファイル

カレントディレクトリで `.gdbinit` ファイルを検索します。 `-x`, `-ex` で指定したコマンドを除いて最後にロードされます。すでにホームディレクトリ初期化ファイルとして読み込まれている場合は再度ロードされることはありません。

2.2 GDB の終了

GDB を終了するには `quit [expression]`, `exit [expression]` または `q` または `ctrl+d` で終了できます。 `expression` に指定した値は終了コードとして帰ります。 `ctrl+c` は実行中の GDB コマンドアクションを終了します。

2.3 シェルコマンド

GDB 起動中にシェルコマンドを使用することができます。

```
shell <command-string>
```

```
!<command-string>
```

`pipe` 命令を使用して `gdb` の出力を他のプログラムに繋ぐことができます。

```
pipe [command] | <shell_command>
```

```
| [command] | <shell_command>
```

```
pipe -d <delim> <command> <delim> <shell_command>
```

```
| -d <delim> <command> <delim> <shell_command>
```

`command` が `|` を含むときには `-d` で別の記号(列)を指定します。

2.4 ロギング出力

GDB の出力をファイルに行うことができます。GDB にはロギングを制御するコマンドがいくつか用意されています。

set logging enabled [on|off] ロギングのオンオフ切り替え

set logging file <file> 現在のログファイルの名前を変更。デフォルト値は `gdb.txt`

set logging overwrite [on|off] 上書きか書き足しか(`on` で上書き)。デフォルト値は `off`

set logging redirect [on|off] `on` にすると GDB の出力がログファイルにのみ行われる。デフォルト値は `off`

set logging debugredirect [on|off] on にすると GDB デバッグの出力がログファイルにのみ行われる。デフォルト値は off
show logging ロギングの設定を表示する

3 GDB コマンド

GDB コマンドは曖昧性がなければコマンド名の最初の数文字のみで使用できます。また、ret(エンター)を入力すると特定の GDB コマンドを繰り返し実行できます。また、TAB キーによる補完機能が有効です。

3.1 コマンド構文

GDB コマンドは一行の長さ無制限の入力です。command [args] の形をしています。run など一部コマンドを除いて空白行を入力すると直前のコマンドを繰り返します。list 及び x コマンドでは引数が変わります(???)。

3.2 コマンド設定

多くのコマンドは変数及び設定で動作が変わります。これらの設定は set コマンドで変更できます。

gdbinit ファイルに書き込むことで初期化時に設定できますし、対話中にコマンドを実行して設定することもできます。

with コマンドを使用して、コマンド呼び出しの期間中一時的に設定を変更することもできます。

```
with <setting> [value] [-- command]
w <setting> [value] [-- command]
```

3.3 コマンド補完

GDB では TAB キーによる補完が有効です。候補が唯一の場合は自動で入力が保管され、複数ある場合は候補が表示されます。TAB を二回押して候補を表示する代わりに esc ?で表示することもできます。

以下のコマンドで補完候補の最大数を設定できます。デフォルト値は 200 です。

```
set max-completions <limit>
```

limit には整数値または unlimited が指定できます。

```
show max-completions
```

で現在の設定を確認できます。

3.4 ファイル名引数

ファイル名をコマンドの引数として渡す場合、ファイル名に空白、ダブルクォート、シングルクォートが含まれていない場合は単純な文字列として記述できます。これらが含まれている場合、いくつか方法があります。

- GDB に任せる
- エスケープを使う
- クォートで囲う

3.5 コマンドオプション

一部コマンドは先頭に-がついたオプションを受け付けます。コマンド名と同様に、明確な場合は省略形を使うことができます。また、補完も効きます。

一部コマンドの引数にハイフンを含む場合は--を使うことでそれ以降の引数をオプションとして解釈しなくなります。

3.6 ヘルプ

help コマンドを使用してコマンドのヘルプを閲覧できます。

help, h

引数なしの help コマンドはコマンドのクラスのリストを表示します。

help <class>

ヘルプクラスを指定するとそのクラスの個々のコマンドのリストを表示します。

help <command>

コマンドを指定するとそのコマンドの短い使用方法を表示します。

apropos [-v] <regexp>

コマンド、エイリアス及びそのドキュメントを検索し、引数で指定した正規表現を検索します。見つかったすべてを表示します。-v オプションをつけるとドキュメントの一致部分をハイライトして表示します。

complete <args>

コマンドの先頭部分の一致候補を表示します。

info, show. set コマンドを使用して、プログラムの状態や GDB の状態を設定および照会することができます。

info, i

プログラムの状態を表示します。help info でサブコマンドの一覧を閲覧できます。

set

式の結果を環境変数に割り当てます。

show

GDB の状態を表示します。set できるものは大体 show できます。

show にあって set できないものを以下に示します。

- version

バージョン情報を表示します。

- copying

著作権表示を行います。

- warranty

保証情報を表示します。

- configuration

GDB のビルド情報を表示します。

4 GDB でプログラムを実行する

GDB でプログラムを実行するにはコンパイル時にデバッグ情報を付与する必要があります。

任意の環境で、引数を指定して GDB を起動できます。ネイティブデバッグではプログラムの IO をリダイレクトしたり、実行中プロセスをデバッグしたり、子プロセスを強制終了したりできます。

4.1 デバッグのためのコンパイル

プログラムを効果的にデバッグするにはコンパイル時にデバッグ情報を生成する必要があります。この情報はオブジェクトファイルに保存され、各変数、関数のデータ型と実行可能コード内のソース行番号とアドレスの対応関係が記述されます。

デバッグ情報の生成は `-g` オプションで行うことができます。

GCC では `-g` オプションは `-O`(最適化オプション)と併用できます。

4.2 プログラムの開始

run, r

GDB でプログラムを実行するには `run` コマンドを使用します。このコマンドを使用するには GDB 起動時またはコマンドでプログラムを指定する必要があります。

引数

`run` コマンドの引数はそのままプログラムのコマンドライン引数として渡されます。

環境

プログラムは GDB から環境を継承します。 `set` コマンドで環境を変更することもできます。

作業ディレクトリ

`set cwd` でプログラムの作業ディレクトリを設定できます。設定しない場合、GDB の作業ディレクトリを引き継ぎます。リモートデバッグの場合にはリモートサーバの作業ディレクトリを引き継ぎます。

標準入出力

通常、プログラムの標準入出力は GDB と同じになります。 `tty` コマンドで別のデバイスを設定することもできます。

`run` コマンドで実行したプログラムは直ちに実行を開始します。

GDB はシンボルファイルの変更を検出し、再読み込みを行います。

start

`start` コマンドはメインプロシージャにブレークポイントを設置して `run` します。引数の扱いは `run` と同様です。

starti

start と同様ですが、ブレークポイントの位置は最初の命令です。

```
set exec-wrapper <wrapper>
show exec-wrapper
unset exec-wrapper
```

を使用してデバッグ用プログラムの起動します。つまり Shell コマンド `exec wrapper program` を実行します。

```
set startip-with-shell
set startip-with-shell on
set startip-with-shell off
show startip-with-shell
```

プログラムをシェルで実行します。

```
set auto-connect-native-target
set auto-connect-native-target on
set auto-connect-native-target off
show auto-connect-native-target
```

現在の下位プロセスがターゲットに接続されていないときにローカルマシンで実行します。

```
set disable-randomization
set disable-randomization on
```

プログラムの仮想アドレス空間のネイティブランダム化をオフにします。

4.3 プログラムの引数

プログラムへの引数は `run` コマンド実行時に指定します。指定しなかった場合は以前の `run` 実行時の引数または `set args` で指定した引数を使用します。

```
set args
show args
```

4.4 プログラムの環境

環境は、環境変数とその値のことを指します。

```
path <directory>
```

環境変数の先頭にディレクトリを追加します。GDB が使用する環境変数の値は変化しません。

```
show paths
```

実行可能ファイルの検索パスのリストを表示します。

```
show environment [varname]
```

環境変数 `varname` の値を表示します。指定しない場合はすべての環境変数とその値を表示します。

```
set environment varname [=value]
```

環境変数 `varname` の値を設定します。値はプログラムに対してのみ変更され、GDB が読む変数の値は変わりません。

4.5 プログラムの作業ディレクトリ

```
set cwd [directory]
```

下位の作業ディレクトリを `directory` に変更します。

```
show cwd
```

下位プロセスの作業ディレクトリを表示します。

```
cd
```

GDB の作業ディレクトリを変更します。

```
pwd
```

GDB の作業ディレクトリを表示します。

4.6 プログラムの入出力

デフォルトでは GDB が実行するプログラムは GDB と同じターミナルに入出力を行います。

```
info terminal
```

プログラムが使用している端末モードについての情報を表示します。

`run` コマンドでシェルのリダイレクト機能を使えます。

```
run > <output file>
```

`tty` コマンドで入出力の場所を指定できます。

```
tty <file>
```

```
set inferior-tty <tty>
```

```
show inferior-tty
```

デバッグ対象のプログラムの `tty` を設定、表示します。

4.7 すでに実行中のプロセスのデバッグ

```
attach <pid>
```

すでに実行中のプロセスのプロセス ID を指定してデバッグをアタッチします。

```
set exec-file-mismatch 'ask|warn|off'
```

```
show exec-file-mismatch
```

GDB がロードした実行ファイルとアタッチしたプログラムの実行ファイルが一致するか確認した際に不一致だった場合の挙動を設定します。 `ask` は警告を出しプロセスの実行ファイルをロードするか確認します。 `warn` は警告を表示のみします。 `off` は不一致確認を行いません。

GDB はプロセスにアタッチするとそのプロセスを停止します。続行するには `continue` をします。

```
detach
```

プロセスからデタッチします。

4.8 子プロセスの終了

```
kill
```

GDB で実行されている子プロセスを終了します。

4.9 複数の下位接続とプログラムのデバッグ

GDB では 1 セッションで複数のプログラムを実行、デバッグできます。一部システムでは同時に行えます。

GDB では各プログラムの状態を `inferior` と呼ばれるオブジェクトで管理します。

```
info inferior
```

現在存在する `inferior` を表示します。

```
inferior
```

現在の `inferior` の情報を表示します。

```
info connection
```

現在開いているターゲット接続を表示します。

```
inferior <infno>
```

現在の `inferior` を `infno` に変更します。

```
add-inferior [-copies <n>] [-exec <executable>] [-no-connection]
```

実行ファイルを `executable` とする `inferior` を `n` 個追加します。

```
clone-inferior [-copies <n>] [infno]
```

`inferior` を `n` 個コピーします。

```
remove-inferiors infno...
```

`inferior` を削除します。

```
detach inferior infno...
```

```
kill inferior infno...
```

`inferior` を指定して `detach`, `kill` します。

```
set print inferior-events [on|off]
```

```
show print inferior-events
```

`inferior` のプロセスが開始または終了したときに通知を受け取ります。

```
maint info program-spaces
```

GDB によって管理されているすべてのプログラムスペースのリストを出力します。

4.9.1 inferior 固有のブレークポイント

複数の `inferior` プロセスをデバッグする場合、すべての `inferior` プロセスにブレークポイントを設定するか、個別にするか選択できます。

```
break <locspace> inferior inferior-id
```

```
break <locspace> inferior inferior-id if ...
```

4.10 複数スレッドのプログラムのデバッグ

GDB はマルチスレッドデバッグを行うために以下の機能を提供します。

- 新しいスレッドの自動通知
- スレッドを切り替えるコマンド
- 既存のスレッドの情報を表示するコマンド
- スレッドのリストにコマンドを適用するコマンド
- スレッド固有のブレークポイント
- スレッド開始、終了時のメッセージ設定

GDB では複数スレッドを観察できます。今見えているスレッドをカレントスレッドといいます。デバッグコマンドはカレントスレッドに適応されます。

新しいスレッドを検出するとスレッド識別子を表示します。

```
info threads [-gid] [thread-id-list]
```

スレッドの情報を表示します。引数を指定しなければすべてのスレッドの情報が表示されます。-gid を指定するとグローバルスレッド番号も表示します。

```
thread <tid>
```

カレントスレッドを tid のスレッドに変更します。

```
thread apply [thread-id-list | all [-ascending]] [flag]... <command>
```

指定したスレッド全体にコマンドを適用します。フラグに設定できるのは以下のとおりです。

-c コマンド内のエラーを表示してその後のコマンドは続行されます。

-s コマンド内のエラーを無視します。

-q スレッド情報を表示しません。

```
taas [option]... <command>
```

thread apply all -s [option]... <command>のショートカット。

```
tfaas [option]... <command>
```

thread apply all -s -- frame apply all -s [option]... <command>のショートカット。

```
thread name [name]
```

カレントスレッドに名前をつけます。名前を指定しない場合は名前を削除します。

```
thread find [regex]
```

指定した正規表現と一致する名前またはシステムタグを持つスレッドを検索して表示します。

```
set print thread-event [on|off]
```

```
show print thread-event
```

スレッド開始、終了時のメッセージを有効または無効にします。

4.11 フォークのデバッグ

```
set follow-fork-mode <mode>
```

fork の呼び出しに対する応答を設定します。

- parent 元のプロセスは fork のあとデバッグされます。子プロセスは妨げられずに実行されます。デフォルト。
- child 新しいプロセスが fork のあとにデバッグされます。親プロセスは妨げられずに実行されます。

```
set detach-on-fork 'on|off'
```

- on 子プロセスは切り離され、独立して実行されます。(デフォルト)

- `off` 両方のプロセスが GDB の制御化に置かれます。片方のプロセスは中断されま
す。(別 inferior)

4.12 チェックポイント、再起動

デバッグ中の状態にチェックポイントをおいて戻ることができます。

`checkpoint`

現在の状態にチェックポイントを起きます。

`info checkpoint`

設置されたチェックポイントの一覧を表示します。

`restart <checkpoint-id>`

チェックポイントに戻ります。

`delete checkpoint <checkpoint-id>`

チェックポイントを削除します。

5 停止と継続

デバッガを使用することでプログラムを停止することができます。

5.1 ブレークポイント、ウォッチポイント、キャッチポイント

- ブレークポイント

プログラム中の特定の場所で、そこに到達すると停止する。

- ウォッチポイント

式の値が変化したときに停止する。

- キャッチポイント

例外やライブラリロードなどで停止する。

5.1.1 ブレークポイントの設定

ブレークポイントは `break`, `b` コマンドで設定できます。さらに変数 `$bpnum` にブレークポイントの数が保存されています。

一つのブレークポイントが複数のコードの位置にマッピングされることがあります。例えば C++ の `template` やオーバーロードなど。その場合、設定時にその数を出力します。

デバッグ中のプログラムがブレークポイントに到達すると、変数 `$_hit_bpnum` と `$_hit_locno` がセットされます。

`break, b`

引数無しで `break` コマンドを実行すると、選択されたスタックフレームの次に実行される命令にブレークポイントが設置されます。

`break ... [-force-condition] if <cond>`

条件付きブレークポイントを設定します。このブレークポイントに到達したとき、`cond` の式がゼロでない場合にプログラムは停止します。 `-force-condition` を指定すると、式 `cond` が無効な式でもブレークポイントを設置します。

`tbreak args`

一回限りのブレークポイントを設置します。引数は `break` と同じです。このブレークポイントに一回プログラムが到達するとそのブレークポイントは自動的に消去されます。

`hbreak args`

ハードウェアブレークポイントを設置します。

`thbreak args`

一回限りのハードウェアブレークポイントを設置します。

`rbreak <regex>`

正規表現 `regex` にマッチするすべての関数にブレークポイントを設置します。`regex` に `.` を指定すればすべての関数にブレークポイントを設置できます。

`rbreak <file>:<reex>`

ファイル名を指定して `rbreak` を実行します。

`info breakpoints [list...]`

`info break [list...]`

全てのブレークポイント、ウォッチポイント、トレースポイント、キャッチポイントを表示します。`list` を指定すると指定したものだけを表示できます。

ブレークポイントは共有ライブラリを読み込んだ場合などに再計算されます。また、共有ライブラリロード以前にブレークポイントを設定しておくことも可能です。

`set breakpoint pending auto`

通常の動作です。GDB がロケーションを解決できない場合、作成するかどうかをユーザに問い合わせます。

`set breakpoint pending on`

`set breakpoint pending off`

`on` の場合、解決できなくても作成します。`off` ではしません。以上の設定はブレークポイントを設定するときにだけ適用されます。一度設置されたブレークポイントは自動で再計算されます。

`set breakpoint auto-hw 'on|off'`

自動でハードウェアブレークポイントを使用するかどうかの設定です。

`set breakpoint always-inserted 'off|on'`

`off` がデフォルト値です。プログラムが停止したときにブレークポイント用に書き換えたプログラムコードを元に戻すかどうかの設定です。

5.1.2 ウォッチポイントの設定

ウォッチポイントで監視できるものは以下の通りです。

- 単一の変数
- 適切なデータ型にキャストされたアドレス
- 式

ウォッチポイントはその計算が可能になる前から設定できます。そして有効な値になったときにプログラムを停止します。

5.1.3 キャッチポイントの設定

キャッチポイントを使用することでプログラムの例外や共有ライブラリロードなどのイベントによりデバッガに停止させることができます。

`catch <event>`

`event` が発生すると停止します。イベントは以下の通りです。

`throw [regexp]`

`rethrow [regexp]`

`catch [regexp]`

C++の例外が投げられた、再び投げられた、キャッチされた。 `regexp` が与えられている場合、その正規表現にマッチする例外だけがキャッチされます。

`syscall [name | number | group:groupname | g:groupname]`

システムコール発行時または復帰。

`fork`

`vfork`

`fork` および `vfork` 呼び出し

`load [regexp]`

`unload [regexp]`

共有ライブラリの読み込み、アンロード

`signal [signal... | 'all']`

シグナル発行。

`tcatch`

一回限りのキャッチポイントを設置

5.1.4 ブレークポイントの削除

`clear [locspec]`

引数を指定しない場合、次の命令のブレークポイントを削除します。指定した場合、そのブレークポイントを削除します。

`delete [breakpoints] [list...]`

引数で指定したブレークポイント、ウォッチポイント、キャッチポイントを削除します。引数を指定しない場合、すべて削除します。

5.1.5 ブレークポイントを無効にする

ブレークポイント、ウォッチポイント、キャッチポイントには以下の状態があります。

- 有効
 - 有効なブレークポイント。
- 無効
 - 無効なブレークポイント。
- 一度だけ有効
 - 一度プログラムが停止すると無効になる。
- 何回か有向
 - 指定した回数プログラムが停止すると無効化される。

- 有効のち削除
 - 一度プログラムが停止すると削除される。

```
disable [breakpoints] [list...]
enable [breakpoints] [list...]
```

指定したブレークポイントを無効化、有効化する。disable は引数を指定しない場合、何もおこらない。enable は全て有効になる。

```
enable [breakpoints] once <list...>
enable [breakpoints] count <count> <list...>
enable [breakpoints] delete <list...>
```

一度だけ有効、何回か有効、有効のち削除にする。

5.1.6 ブレーク条件

ブレークポイントには条件を付けてそれを満たす場合のみプログラムを停止することができます。

```
condition [-force] <bnum> <expression>
```

bnum のブレークポイントに条件を付与します。-force オプションをつけると現時点で無効な式も使用できます。expression を指定せずに実行すれば条件式を外すことができます。

```
ignore <bnum> <count>
```

ブレークポイントに到達した count 回は無視して、次からは停止します。

5.1.7 ブレークポイントコマンドリスト

ブレークポイントで停止したときに実行するコマンドを指定することができます。

```
commands [list...]
... <command-list> ...
end
```

list で指定したブレークポイントにコマンドリストを割り付けます。削除するにはコマンドリストを指定せずに実行します。

5.1.8 動的 printf

動的 printfdprintf ブレークポイントと printf を組み合わせたようなコマンドです。

```
dprintf <locspec>, <template>, <expression> [, <expression>...]
```

locspec で指定した場所にプログラムが到達すると template に従って式の expression を出力します。

```
set dprintf-style <style>
```

dprintf の以下のスタイルを指定します。

- gdb

GDB の printf のハンドル。%V の指定子が使えます。

- call

ユーザプログラムの関数を使用します。通常は printf。%V は使えません。

- agent

リモートデバッグエージェントに出力させます。%V は使えません。

```
set dprintf-function <function>
```

call のときに使用する関数を設定します。

```
set dprintf-channel <channel>
```

channel に空でない値を設定すると、fprintf-function の第一引数にそれを与えて評価します。

```
set disconnected-dprintf 'on|off'
```

agent のときに、ターゲットが切断されたときに dprintf の実行を続けるかどうかの設定です。

5.1.9 ブレークポイントをファイルに保存する方法

```
save breakpoints [<filename>]
```

filename のファイルにブレークポイントを保存します。この際、コマンドやカウンタも保存されます。これを読むには、source コマンドを使用します。式付きのウォッチポイントは無効で失敗する場合があります。

5.1.10 静的プロブポイントの一覧表示

GDB は SDT(Statically Defined Tracing; 静的定義トレース)をサポートしています。プロブは小規模なランタイムコードやフットポイント、動的再配置をデザインします。

現在以下のタイプのプロブが ELF 互換システムで実装されています。

- SystemTap アセンブリ、C、C++に対応
- DTrace C、C++に対応

5.1.11 ブレークポイントを挿入できません。

ハードウェアブレークポイントを挿入しすぎるとこのエラーが出ます。無効化または削除してください。

5.2 継続とステップ

継続は通常の実行と同様にプログラムが終了するまで実行することで、ステップは 1" ステップ"だけ実行することを意味します。ここで 1 ステップは一行だったり位置命令だったりします。

```
continue [ignore-count]
```

```
c [ignore-count]
```

```
fg [ignore-count]
```

次の停止場所まで継続実行します。ignore-count を指定するとその回数分ブレークポイントを無視します。3つのコマンドは完全に同じ動作をします。

```
step [count]
```

```
s [count]
```

ソースコード上の次の行までを実行します。関数呼び出しでは、その内部にデバッグ情報があればそこで停止します。また、停止する位置はソース行の最初の命令です。

step は関数の行番号情報がある場合にだけ関数に入ります。

count を指定するとその回数分 step を実行します。

next [count]

現在のスタックフレーム内の次のソースコード行に進みます。つまり関数の内部では停止しません。

set step-mode [on|off]

on のときデバッグ行情報を含まない関数の最初の命令で停止します。off の場合にはデバッグ情報を含まない関数はスキップされます。off がデフォルトです。

finish

fin

選択したスタックフレームの関数がリターンするまで実行を続けます。戻り値がある場合にはそれを表示します。

set print finidh [on|off]

finish 終了時に戻り値を表示するかどうかの設定です。デフォルトは on です。

until

u

現在のスタックフレームで、現在の行を過ぎたソース行に到達するまで継続実行する。until 中にジャンプ命令があったとき、PC がジャンプアドレスより大きくなるまで継続することを除いて next と同じである。ループの最後の行で until をすればループを抜けるまで実行することができるが、機械語の配置によっては直感通りの動作をしない場合がある。

until <locspec>

u <locspec>

引数で指定した位置に到達するまたは現在のスタックフレームに戻るまで継続実行する。あくまで現在のスタックフレームで停止する。

advance <locspec>

引数で指定した位置まで継続実行する。until とことなり、スタックフレームは無関係である。

stepi [arg]

si

機械語命令を一つ実行します。引数は step とおなじです。

nexti [arg]

ni

機械語命令を一つ実行します。call 命令の場合は帰るまで実行します。引数は next と同じです。

5.3 関数とファイルのスキップ

skip [options]

指定されたものをスキップします。指定できるものは以下です。

- -file <file>, -fi <file>: file 内にある関数すべてをスキップします。

- `gfile <file-glob-pattern>, -gfi <file-glob-pattern>`: パターンにマッチするファイル内の関数をスキップします。
- `function <linespec> -fu <linespec>`: 指定場所を含むまたは名前を持つ関数をスキップします。
- `rfunction <regex>, -rfu <regex>`: 正規表現 `regex` にマッチする関数をスキップします。
- 指定なし: 現在の関数をスキップします。

```
skip function [linespec]
skip file [file]
```

このコマンドの実行後、ステップ実行では指定した関数またはファイル内の関数がスキップされます。ファイル名、`linespec` を指定しない場合、現在の関数/ファイルが指定されます。

```
info skip [range]
```

指定されているスキップ対象を表示します。

```
skip delete [range]
skip enable [range]
skip disable [range]
```

スキップ対象を削除、有効化、無効化します。

```
set debug skip [on | off]
```

ファイルや関数のスキップに関するデバッグ出力をするかどうかの設定です。

5.4 シグナル

GDB にはプログラム中で発生したシグナルを検出する機能があります。通常ではエラーでないシグナルはそのままプログラムに渡されます。

```
info signals <sig>
info handle
```

すべてのシグナルと、GDB がどう扱うかの表を出力します。`<sig>`を指定した場合はそれのみを出力します。

```
catch signal [signal...|'all']
```

シグナルにキャッチポイントを設定します。

```
handle signal [signal...] [keywords...]
```

シグナル検出時の挙動を設定します。キーワードに設定できるものは以下です。

- `nostop`: そのままプログラムの実行を継続します。`print` と併用できます。
- `stop`: プログラムを停止します。`print` と併用できます。
- `print`: シグナル検出の旨を表示します。
- `noprint`: シグナル検出の旨を表示しません。
- `pass, noignore`: シグナルをプログラムに渡します。
- `nopass, ignore`: シグナルをプログラムに渡しません。

5.5 スレッドストップ

5.5.1 オールストップモード

このモードでは GDB がプログラムを停止するたびに他のすべてのスレッドも停止する。再開する際も同様にすべてのスレッドが再開する。

```
set scheduler-locking <mode>
```

スケジューラロックモードを設定します。

- off: すべてのスレッドはいつでも実行できる。
- on: 実行再開時には現在のスレッドだけが再開する。
- step: ステップ実行時には on、それ以外では off のような動作をする。continue, until, finish などでも復帰できる。
- replay: replay モードでは on それ以外では off のような動作をする。

```
set schedule-multiple
```

実行コマンド発行時に、複数プロセスのスレッドの再開を許可するモードの設定。

5.5.2 ノンストップモード

このモードでは GDB がプログラムを停止すると、その当該スレッドのみが停止します。さらに実行コマンド各種は現在のスレッドにのみてきようされます。

ノンストップモードに入るには以下のコマンドを使用します。

```
set pagination off
```

```
set non-stop on
```

```
set non-stop 'on|off'
```

ノンストップモードをオン、オフにします。

```
continue -a
```

ノンストップモードで、すべてのスレッドに continue を適用します。continue 以外の実行コマンドは現在、-a をサポートしていません。

5.5.3 プログラムを非同期実行する

GDB の実行コマンドにはフォアグラウンド動作とバックグラウンド動作があります。フォアグラウンドではプログラムがあるスレッドが停止したことを報告するのを待ってから別のコマンドのプロンプトを表示します。バックグラウンドでは直ちにコマンドプロンプトを表示します。これにより実行中にコマンドを発行することができます。

ターゲットが非同期モードをサポートしていない場合、バックグラウンド系のコマンドはエラーを吐きます。

バックグラウンド実行を指定するには、コマンドに & をつけます。例えば continue& など。バックグラウンドを受け付けるコマンドは以下のとおりです。

- run
- attach
- step

- stepi
- next
- nexti
- continue
- finish
- until

バックグラウンド実行中のプログラムを中断するには以下のコマンドを使用します。

```
interrupt [-a]
```

オールストップモードではすべての、ノンストップモードでは現在のスレッドを一時停止します。-a をつけるとノンストップモードでもすべてのスレッドでプログラムを停止します。

5.5.4 スレッド固有のブレークポイント

プログラムが複数のスレ度を持つ場合、すべてのスレッドまたは特定のスレッドにブレークポイントを設置することができます。

```
break <locspec> [thread <thread-id>] [if ...]
```

thread <thread-id>を指定しない場合、すべてのスレッドにブレークポイントを設置します。指定した場合はそのスレッドにのみ設置されます。

スレッド固有のブレークポイントはそのスレッドがなくなった場合、自動的に削除されます。

5.5.5 システムコール割り込み

マルチスレッドプロセスのデバッグに GDB を使用すると副作用により、ブレークポイント停止やその他イベントのために使用するシグナルが原因でシステムコールが早期リターンする場合があります。

この問題はプログラム側でシステムコールの戻り値を使用して対応する必要があります。

5.5.6 オブザーバモード

ノンストップモードでビルドし、GDB による中断の影響を排除するためには変数を設定して、メモリ書き込みやブレークポイントの挿入などデバッガが状態を変更する動作をすべて無効にすることができます。

```
set observer 'on|off'
```

on の場合、以下のすべての変数が無効になり、ノンストップモードになります。off にすると通常のデバッグに復帰しますが、ノンストップモードのままです。

```
set may-write-register 'on|off'
```

GDB が print の代入式やジャンプコマンドでレジスタの値を変更するかどうかの制御。デフォルトは on です。

```
set may-write-memory 'on|off'
```

GDB が print の代入式などでメモリの内容を変更するかどうかの制御。デフォルトは on です。

```
set may-insert-breakpoints 'on|off'
```

GDB がブレークポイントの挿入を試みるかどうかの制御。デフォルトは on です。

```
set may-insert-tracepoints 'on|off'
```

GDB がトレースポイントの挿入を試みるかどうかの制御。デフォルトは on です。

```
set may-insert-fast-tracepoints 'on|off'
```

GDB がファストトレースポイントの挿入を試みるかどうかの制御。デフォルトは on です。

```
set may-interrupt 'on|off'
```

GDB が割り込みを試みるかどうかの制御。デフォルトは on です。

6 後ろ向きのプログラム実行

サポートされている環境であれば、プログラムの実行を元に戻すことができます。

```
reverse-continue [ignore-count]
```

```
rc [ignore-count]
```

逆順に continue します。

```
reverse-step [count]
```

逆順に step します。

```
reverse-stepi [count]
```

逆順に stepi します。

```
reverse-next [count]
```

逆順に next します。

```
reverse-nexti
```

逆順に nexti します。

```
reverse-finish
```

現在の関数の呼び出しポイントまで戻ります。

```
set exec-direction 'reverse|forward'
```

ターゲットの実行向きを指定します。

7 Inferior の実行の記録と再生

いくつかの OS ではレコードすることによって逆順の実行ができます。

```
record <method>
```

メソッドを指定してレコードします。メソッドは以下のとおりです。

- full: GDB ソフトウェアによる完全なレコードです。
- trace <format>: ハードウェア命令によるレコードです。Intel プロセッサでサポートされます。データはリングバッファに書き込まれるので限定的な巻き戻しのみ可能です。フォーマットは以下のとおりです。
 - bts: Branch Trace Store。
 - pt: Intel Processor Trace。実行トレースを圧縮して保存します。

record コマンドを使うにはプログラムを実行しておく必要があります。

ノンストップモードまたは非同期実行モードでは full はサポートされません。

```
record stop
```

レコードを停止します。ログはすべて削除され、Inferior は終了するか最終状態のま

まになります。リプレイモード中にこれを発行するとその時点から通常のデバッグに復帰します。

```
record goto 'begin|start|end'|<n>
```

指定した場所に戻ります。begin と start は同じ場所です。n は n 番目の命令です。

```
record save <filename>
```

レコードを保存します。

```
record restore <filename>
```

ファイル名から実行ログをリストアします。save で保存したものを読みます。

```
set record full insn-number-max <limit>|'unlimited'
```

最大のレコード容量を設定できます。デフォルトでは 200000 です。レコードが最大容量に達すると、一番最初の命令から順番に削除しながらレコードが進みます。

limit に 0 または unlimited が設定された場合、命令は削除されません。

```
set record full stop-at-limit 'on|off'
```

レコードが最大容量に達したときに停止して続行するかどうかを尋ねます。

```
set record full memory-query 'on|off'
```

GDB が full のレコードをするとき、命令によって引き起こされたメモリ変更を記録できない場合の動作を制御します。on の場合にはどうするかを尋ね、off の場合には無視します。

```
set record btrace replay-memory-access 'read-only|read-write'
```

リプレイ中にメモリにアクセスする際の btrace レコードメソッドの動作を制御します。read-only の場合、GDB は readonly メモリへのアクセスのみを許可します。read-write の場合、GDB は readonly および readwrite メモリへのアクセスを許可します。

```
set record btrace cpu <identifier>
```

プロセッサ・エラッタを回避するために使用するプロセッサを設定します。プロセッサ・エラッタとは、設計や製造に起因するプロセッサ動作の血管のことを指します。

引数の identifier は CPU 識別子で vendor:processor identifier という形か none, auto が指定できます。

```
set record btrace bts buffer-size <size>|'unlimited'
```

BTS 形式でのブランチトレースに要求されるリングバッファのサイズを指定します。デフォルト値は 64KB です。

```
set record btrace pt buffer-size <size>|'unlimited'
```

IPT でのリングバッファのサイズを設定します。デフォルト値は 16KB です。

```
info record
```

レコード方式によってさまざまな統計情報を表示します。

```
record delete
```

レコード大賞が過去で実行された場合、それ以降のログを削除し、現在のアドレスからレコードを再開します。

```
record instruction-history
```

レコードされたログから命令を逆アセンブルします。

```
set record instruction-history-size <size>|'unlimited'
```

record instruction-history で表示される命令の数を設定します。

```
record function-call-history
関数単位で実行履歴を表示します。
set record function-call-history-size <size>|'unlimited'
record function-call-history で表示される数を設定します。
```

8 スタックの検証

8.1 スタックフレーム

コールスタックはスタックフレームに分割され、各フレームは関数呼び出しに関するデータを保持します。

GDB は既存のスタックフレームにそれぞれレベルをつけます。レベルは最も内側のフレームが 0、それを呼んだフレームが 1... という風に付きます。

8.2 バックトレース

プログラムの呼び出し履歴をバックトレースと云います。

```
backtrace [option]... [qualifier]... [count]
bt [option]... [qualifier]... [count]
```

すべてのスタックフレームのバックトレースを表示します。count は正の値を指定すると内側いくつか、負の値では外側いくつかを表示します。option に指定できるものは以下のとおりです。

- -full: ローカル変数の値も表示する。
- -no-filters: Python フレームフィルタを実行しません。
- -hide: Python フレームフィルタで elide にされたフレームを表示しません。
- -past-main [on|off]: main 以降もバックトレースを続けるかどうか。backtrace past-main で設定可。
- -past-entry [on|off]: エントリポイント移行もバックトレースを続けるかどうか。backtrace past-entry で設定可。
- -entry-values 'no|only|preferred|if-needed|both|compact|default': 関数入力時の print の設定。print entry-values で設定可。
- -frame-arguments all|scalars|none: 非スカラーフレーム引数の print の設定。print frame-arguments で設定可。
- -raw-frame-arguments [on|off]: フレーム引数を生で表示するかどうか。print raw-frame-arguments で設定可。
- -frame-info auto|source-line|location|source-and-location|location-and-address|short-location: フレーム情報の print 設定。print framw-info で設定可。

qualifier は下位互換のための引数です。

マルチスレッド環境では、現在のスレッドのバックトレースが表示されます。複数のスレッドのバックトレースを表示するには thread apply を使用できます。

8.3 フレームの選択

```
frame [frame-selection-spec]
```

`f [frame-selection-spec]`

指定したフレームを選択します。指定子に指定できるものは以下です。

- `<num>`, `level <num>`: スタックフレームレベル。
- `address <stack-address>`: スタックアドレス。 `info frame` で確認できます。
- `function <function-name>`: 関数名でスタックを指定します。
- `view <stack-address> [pc-addr]`: GDB のバックトレースの一部ではないフレームを表示する。

`up [n]`

`down [n]`

現在選択中のフレームの `n` 個上(外側)、下(内側)のフレームを選択します。

8.4 フレーム情報

`info frame [frame-selection-spec]`

`info f [frame-selection-spec]`

フレーム情報を表示します。

`info args [-q] [-t <type_regexp>] [regexp]`

選択されたフレームの引数を表示します。 `-q` を指定するとヘッダー情報や引数が出力されなかった理由を説明するメッセージが非表示になります。 後ろ 2 つのオプションは引数の型または名前を指定できます。

`info locals [-q] [-t <type_regexp>] [regexp]`

選択されたフレームのローカル変数を表示します。 オプションは `info args` と同じです。

8.5 それぞれのフレームにコマンドを適用する

`frame apply [all|count|-count|level <level>...] [option]... <command>`

指定したフレームにコマンドを適用します。 `option` に指定できるものは以下のとおりです。

- `-past-main`: `main` より先もバックトレースを続けます。
- `-past-entry`: エントリポイント以降もバックトレースを続けます。
- `-c`: エラーがあったときに表示して、継続します。
- `-s`: エラーがあったときに表示せずに、継続します。
- `-q`: フレーム情報を表示しません。

`faas <comamnd>`

`frame applu all -s <command>`のエイリアス。

9 ソースファイルの検証

9.1 リスト

`list <linenum>`

linenum の行を中心に数行のソースコードを表示します。

```
list <function>
```

関数の開始点を中心にコードを表示します。

```
list [+|-|.]
```

+または引数なしの場合、最後に出力された行の続きを表示する。-では最後に出力された行の前を、.では選択中のフレーム内の実行ポイント周辺を表示する。

```
set listsize <count>|'unlimited'
```

list で出力される行数を設定します。

```
list <locspec>
```

locspec で指定した箇所を中心にソースコードを表示します。

```
list [<first>], [<last>]
```

first から last までのソースコードを表示します。いずれかを省略すると最初からまたは最後までとなります。

9.2 位置指定

GDB コマンドでプログラムのコードの場所を指定できます。ここでは指定の方法について説明します。

9.2.1 linespec

linespec はファイル名や関数名などソース場所のパラメータをコロンで区切ったリストです。

```
[filename:]linenum
```

ファイル名と行番号を指定します。filename に相対ファイル名を指定すると、同じ末尾成分を持つファイルがマッチします。ファイル名を指定しない場合、現在のファイルが指定されます。

```
-offset
```

```
+offset
```

現在の行からの相対位置で指定します。

```
[filename:]function[:label]
```

ファイル名、関数名、ラベルを指定します。filename 内にある関数 function の label がある行が指定されます。label を指定しない場合、関数本体の開始行が指定されます。C 言語では中括弧ははじめのある行が指定されます。

```
label
```

ラベルのみを指定します。この場合、現在のスタックフレームに対応する関数内の label の位置を指定します。

9.2.2 明示的位置

オプションと値を使って指定する方法です。

```
-source <filename>
```

ファイル名を指定します。-line または -function を併用する必要があります。

```
-function <function>
```

関数を指定します。

```
-qualified
```

-function で指定された関数名を完全修飾名として解釈します。

```
-label
```

ラベルを指定します。

```
-line <number>
```

行数を指定します。絶対値(符号なし)と相対値(符号あり)が指定できます。

9.2.3 アドレス位置

コードアドレスを指定する方法です。

```
expression
```

現在の作業言語で有効な式が受け付けられます。

```
['filename':]funcaddr
```

関数のアドレスです。C 言語では単に関数名です。ファイル名を指定することもできます。

9.3 編集

ソースファイルの行を編集できます。

```
edit <locspec>
```

locspec で指定した行を指定したプログラムで編集できます。

9.3.1 エディタを変更する

環境変数 EDITOR にエディタを指定すると edit で開かれるエディタを指定できます。

9.4 検索

正規表現でファイルを検索できます。

```
forward-search <regexp>
```

```
search <regexp>
```

```
fo <regexp>
```

最後にリストされた行の次の行から前向きに検索できます。見つかった行はリストされます。

```
reverse-search <regexp>
```

```
rev <regexp>
```

逆順に検索します。

9.5 ソースパス

```
directory [<dirname>...]
```

```
dir [<dirname>...]
```

ソースパスに dirname を追加します。引数なしで実行するとソースパスをリセットできます。

```
set directories <path-list>
```

ソースパスを path-list に設定します。\$cdir:\$cwd がない場合は追加されます。

```
set substitute-path <from> <to>
```

ソースパスの from を to に置換し、最後に追加します。

```
unset substitute-path [path]
```

パスが指定されている場合、そのパスを書き換えるルールを現在の置換ルールのリストから検索し、見つかった場合は削除します。パスを指定しない場合、すべて削除されます。

9.6 機械語

```
info line [<locspec>]
```

指定した(指定しない場合は現在の)行のコンパイル済みコードの開始アドレスと終了アドレスを表示します。info line のあともう一度同コマンドを実行すると次のソース行の情報が表示される。

```
disassemble ['/m|/s|/r|/b']
```

メモリの範囲をマシンコードとしてダンプします。/m, /s はソースとマシン命令を、/r, /b は生の命令を表示します。\\m は非推奨です。

```
set disassembler-options <option1>[,<option2>...]
```

ターゲット固有の情報を逆アセンブラに渡す設定です。

9.7 ソース読み込み無効化

```
set source open ['on|off']
```

GDB がソースコードへアクセスできるかどうかの設定です。デフォルトでは on です。

10 データの検証

```
print [[<options>] --] <expr>
```

```
print [[<options>] --] /<f> <expr>
```

式 expr の値を表示します。指定しない場合は、最後の値を表示します。options に指定できる値は以下です。

- --address [on|off]: アドレスを表示します。
- -array [on|off]: 配列のフォーマットをします。
- -array-indexes [on|off]: 配列の添え字を表示します。
- -characters <number-of-characters>|elements|unlimited: 表示する文字列の長さを制限します。
- -elements <number-of-elements>|unlimited: 表示する配列の要素数を指定します。
- -max-depth <depth>|unlimited: ネストした構造を表略する閾値を設定します。
- -nibbles [on|off]: バイナリを 4 ビットグループで表示するかどうか
- -memory-tag-violations [on|off]: メモリタグ違反に関する情報を表示する。
- -null-stop [on|off]: 文字配列の表示を NULL 文字で停止するかどうか。
- -object [on|off]: C++ の virtual function テーブルの表示設定
- -pretty [on|off]: 構造体の pretty フォーマットの設定
- -repeats <number-of-repeats>|unlimited: 繰り返しの要素表示の閾値の設定
- -static-member [on|off]: C++ の static メンバの表示設定
- -symbol [on|off]: ポインタ表示時のシンボル名の表示設定
- -union [on|off]: 構造体内部の共用体の表示設定
- -vtbl [on|off]: C++ の virtual functioni テーブルの表示設定

10.1 式

print などのコマンドで引数にとる式は使用中の言語の任意の式を受け付けます。

その他にも言語に依存せず以下の演算が用意されています。

@

メモリの一部を配列として扱います。

::

変数が定義されているファイルや関数を指定して変数を指定します。

{<type>} <addr>

addr の値のメモリ位置を type 型として解釈します。

10.2 あいまいな式

set multiple-symbols <mode>

式があいまいな場合の動作を設定します。設定できるものは以下の通りです。

- all: デフォルト。全ての選択肢を選択します。一位に選ぶ必要がある場合、メニューが表示されます。
- ask: あいまいさがある場合、常にメニューを表示します。
- cancel: あいまいさがある場合、エラーをはいてコマンドが中断されます。

10.3 プログラム変数

式の変数は選択中のスタックフレームで解釈されます。

<file>::

とすると他の場所の変数も指定できます。

10.4 人工配列

<arr>@<len>

の形で arr を最初の要素とする長さ len の配列として &arr からのメモリを表示します。

キャストでも同様の動作をさせることはできます。

10.5 出力フォーマット

デフォルトでは GDB は型に沿って値を成型して表示します。一方、フォーマットを指定して表示することもできます。

x

バイナリを 16 進数で表示します。

d

バイナリを 10 進数で表示します

u

バイナリを符号なし 10 進数で表示します。

o

バイナリを 8 進数で表示します。

t

バイナリを 2 進数で表示します。

a

アドレスを表示します。

c

値を整数値にキャストして文字列として表示します。

f

浮動小数として表示します。

s

可能であれば文字列として扱います。

z

ゼロ埋めされた 16 進数で表示します。

r

raw フォーマットで表示します。

10.6 メモリ検査

x[/[<n>][<f>][<u>]] [<addr>]

- n: 表示するメモリ量(単位は u で指定)を指定
- f: フォーマットを指定
- u: 単位を指定。指定できるもの:
 - b: Byte
 - h: Halfwords(2Bytes)
 - w: words(4Bytes)
 - g: Giant words(8Bytes)
- addr: 開始アドレス

10.7 メモリタグ

メモリ・タグは、ポインタを介したメモリ・アクセスを検証するために 1 対のタグを使用するメモリ保護技術である。タグは、アーキテクチャにもよるが、通常は数ビットからなる整数値である。

10.8 自動表示

ある式の値を頻繁に表示したい場合、自動表示が利用できます。

`display[/<fmt>] <expr>|<addr>`

自動表示リストに `expr` を追加します。

`undisplay <dnums>...`

自動表示リストから削除します。

`disable display <dnums>...`

自動表示を無効化します。

`enable display`

自動表示を有効化します。

`display`

現在のリストの上の式の値を表示します。

`info display`

自動表示リストを表示します。値は表示しません。

10.9 表示設定

GDB は表示方法について以下の設定を提供しています。

`set print address [on|off]`

スタックトレース、構造体の値、ポインタの値、ブレークポイントなどの場所を示すメモリアドレスを表示します。デフォルト値は on です。

`set print symbol-filename [on|off]`

on のときシンボルのソースファイル名と行番号をアドレスのシンボル形式で表示します。

`set print max-symbolic-offset <max-offset>|unlimited`

シンボリックアドレスを表示するオフセットの最大値を設定します。最大値以上のオフセットの場合は表示されません。0 と unlimited は等価で、前にシンボルがある限り常に表示します。

`set print symbol [on|off]`

あるアドレスに対応するシンボルがあれば、それを表示します。

`set print array [on|off]`

配列をきれいに整形して表示します。デフォルト値は off です。

`set print array-indexed [on|off]`

配列を表示する際に書く要素のインデックスを憑依します。デフォルト値は off です。

`set print nibbles [on|off]`

print コマンドを /t で表示する場合に 4bit で区切って表示します。

`set print characters <number-of-characters>|elements|unlimited`

GDB が表示する文字列の制限を設定します。elements を設定すると配列の大きさを分表示します。デフォルト値は elements です。

`set print elements <number-of-elements>|unlimited`

GDB が表示する配列の要素数の上限を設定します。デフォルト値は 200 です。

`set print frame-arguments <value>`

フレームを表示するときに引数の値をどのように表示するかを設定します。value に設定できる値は以下のとおりです:

- all: すべての引数が表示されます
- scalars: スカラー値の引数のみ表示します
- none: どの引数の値も表示しません。値は...で置き換えられます
- presence: 引数がある場合は...が、ない場合は何も表示されません

デフォルト値は scalars です。

`set print raw-frame-arguments [on|off]`

フレームの引数をきれいに整形されていない生の状態で表示します。

`set print entry-values <value>`

関数エントリ時のフレーム引数の値の表示を設定します。value に指定できる値は以下のとおりです:

- no: 実際のパタメータ値のみ表示し、エントリポイントからの値は表示しません

- `only`: エントリポイントからの値のみ表示し、実際の値は表示しません
- `preferrrd`: エントリポイントからの値を表示し、それが不明で実際の値が既知の場合それを表示します
- `if-needed`: 実際の値を表示し、それが既知でない場合エントリポイントからの値が既知ならばそれを表示します
- `both`: 常にエントリポイントからの値と実際の値の両方を表示します
- `compact`: 実際の値およびエントリポイントからの値のうち既知の値を表示します。どちらも未知の場合 `optimized out` を表示します。MI モードでない場合、両方の値が既知であれば短縮表記 `param=param@entry=VALUE` を表示します
- `default`: 常に実際の値を表示します。エントリポイントからの値も既知の場合それも表示します。MI モードでなければ短縮表記を使用します。

デフォルト値は `default` です。

```
set print frame-info <value>
```

フレームを表示するときに表示される情報を制御します。value に設定できる値は以下のとおりです:

- `short-location`: フレームレベル、PC(ソース行の先頭でなければ)、関数、引数を表示します
- `location`: `short-location` に加えソースファイルと行番号も表示します
- `location-and-address`: `location` に加え、ソース行の先頭でも PC を表示します
- `source-line`: PC(ソースの先頭でなければ)、行番号、ソース行を表示します
- `source-and-location`: `location` と `source-line` を表示します
- `auto`: 使用するコマンドによって自動的に表示される情報を決定します。

デフォルト値は `auto` です。

```
set print repeats <number-of-repeats>|unlimited
```

配列の繰り返し要素の表示を抑制する閾値を設定します。配列の連続した同一要素の数が閾値を超えると `<repeats n times>` を表示します。0 と `unlimited` は同等です。デフォルト値は 10 です。

```
set print max-depth <depth>|unlimited
```

ネストした構造体を省略記号に置き換える深さの閾値を設定します。

```
set print memory-tag-violations [on|off]
```

ポインタとアドレスを表示するときにメモリタグ違反に関する情報を表示します。

```
set print null-stop [on|off]
```

文字列を表示するときに最初の NULL で表示を停止するかどうかの設定です。デフォルト値は `off` です。

```
set print pretty [on|off]
```

構造体を整形して表示するかどうかの設定です。

```
set print raw-values [on|off]
```

値のための整形を行わず生の値を表示するかどうかの設定です。

```
set print sevenbit-strings [on|off]
```

8bit 文字を `\nnn` という形式で表示します。

```
set print union [on|off]
```

構造体や他のユニオンに含まれるユニオンを表示するかどうかの設定です。デフォルトは on です。

以下は C++ 関連の設定です。

```
set print demangle [on|off]
```

C++ の名前を型安全リンケージのためにアセンブラやリンカに渡す mangle 形式ではなくソース形式で表示します。デフォルト値は on です。

```
set print asm-demangle [on|off]
```

アセンブラコードの表示時に mangled 形式でなくソース形式で表示します。デフォルト値は off です。

```
set demangle-style [<style>]
```

C++ の名前を表現するためにエンコーディングスキームを選択します。style を省略すると設定可能なスタイルのリストが表示されます。デフォルト値は auto です。

```
set print object [on|off]
```

オブジェクトへのポインタを表示する場合、仮想関数テーブルを使用して、宣言された型ではなく、オブジェクトの実際の型を識別します。

```
set print static-members [on|off]
```

C++ オブジェクトの静的メンバを表示します。デフォルト値は on です。

```
set print pascal_static-members [on|off]
```

パスカルのオブジェクトを表示するときに静的メンバを表示します。デフォルト値は on です。

```
set print vtbl [on|off]
```

C++ 仮想関数テーブルをきれいに表示します。

10.10 きれいな表示

10.10.1 pretty-printer 導入

GDB が値を表示するとき、まずその値に対応する pretty-printer が登録されているか確認します。あればそれ呼び出し、なければ通常通り表示されます。

10.10.2 pretty-printer の例

C++ の `std::string` は pretty-printer なしでは

```
$1 = {
  static npos = 4294967295,
  _M_dataplus = {
    <std::allocator<char>> = {
      <__gnu_cxx::new_allocator<char>> = {
        <No data fields>}, <No data fields>
      },
    members of std::basic_string<char, std::char_traits<char>,
      std::allocator<char> >::_Alloc_hider:
    _M_p = 0x804a014 "abcd"
  }
}
```

のように表示されます。これを pretty-printer に通すと

```
$2 = "abcd"
```

と表示されます。

10.10.3 pretty-printer のコマンド

```
info pretty-printer [object-regexp [name-regexp]]
```

インストールされている pretty-printer を名前とともにリストアップします。

```
disable pretty-printer [object-regexp [name-regexp]]
```

```
enable pretty-printer [object-regexp [name-regexp]]
```

pretty-printer を無効化、有効化します。

10.11 値の履歴

print コマンドで出力された値は GDB の値履歴に保存されます。これにより他の式で値を参照することができます。値はシンボルテーブルが再読み込みまたは破棄されるまで保持されます。

過去の値は `$<num>` で参照できます。`$$<num>` とすれば最後の履歴から逆順に参照できます。単に `$` とすれば最新の履歴を参照します。

```
show values [<n>|+]
```

最後の十個の履歴を表示します。このコマンドは履歴を変更しません。数値を指定するとその番号から 10 個を、+ を指定すると前回表示した続きの 10 個を表示します。

10.12 コンビニエンス変数

GDB は、GDB の内部で値を保持し、参照するために使用できるコンビニエンス変数を提供します。

事前に定義されていない限り `$` から始まる名前なら何でもコンビニエンス変数になる。

```
set $foo = *obj_ptr
```

のように変数をセットできます。型はありません。

```
show convinience
```

コンビニエンス変数の一覧を表示します

```
init-if-undefined $<var> = <expression>
```

コンビニエンス変数がまだ定義されていなければ `expression` で初期化します。

以下は事前に定義されるコンビニエンス変数です。

```
$_
```

`x` コマンドで自動的に最後に調べたアドレスに設定されます。`$_` へのポインタです。

```
$__
```

`$_` の指すアドレスの値です。

```
$_exitcode
```

デバッグ対象のプログラムが正常終了すると終了コードに設定されます。このとき `$_exitcode` を `void` にリセットします。

`$_exitsignal`

キャッチされなかったシグナルによってデバッグ対象のプログラムが終了すると、そのシグナルの番号が設定されます。このとき `$_exitcode` を `void` にリセットします。

`$_exception`

例外のキャッチポイントでスローされる例外オブジェクトが設定されます。

`$_ada_exception$`

Ada の例外キャッチポイントでスローされる例外のアドレスがセットされます。

`$_probe_argc`

`$_probe_arg0 ... $_probe_arg11`

静的プローブの引数です。

`$_sdata`

追加で収集された静的トレースポイントデータが格納されます。

`$_siginfo`

追加のシグナル情報が格納されます。

`$_tlb`

MS-Windows 上でネイティブモードで作動しているアプリケーションをデバッグする場合または `qGetTIBAddr` リクエストをサポートしている `GDBServer` に接続している場合スレッド情報ブロックのアドレスが格納されます。

`$_inferior`

現在の `inferior` の数が格納されます。

`$_thread`

現在のスレッドのスレッド番号が格納されます。

`$_gthread`

現在のスレッドのグローバルスレッド番号が格納されます。

`$_inferior_thread_count`

現在の `inferior` でアクティブなスレッドの数が格納されます。

`$_gdb_major`

`$_gdb_minor`

GDB のメジャー、マイナーバージョン番号が格納されます。

`$_shell_exitcode`

`$_shell_exitsignal`

シェルコマンドの `exitcode` および `exitsignal` が格納されます。

10.13 コンビニエンス関数

コンビニエンス引数に似たコンビニエンス関数もあります。これらも式中で使うことができますが、GDB の内部で実装されています。

`$_isvoid(<expr>)`

`expr` が `void` かどうか判定します。

`$_gdb_setting_str(<setting>)`

GDB の設定の内容を文字列として取得します。

`$_gdb_setting(<setting>)`

GDB の設定の内容を取得します。型は設定によります。

`$_gdb_maint_setting_str(<setting>)`

メンテナンス設定変数版の `$_gdb_setting_str` です。

`$_gdb_maint_setting(<setting>)`

メンテナンス設定変数版の `$_gdb_setting` です。

`$_shell(<command_string>)`

シェルコマンドを実行します。

`$_memeq(<buf1>, <buf2>, <length>)`

`buf1`, `buf2` で与えられるアドレスの長さ(Byte)が `length` に等しいかどうかを判定します。

`$_regex(<str>, <regex>)`

文字列 `str` が正規表現 `regex` にマッチするかどうかを判定します。

`$_streq(<str1>, <str2>)`

2つの文字列が等しいかどうかを判定します。

`$_strlen(<str>)`

文字列の長さを返します。

`$_caller_is(<name>[, <number_of_frames>])`

呼び出し元の関数名が `name` であるかどうかを判定します。`number_of_frames` を指定すると特定のスタックフレームに注目します。指定しない場合は1が入ります。

`$_caller_matches(<regex>[, <number_of_frames>])`

`$_caller_is` の正規表現版です。

`$any_caller_is(<name>[, <number_of_frames>])`

呼び出し元の関数に `name` があるかどうかを判定します。

`$any_caller_match(<name>[, <number_of_frames>])`

`$any_caller_is` の正規表現版です。

`$_as_string(<value>)`

非推奨の関数です。代わりに `%v` の仕様が推奨されます。`value` の文字列表現を返します。

`$_cimag(<value>)`

`$_creal(<value>)`

複素数値の実部、虚部を返します。

`help function`

コンビニエンス関数の一覧を表示します。

10.14 レジスタ

`info registers [<reggroup>...|<regname>...]`

浮動小数点レジスタとベクタレジスタを除くすべてのレジスタの名前と値を表示します。`reggroup` を指定した場合、そのグループのレジスタに関して表示を行います。指定できるグループの一覧は `maint print reggroups` で確認できます。`regname` を指定した場合、その名前のレジスタに関して表示を行います。

`info all-registres`

すべてのレジスタの名前と値を表示します。

10.15 浮動小数点ハードウェア

`info float`

浮動小数点ユニットに関するハードウェア依存の情報を表示します。

10.16 ベクタユニット

`info vector`

ベクタユニットに関する情報を表示します。

10.17 OS 情報

`info auxv`

実行中のプロセスまたはコアダンプファイルである `inferior` の補助ベクタを表示します。

`info os [<infotype>]`

OS の情報を表示します。 `infotype` には GNU/Linux では以下の値を指定できます。

- `cpus` : すべての CPU、コアのリストを表示します。
- `files` : ターゲット上でオープンされているファイルディスクリプタのリストを表示します。
- `modules` : ターゲットにロードされているすべてのカーネルモジュールのリストを表示します。
- `msg` : ターゲット上のすべての System V メッセージキューのリストを表示します。
- `processes` : ターゲット上のプロセスのリストを表示します。
- `procgroups` : ターゲット上のプロセスグループのリストを表示します。
- `semaphores` : ターゲット上のすべての System V セマフォセットのリストを表示します。
- `shm` : ターゲット上のすべての System V 共有メモリ領域のリストを表示します。
- `sockets` : ターゲット上のすべてのインターネットドメインソケットのリストを表示します。
- `threads` : ターゲット上で実行中のスレッドのリストを表示します。

`infotype` を指定しない場合、すべての OS 情報が表示されます。

10.18 メモリ領域属性

`mem <lower> <upper> <attributes>...`

`lower` と `upper` で囲まれたメモリ領域を `attributes...` で定義し、GDB が監視する領域のリストに追加します。

`mem auto`

メモリ領域に対するユーザの変更をすべて破棄し、ターゲットが提供する領域があればそれを利用し、サポートしていない場合は使用しません。

`delete mem <nums>...`

GDB が監視する領域リストからメモリ領域を削除します。

`disable mem <nums>...`

```
enable mem <nums>...
```

GDB が監視する領域リストのメモリ領域の監視を無効化/有効化します。

```
info mem
```

定義されたすべてのメモリ領域の一覧を表示します。

10.18.1 属性

10.18.1.1 メモリアクセスモード

メモリアクセスモードは GDB がメモリ領域に対して行うアクセスのモードです。

```
ro
```

```
wo
```

```
rw
```

読み込み専用/書き込み専用/読み書きのモードです。デフォルト値は rw です。

10.18.1.2 メモリアクセスサイズ

```
8
```

```
16
```

```
32
```

```
64
```

指定した bit のメモリアクセスサイズを持ちます。

10.18.1.3 データキャッシュ

GDB がターゲットメモリをキャッシュするかどうかの設定です。

```
cache
```

```
nocache
```

キャッシュあり/なしです。

10.18.1.4 メモリアクセスチェック

```
set mem inaccessible-by-default [on|off]
```

メモリ範囲に明示的に記述されていないメモリ領域へのアクセスを拒否するかどうかの設定です。デフォルト値は on です。

10.19 メモリーファイル間のコピー

```
dump [format] memory <filename> <start_addr> <end_addr>
```

```
dump [format] value <filename> <expr>
```

メモリの start_addr から end_addr までの内容または expr の値を与えられたフォーマットで filename にダンプします。指定できるフォーマットは以下のとおりです。

- binary : 生のバイナリです。
- ihex : Intel Hex format です。
- srec : Motorola S-record format です。
- tekhex : Tektronix Hex format です。
- verilog : Verilog Hex format です。

```
append [format] memory <filename> <start_addr> <end_addr>
```

```
append [format] value <filename> <expr>
```

メモリの start_addr から end_addr までの内容または expr の値を与えられたフォーマットで filename に追加します。

```
restore <filename> [binary] <bias> <start> <end>
```

ファイルの内容をメモリにリストアします。生バイナリ以外の場合は自動でフォーマットを判定します。生バイナリを指定する場合は binary の指定が必要です。

bias がゼロでない場合、値がファイルに含まれるアドレスに追加されます。

10.20 コアファイルの生成

コアファイルは実行中のプロセスのメモリーイメージとプロセス状態を記録したファイルです。

```
generate-core-file [<file>]
```

```
gcore [<file>]
```

inferior のコアダンプを生成します。ファイル名を指定しない場合、core.pid の形式で命名されます。

```
set use-coredump-filter [on|off]
```

コアダンプ生成時に、/proc/pid/coredump_filter のフィルタを使用するかどうかを設定します。

```
set dump-excluded-mappings [on|off]
```

on の場合、VM_DONTDUMP でマークされたメモリマッピングをダンプします。デフォルトでは off です。

10.21 文字セット

```
set target-charset <charset>
```

現在のターゲットの文字セットを設定します。サポートされている文字セットは set target-set まで入力して TAB で補完すると確認できます。

```
set host-charset <charset>
```

現在のホストの文字セットを設定します。

```
set charset <charset>
```

ホスト、ターゲットの両方で文字セットを設定します。

```
set target-wide-charset <charset>
```

ターゲットのワイド文字セットを設定します。

10.22 ターゲットのデータをキャッシュする

```
set remotecache [on|off]
```

現在このオプションは無効です。

```
set stack-cache [on|off]
```

スタックアクセスのキャッシュを有効化/無効化します。デフォルトでは on です。

```
set code-cache [on|off]
```

コードセグメントへのアクセスのキャッシュを有効化/無効化します。デフォルトで

は on です。

```
info dcache [<line>]
```

現在の inferior のアドレス空間のデータキャッシュのパフォーマンスに関する情報を表示します。

```
set dcache size <size>
```

dcache のエントリ数の最大値を設定します。

```
set dcache line-size <line-size>
```

dcache のエントリのバイト数を指定します。2 のべきの値のみ指定できます。

```
maint flush dcache
```

キャッシュをフラッシュします。

10.23 メモリ検索

```
find [/sn] <start_addr>, +<len>, <val1> [, <val2>, ...]
```

```
find [/sn] <start_addr>, <end_addr>, <val1> [, <val2>, ...]
```

val で指定されたバイト列のメモリを検索します。検索は len または end_addr までが対象です。s, n はオプションです。

- s : サーチクエリサイズ。以下が設定できます。
 - b : Byte
 - h : Halfwords(2Bytes)
 - w : words(4Bytes)
 - g : giant words(8Bytes)
- n : マッチして表示する数の最大値を設定します。

10.24 値のサイズ

GDB が値を表示するときにその値の内容を保持するためにメモリ領域を確保します。

```
set max-value-size <bytes>|unlimited
```

GDB が値の内容に割り当てるメモリの最大サイズを設定します。

11 最適化されたコードのデバッグ

最適化されたコードは機械語とソースコードが単純に対応していない場合があります。

11.1 インライン関数

フレームを確認すれば関数がインライン展開されたかどうかを確認できます。

11.2 テールコールフレーム

関数の最後に関数呼び出しを行った場合、二重にリターンすることがあります。最適化でこれを一つのジャンプにまとめることがあり、これをテールコールといいます。

```
set debug entry-values [on|off]
```

関数エントリ時のフレーム引数値とテールコール時のフレーム引数の両方の解析メッセージを表示できるようになります。