Debugging Quickly & Painlessly with GDB

GDB

ハンドブック

O'REILLY®
オライリー・ジャパン

Arnold Robbins 著
千住 治郎 訳

GDBハンドブック

Arnold Robbins 著
千住 治郎 訳

O'REILLY®
オライリー・ジャパン

GDB

Pocket Reference

Arnold Robbins

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

はじめに

GNU デバッガ（GNU Debugger、GDB）は GNU ／ Linux システムや BSD システムで標準的に使用されているデバッガです。GNU ／ Linux、BSD 以外の Unix システムでも C コンパイラをインストールしており、一般的なオブジェクトファイルフォーマットのどれか1つにでも対応していればGDBは利用できます。さらにUnix以外のシステムにも対応しています。GDBは非常に豊富な機能を備えており、世界中の多くの開発者達に好んで使用されています。

このポケットリファレンスはGDBのコマンドラインの構文、初期化ファイル、式、変数、コマンドをすべて網羅したものです。またGDB以外に、内部でGDBを使用しているグラフィカルインタフェースを備えたデバッガも 2 種類紹介します。

GDBの詳細な解説はGDBのソースコードに付属しているドキュメントを参照してください。付属ドキュメント『Debugging with GDB: The GNU Source-Level Debugger』† （Richard M. Stallman、Roland Pesch、Stan Shebs 他著）は Free Software Foundation からも入手可能です。

表記について

ゴシック（サンプル）

強調や新出の用語を表します。

固定幅（sample）

ディレクトリ名、コマンド、プログラム名、関数、変数、オプションを示します。記述どおりに入力する文字はすべてこのフォントで示します。またファイルの内容やコマンドからの出力にも使用します。

固定幅ボールド（sample）

例中でユーザが記述どおりに入力すべきテキストを表します。

† 訳者注：邦訳『GDB デバッギング入門』アスキー刊、コスモ・プラネット訳

固定幅イタリック(*sample*)
構文やコマンド群で、一般的な引数やオプションを示すのに使います。つまりユーザが実際に使用する値や文字列で置き換えるべきものです。

$
Bash、Bourne、Korn シェルのプロンプトを表します。

プログラム(*N*)
プログラムのオンラインマニュアル(manpage) を表します。*N*はセクションを表し、例えばecho(1)と表記した場合はechoコマンドのオンラインマニュアルという意味になります。

[]
構文の説明でオプション要素を囲みます。括弧自体は入力しません。多くのコマンドは[*files*]という引数を持ちます。ファイル名が省略された場合は標準入力(通常はキーボードからの入力)になります。最後は EOF 文字を入力します。

^x
「制御文字」を示します。コントロールキーを押しながらxで指定された文字を押します。

|
構文の説明において選択可能な複数の項目を区切ります。

☞
参照すべき項目を示します。

ご意見とご質問

本書に関するご意見やご質問は以下の出版元までお寄せください。

株式会社オライリー・ジャパン
〒 160-0002　東京都新宿区坂町 26 番地 27　インテリジェントプラザビル 1F
電話　03-3356-5227
FAX　03-3356-5261
電子メール　japan@oreilly.co.jp

本書のWebサイトには正誤表、サンプル、追加情報が掲載されています。このサイトには以下のアドレスでアクセスできます。

http://www.oreilly.com/catalog/gdbpr/（英語）
http://www.oreilly.co.jp/books/487311246X/（日本語）

目　次

1章 概要

デバッガ(debugger)とは別のプログラムを実行するものですが、本書では実行されるプログラムのことを**デバッギー**(debuggee)と呼びます。一般に被デバッグプログラム、デバッグ対象プログラムとも呼ばれます。デバッガを使用すると、デバッギーの状態を確認、変更できます。また実行も制御できます。特にシングルステップ(ステップ実行)は有用です。シングルステップは1命令(アセンブリ言語レベル)ずつの実行や、1行(高級言語ソースコードレベル)ずつデバッギーの実行を進めるもので、デバッギーの動作を細かく観察できます。

デバッガには2つの種類があります。CPUの命令単位で操作する**アセンブリ言語レベル**のデバッガと、高級言語のソースコードのままで操作できる**ソースコードレベル**のデバッガです。ソースコードレベルのデバッガは非常に使いやすく、また必要に応じてアセンブリ言語レベルでも操作できる機能を備えているものがほとんどです。GDBはソースコードレベルのデバッガであり、恐らく現代のデバッガの中ではもっとも広く(対応しているCPUアーキテクチャの種類がもっとも多い)使用されているものです。

GDB単体では2種類の操作方法(インタフェース)を備えています。古典的なコマンドラインインタフェース(CLI、command-line interface)とテキストユーザインタフェース(TUI、text user interface)です。TUIは通常の端末や端末エミュレータ上で操作するもので、画面をいくつかの領域(ウィンドウ)に分割し、それぞれの領域でソースコード、レジスタ等を表示します。

GDBはC、C++、Objective C、Java †、Fortranで記述されたプログラムもデバッグできます。完全ではありませんが、GNU Modula-2コンパイラでコンパイルしたModula-2プログラムや、GNAT(GNU Adaトランスレータ)でコンパイルしたAdaプログラムにも対応しています。必要最小限という程度ですが、Pascalプログラムのデバッグも可能です。Chill言語の対応は打ち切られました。

C++とObjective Cの場合では、name demangling(デマングリング、名前の復元)にも対応

† GJC(GNU Compiler Collection(GCC)に含まれるGNU Javaコンパイラ)でネイティブコードにコンパイルする必要があります。

しています。C++とObjective Cではオーバロードしたプロシージャ名を、戻り値の型、引数の型、クラスメンバ情報を含んだ一意な名前へ変換します。これをname mangling(**名前のマングリング**)と呼びます。この変換により**タイプセーフなリンク**が保証されます。name manglingには色々な方式がありますが、GDBでは複数の方式に対応しており、ユーザが選択できるようになっています。通常は自動的にdemanglingした名前を使用します。

GNU Compiler Collection(GCC)で`-g3`と`-gdwarf-2`オプションを使用すると、Cプリプロセッサのマクロもgdbから参照可能になります。複雑な構造体や共用体を簡略化するためのマクロなどの場合に非常に効果的です。GDB単独でもプリプロセッサのマクロ展開には限定的に対応していますが、さらに強化が検討されています。

GDBではデバッグの際に複数のファイルを処理します。

- **実行ファイル**(exec file)はデバッグするプログラムそのものです。
- 必要に応じて**コアファイル**(core file)も指定可能です。コアファイルはプログラムが異常終了した際のメモリ空間をダンプしたものです。実行ファイルとコアファイルを使用すると、異常終了後にデバッグ作業が可能になります。コアファイルは通常のUnixシステムでは「core」というファイル名で生成されます[†]。必要に応じて、BSDシステムでは「プログラム名.core」というファイル名に変更できますし、GNU/Linuxシステムでも「core.PID」というファイル名に変更できます。PIDの部分はプロセス番号に置き換えられます。PID付きのファイル名は複数のコアファイルを保存する場合に便利です[††]。
- **シンボルファイル**(symbol file)はシンボル情報を収めた独立したファイルです。シンボル情報とは変数の名前、型、サイズ、実行ファイル内での位置などの情報です。

これらのファイルは通常はコンパイラが生成するものですが、必要に応じてGDBが書き込み、生成することも可能です。シンボルファイルはあまり馴染みがないかもしれません。一般的なデバッグ作業ではあまり使用しません。

プログラムの実行を途中で一時停止させるにはブレークポイント、ウォッチポイント、キャッチポイントなどの方法あります。

† 訳者注:コアファイルを生成しないように、shellを設定している場合があります。Bourne shell系の場合は`ulimit -c`を実行し、生成するコアファイルのサイズ上限を設定します。この値が0になっているとコアファイルは生成されません。C shell系では`limit coredumpsize`です。

†† 訳者注:GNU/Linuxシステムでは、カーネルのバージョンにもよりますが、`/proc/sys/kernel`下にある`core_uses_pid`、`core_pattern`などを操作することでコアファイル名を変更可能です。

- ソースコード内の特定の位置で実行を停止させる場合、その位置を**ブレークポイント**(breakpoint)と呼びます。

- メモリ内のあるアドレスの内容が変化した時にプログラムの実行を停止させる場合は、**ウォッチポイント**(watchpoint)を使用します。メモリ内のアドレスの指定には通常の変数名や、ポインタを使用した式(expression)を使用します。GDBがウォッチポイントを実現する方法は何通りかあります。ハードウェアがウォッチポイントに対応している場合は、GDBはこの機能を使用します†。ウォッチポイントのもっとも効率的な実現方法です。ハードウェアが対応していなければ、GDBは仮想メモリを使用しウォッチポイントを実現します。この場合でも効率はそれほど悪くなりません。仮想メモリも使用できない場合は、GDBはシングルステップ(プログラムを1命令ずつ実行する)を使用し、ウォッチポイントを実現します。

- 特定のイベント(event、事象)が発生した場合に実行を停止する方法を**キャッチポイント**(catchpoint)と呼びます。

GDBのドキュメントやコマンドでは上記3種類の実行停止方法をすべてまとめて**ブレークポイント**と呼んでいる場合が多いようです。特に**有効化**(enable)、**無効化**(disable)、**削除**(remove)する場合は3種類の停止方法ともに同じコマンドを使用します。

GDBはブレークポイント(含ウォッチポイント、キャッチポイント)に状態を持たせています。有効な状態ではブレークポイントに達した(または条件が合致した)場合にプログラムは実行を停止します。無効な状態ではGDBは内部にブレークポイントを保持はしていますが、プログラムは停止しません。削除すると、ブレークポイントはGDB内部から消去されます。ブレークポイントを一度だけ有効にする特殊な使用方法があります。この場合はブレークポイントに達した場合にプログラムは停止し、ブレークポイントは無効化されます(削除はされません)。

ブレークポイントには条件を加えることも可能です。プログラムの実行がブレークポイントに達した時、GDBは与えられた条件を評価し、結果が真の場合にプログラムを停止します。

また、ブレークポイントには**ignoreカウント**(無視カウント、ignore count)も設定できます。ignoreカウントは、プログラムがそのブレークポイントに達した時に停止せず、ブレー

† 訳者注：具体的にはデバッグレジスタを備えたCPUなどです。デバッグレジスタに設定したアドレス(命令、データどちらも可)へのアクセスがあると実行を停止するCPUの機能を利用します。設定可能なハードウェアウォッチポイントの数には上限があります。`watch`コマンドでウォッチポイントを表示した際にHardware watchpointと表示されないものはソフトウェアで実現されています。

クポイントを無視する回数です。ignoreカウントが0になるまでは、ブレークポイントに付加された条件は評価されません。

GDBを使用する際の基本的な概念の1つに**フレーム**(frame)があります。コンパイラ分野で**スタックフレーム**(stack frame)と呼んでいるものを、縮めて呼んでいます。スタックフレームとは関数レベルでの実行に必要な情報をまとめたものです。関数の引数や局所変数、それに戻り値の格納場所や関数の戻り番地などのリンケージ情報を含んでいます。GDBではフレームに番号を割り振っています。フレーム番号は0から始まり増加していきます。フレーム0がもっとも内側のフレームになります。つまり最後に呼び出された関数を表します。

GDBでは、bashと同じreadlineライブラリを使用しています。readlineライブラリにはヒストリ、コンプリーション(名前の補完)、コマンドライン編集機能があります。emacsスタイルとviスタイルの2種類の編集方式が使用可能です。

また、GDBはプログラミング言語と同等の機能も多く備えています。ユーザが独自に変数を定義できますし、一般的な演算も行えます。ユーザコマンドやフックも定義できます。フックはユーザコマンドの1種類で、GDBが内部コマンドを実行する直前または直後にユーザが独自の処理を加えるものです(詳細は「**10章 GDBコマンド(アルファベット順)**」を参照してください)。`while`ループや`if ... else ... end`を使った条件分岐も使用できます。

通常、GDBは被デバッグプログラムを実行しているマシン上で使用しますが、**クロスデバッグ**、つまりアーキテクチャが異なる別マシン上で被デバッグプログラムを実行し、リモートデバッグ(遠隔デバッグ、remote debug)することも可能です。リモートターゲットとはシリアルポートやネットワークを経由して接続しますが、リモートデバッグは一般的ではないため、本書では取り上げません。必要に応じてGDB付属ドキュメントを参照してください。

1.1 ソースコードのURL

GDBはGNU/LinuxシステムやBSDシステムでは標準のデバッガになっています。現代のUnixシステムではほぼ間違いなく動作しますし、多くの従来のシステムでも動作するでしょう(しかし、システムが古すぎるとGDBのバージョンも古いものにする必要があるかもしれません)。さらにGDBはコマンドラインインタフェースとテキストユーザインタフェースを備えていますし、GUIを提供する別ツールも存在します。GDBのGUIとして普及しているものにddd(Data Display Debugger)とInsightの2つがあります。両者とも内部のデバッガ本体にはGDBを使用しています。次の表にソースコードのURLをまとめておきます。さらに詳しい情報は「**3章 GDBのグラフィカルユーザインタフェース**」を参照してください。

デバッガ	URL
ddd	ftp://ftp.gnu.org/gnu/ddd/
GDB	ftp://ftp.gnu.org/gnu/gdb/
Insight	http://sources.redhat.com/insight/

2章 コマンドラインの構文

GDB は次のように起動します。

```
gdb ［オプション］ ［実行ファイル ［コアファイルまたはPID］］
gdb ［オプション］ --args 実行ファイル ［実行ファイルの引数 ...］
```

gdbtui コマンドは gdb --tui と起動した場合と同義です。--tui オプションは GDB のテキストユーザインタフェース（Text User Interface、TUI）を起動します。TUIは「**6章　GDBテキストユーザインタフェース**」で後述します。

GDBは古典的な短い形式のオプションとGNUスタイルの長い形式のオプションのどちらも使用可能です。長い形式のオプションは1つまたは2つのハイフンから始まるものです。コマンドラインに与えられるオプションを以下に挙げます。

--args
　コマンドライン上で実行ファイル名より後ろに置かれたものを、引数として実行ファイルへ渡す。

--async, --noasync
　コマンドラインインタフェースの非同期モードを有効／無効にする。

-b *baudrate*, --baud *baudrate*
　リモートデバッグ時に使用するシリアルポートのボーレートを指定する。

--batch
　オプションを処理し、終了する。

--cd *dir*
　カレントディレクトリを変更する。

-c *file*, --core *file*
　コアファイルを指定する。

-d *dir*, --directory *dir*
　ソースファイルを検索するディレクトリを指定する。

-e *file*, --exec *file*
　実行ファイルを指定する。

-f, --fullname
　Emacs-GDB インタフェース用の情報を出力する。

--help
　使用方法とオプションの説明を表示し終了する。

--interpreter *interp*
　インタプリタ／ユーザインタフェースを指定する。デフォルトはコマンドラインインタフェースになっており、フロントエンドプログラム用の別インタフェースもある。

-m, --mapped
　マップトシンボルファイルを使用する(システムが対応している場合のみ)。

-n, --nx
　.gdbinit ファイルを無視する。

-nw, --nowindows
　ウィンドウインタフェースが使用可能な場合でも、強制的にコマンドラインインタフェースを使用する。

-p *pidnum*, -c *pidnum*, --pid *pidnum*
　実行中のプロセスにアタッチする。

-q, --quiet, --silent
　起動時のバージョン番号表示を禁止する。

-r, --readnow
　最初のアクセス時にシンボルファイルをすべて読み取る。

-s *file*, --symbols *file*
　シンボルファイルを指定する。

--se *file*
　指定された1つのファイルを、シンボルファイルと実行ファイルとして使用する。

--statistics
　コマンド実行後にCPU時間とメモリの使用状況を表示する。

-t *device*, --tty *device*
　被デバッグプログラムの入出力デバイスを指定する。

--tui
　テキストユーザインタフェース(Text User Interface、TUI)を使用する。

-x *file*, --command *file*
　コマンドを指定されたファイルから読み取り、実行する。

--version
　バージョン情報を表示し、終了する。

-w, --windows
　強制的にウィンドウインタフェースを使用する(使用可能な場合)。

--write
　実行ファイルとコアファイルを書き込み可能にする。

3章 GDBのグラフィカルユーザインタフェース

ここではInsightとdddの特徴とGDBをGNU Emacsから利用する方法を補足します。ソースコードの編集にGNU Emacsを使用しているユーザは、同じエディタからGDBを使用した方が見ためや操作性の点から使いやすいかもしれません。

3.1 Insight

古くはGDBtkと呼ばれていたもので、機能的にGDBそのままのシンプルなGUIです。起動方法、コマンドラインに与えるパラメータはGDBへそのまま渡されます。

後述するddd同様にGDBのコマンドがメニュー化されていますが、デバッギーの標準入出力はInsightを起動した端末であり、この部分はGUI化されていません。

ソースコード内の変数名にマウスポインタを移動するだけでクリックせずに変数の現在値がポップアップ表示される点は便利でしょう。

3.2 ddd

コマンドラインに与えるパラメータは基本的にGDBと同じですが、dddがGUIを操作するためのオプションが多数追加されています。

dddを起動すると表示されるメインウィンドウでは、GDBの機能がメニュー化されており、良く使う機能はツールバーやメニューとは独立した小さなウィンドウにボタンとしてわかりやすく並べられています。

Insight同様に変数の値をポップアップ表示する機能や、さらにデータをグラフ構造として表示する機能もあり、複雑な構造体などを視覚化できます。また配列など大量の数値を表示する場合はGDBによるテキスト形式の表示は不向きですが、dddでは外部コマンド`gnuplot`を起動し数値をグラフ表示できます。

デバッグセッションを保存する機能もあり、dddを再起動する場合などに使用できます。

ddd内部で起動するデバッガをGDB以外のものにも変更可能で、Java、Perlなどのスクリ

プト言語でも ddd からデバッグ可能です。

3.3 GNU Emacs Debugger mode

内容的にキャラクタ端末での操作と同等ですので、厳密に言えばGUIよりもCUI、TUIという言葉の方がふさわしいでしょう。

通常はEmacs内部から`M-x gdb`と起動し、GDBのコマンドラインに与えるパラメータを入力します。

ソースコードの複数行表示ももちろんですが、ブレークポイントの設定、削除などよく使用する GDB の操作の多くが、Emacs で一般的なコントロールキーを組み合わせたキーボード操作から可能です。もちろんEmacsで一般的なカスタマイズが可能です。EmacsのShell-mode に似たバッファ内で GDB が起動されており、直接操作も可能です。

Insightやdddのように複数のウィンドウを必要としないデバッグセッションの場合には手軽に使え、ソースコードも普段見慣れたエディタで表示できる点などが良いでしょう。

内部で起動するデバッガをGDB以外のものにも変更可能で、Unixの古典的なデバッガsdbや dbx にも対応しています。また、ddd のように Perl スクリプトなどもデバッグ可能です。

4章
初期化ファイル

GDB 用と readline ライブラリ用それぞれに初期化ファイルが使用可能です。

4.1 .gdbinit ファイル

GDBは起動時に初期化ファイルを読み取ります。初期化ファイルにはコマンド、オプションなど、GDB を使用する際に毎回設定する内容を記述します。初期化ファイル名は BSD、Linux などの Unix システムでは .gdbinit です。Windows などの場合では gdb.ini というファイル名になります。初期化ファイルには空行やコメントも記述できます。行頭に # を書くと行末までコメントとなります。GDBは初期化ファイルから読み取ったコマンドと、コマンドラインに与えられたコマンドを次の順序で実行します。

1. $HOME/.gdbinit 内に記述されたコマンド。毎回初期化する「グローバル」な内容はここに記述します。
2. コマンドラインに与えられたオプション。
3. ./.gdbinit内に記述されたコマンド。プログラムのソースファイルと同じディレクトリに初期化ファイルを置くことにより、被デバッグプログラムごとに独自のオプション設定などが可能になります。
4. -x オプションで指定されたコマンドファイル。

-nx オプションを使用し、初期化ファイルを無視することもできます。

4.2 .inputrc ファイル

GDB ではヒストリやコマンドラインの編集機能に bash と同じ readline ライブラリを使用

しています。コマンドラインの編集方式にはviスタイルとemacsスタイルが使用できます。readlineライブラリは初期化ファイルとして~/.inputrcを読み取ります。詳細は本書の範疇を超えるため、bashやGDBに付属するドキュメント、またはinfoファイルを参照してください。ここでは.inputrcの一例を挙げておきます。

コマンド	説明
set editing-mode vi	編集方式にviスタイルを使用します。
set horizontal-scroll-mode On	カーソルの動きに合わせ左右にスクロールします。
control-h: backward-delete-char	^Hをバックスペースとして使用します。
set comment-begin #	bash同様に#をコメント行とします。
set expand-tilde On	~を展開します。
"\C-r": redraw-current-line	^Rをカレント行の再表示として使用します。

5章 GDBの式

GDBは特殊なプログラミング言語とも言えます。C言語の変数や演算子に似た構文や、デバッグ用の特殊な機能も備えています。ここではGDBを理解するため、さまざまな式(expression)を解説します。

5.1 値ヒストリ

printコマンドで値を表示するたびに、GDBは表示した値を内部で記憶しています。これを**値ヒストリ**(value history)と言います。ヒストリ内の番号を指定し、記憶した値を後から再度参照することが可能です。番号の前には$記号を付加します。また、GDBはヒストリへの注意を促す意味で$n = valという文字列も表示します。

```
$ gdb whizprog
...
(gdb) print stopped_early
$1 = 0
(gdb) print whiny_users
$2 = TRUE
(gdb)
```

ヒストリ番号を意識しないですむように、単なる$記号はヒストリ内の最後の値を表します。最後にprintした値がポインタ変数の場合は、次のような使用方法も可能です。

```
(gdb) print *$
```

このコマンドはポインタ変数が参照する内容を表示します。$記号を2つ続け$$とすると、ヒストリ内の最後から1つ前の値を表します。$$*n*とすると最後から*n*番目の値を表します($*n*が先頭(古い順)から*n*番目を表すのに対し、$$*n*は末尾(新しい順)から*n*番目を表します)。

show valuesコマンドを実行すると、ヒストリ内の値をすべて表示します。実行ファイルをリロードし、シンボルテーブルを再読み込みすると、値ヒストリは消去されます。ヒストリ

内の値がシンボルテーブルを参照している場合があり、シンボルテーブルのリロードにより不正になってしまったポインタの参照を防ぐためです。

5.2 変数とレジスタ

GDBではユーザが独自に変数を使用できます。変数名は$で始め英数字を続けます。ヒストリ番号と混同してしまうため、$直後の文字に数字は使用できません。例えば、配列の値を参照する際の添字に使用すると便利です。

```
(gdb) set $j = 0
(gdb) print data[$j++]
```

GDBでは何も入力せずにENTERキーを押すと直前に入力したコマンドを再度実行するため、上記の2つのコマンドを実行後に、単にENTERキーを押すだけで配列の内容を1つずつ確認できます。

GDBには定義済みの変数もあり、CPUのレジスタなどを参照できるようになっています。どのアーキテクチャでも使用できるレジスタ名が4つ定義されており、マシンアーキテクチャの差異を吸収できます。

既定義の変数(コンビニエンス変数、便利変数)を一覧にまとめておきます。末尾の4つの変数はどのアーキテクチャでも使用できるレジスタ名です。

変数	説明
$	値ヒストリ内の末尾(最新)の値
$*n*	値ヒストリ内の先頭(最古)から *n* 番目の値
$$	値ヒストリ内の末尾から1つ前の値
$$*n*	値ヒストリ内の末尾から *n* 番目の値
$_	最後にxコマンドで表示したアドレス
$__	最後にxコマンドで表示したアドレスの内容
$_exitcode	デバッギーが終了した際の終了コード
$bpnum	最後に設定されたブレークポイント番号
$cdir	オブジェクトファイル内に格納されている、カレントソースファイルのパス(格納されていない場合もある)
$cwd	カレントディレクトリ
$fp	フレームポインタレジスタ
$pc	プログラムカウンタレジスタ
$ps	プロセッサステータスレジスタ
$sp	スタックポインタレジスタ

5.3 特殊な式

GDBでは被デバッグプログラムを記述した言語の文法(型、演算子、演算子の優先順位)が使用できます。$i++のようにGDBの変数を操作する場合にも使用できます。また、被デバッグプログラムの言語だけでは表現できない内容でも、GDBには特殊な構文が用意されており、使用可能になっています。

配列定数

デバッギーのメモリ内に配列定数を作成できます。配列は要素を括弧で囲んで記述します。

```
{ 1, 2, 3, 42, 57 }
```

配列演算

配列演算子(@記号)は与えられた添字にしたがい、配列内の複数の要素を参照します。例えば、被デバッグプログラムがmalloc()を使用し、メモリを割り当てたとします。

```
double *vals = malloc(count * sizeof(double));
```

配列内の要素を表示するには、通常は添字を与える必要があります。

```
(gdb) print vals[3]
$1 = 9
```

配列内の 0 番目から 2 番目の要素を参照する場合に、@ 記号が使用できます。

```
(gdb) print *vals@3
$2 = {0, 1, 4}
```

ファイル名のスコープ

複数のソースファイルで同じ変数名を使用している場合(例えばそれぞれが静的変数になっている場合)、ファイル名::変数名と指定することで、解決できます。

```
(gdb) print 'main.c'::errcount
$2 = 0
```

C++ の::演算子と混同しないために、main.c をシングルクォートで囲む必要があります。

6章 テキストユーザインタフェース

GDBはデフォルトでは、コマンド入力による古典的なコマンド行による操作方法を踏襲しています。シングルステップ時にはソースファイルを1行ずつ表示します。GUIを備えたデバッガではより多くの行数を表示できますし、多数のプログラマが、この理由のためだけに、GUIデバッガを好んで使用しています。しかし、最近のバージョンのGDBでは、テキストユーザインタフェース(text user interface、TUI)を備えています。TUIでは成熟したcursesライブラリを使用しており、複数の「ウィンドウ」を通常の端末やxtermなどの端末エミュレータ上に開けます。この機能は非常に効果が高く、特にすべての操作をキーボードから行えるのでデバッグ作業がスムーズに進められます。

TUI用に多数のsetオプションやコマンドが用意されています。その他のオプション、コマンドと一緒に「8章　set、showコマンド」と、「10章　GDBコマンド(アルファベット順)」にまとめてあります。

残念ながらGDB 6.3ではTUIは完成されておらず、動作させるのに必要な情報も入手できませんでした。そのため本書でも具体的に解説できません。しかし、今後もユーザ(もちろん本書の読者も含みます)と開発者により、改良され続けることは間違いないでしょう。

7章 GDBコマンド（グループ別）

ここではコマンドを内容別にまとめます。GDBの開発者しか使用しないメンテナンス用のコマンドや、シリアルポートやネットワーク経由でリモート操作するクロスデバッグ用のコマンドは割愛してあります。

7.1 コマンドのエイリアス

エイリアス	正式名
bt	backtrace
c	continue
cont	continue
d	delete
dir	directory
dis	disable
do	down
e	edit
f	frame
fo	forward-search
gcore	generate-core-file
h	help

エイリアス	正式名
i	info
l	list
n	next
ni	nexti
p	print
po	print-object
r	run
s	step
share	sharedlibrary
si	stepi
u	until
where	backtrace

7.2 ブレークポイント

コマンド	説明
awatch	式にウォッチポイントを設定する（参照、変更共）
break	行や関数名にブレークポイントを設定する
catch	イベントにキャッチポイントを設定する
clear	指定したブレークポイントを削除する

コマンド	説明
commands	ブレークポイントに達した場合に実行するコマンドを設定する
condition	指定したブレークポイントに条件を設定する
delete	1つまたは複数のブレークポイントや、自動表示(auto-display)設定を削除する
disable	1つまたは複数のブレークポイントを無効にする
enable	1つまたは複数のブレークポイントを有効にする
hbreak	ハードウェア機能を利用したブレークポイントを設定する
ignore	指定したブレークポイントに ignore カウントを設定する
rbreak	正規表現にマッチする関数名すべてにブレークポイントを設定する
rwatch	式にウォッチポイントを設定する(参照のみ)
tbreak	一時ブレークポイントを設定する
tcatch	一時キャッチポイントを設定する
thbreak	一時ハードウェアブレークポイントを設定する
watch	式にウォッチポイントを設定する(変更のみ)

7.3 データの操作

コマンド	説明
call	プログラム内の関数を呼び出す
delete display	プログラムの実行停止時に自動的に表示するように設定した、1つまたは複数の式を削除する
delete mem	メモリリージョンを削除する
disable display	プログラムの実行停止時に自動的に表示するように設定した、1つまたは複数の式を無効にする
disable mem	メモリリージョンを無効にする
disassemble	メモリ内容を逆アセンブルする
display	プログラムの実行停止時に式の値を自動的に表示する
enable display	プログラムの実行停止時に自動的に表示するように設定した、1つまたは複数の式を有効にする
enable mem	メモリリージョンを有効にする
inspect	print と同義
mem	メモリリージョンの属性を定義する
output	値ヒストリに記憶しない点と改行しない点を除き、printと同義。スクリプト用
print	式の値を表示する
print-object	Objective C のオブジェクトに自身の情報を表示させる

コマンド	説明
printf	printf の構文で値を表示する
ptype	指定された型の定義を表示する
set	式を評価し、結果をプログラムの変数に代入する
set variable	set と同義。GDB の変数と混同しないように。
undisplay	プログラムの実行停止時に自動的に表示するように設定した、1つまたは複数の式をキャンセルする
whatis	式の型を表示する
x	x/fmt アドレス形式でメモリ内容を表示する。後述する「**10章 GDBコマンド(アルファベット順)**」の x を参照。

7.4 ファイルの操作と表示

コマンド	説明
add-symbol-file	動的にファイルをロードし、GDBのシンボルテーブルにシンボルを追加する
add-symbol-file-from-memory	GDBのメモリ中にあるロード済みのオブジェクトファイルからシンボルをロードする
cd	GDB とデバッギーのカレントディレクトリを変更する
core-file	メモリとレジスタの内容を含むコアファイルを指定する
directory	ソースファイルサーチパスの先頭にディレクトリを挿入する
edit	ファイル、関数を編集する
exec-file	実行ファイルを指定する
file	デバッグするプログラムのファイル名を指定する
forward-search	表示された最終行から、カレントソースファイルを前方に正規表現で検索する
generate-core-file	デバッギーの現在のメモリ内容をダンプし、コアファイルを生成する
list	関数または行を表示する
nosharedlibrary	共有ライブラリのシンボルをすべてアンロードする
path	1つまたは複数のディレクトリをオブジェクトファイルサーチパスに追加する
pwd	カレントディレクトリを表示する
reverse-search	表示された最終行から、カレントソースファイルを後方に正規表現で検索する
search	forward-search と同義
section	実行ファイル内の指定したセクションのベースアドレスを変更する
sharedlibrary	正規表現にマッチしたファイル名の共有ライブラリのシンボルをロードする
symbol-file	指定した実行ファイルのシンボルテーブルをロードする

7.5 プログラムの実行

コマンド	説明
advance	指定された位置までプログラムを実行する
attach	実行中のプロセスやファイルにアタッチする
continue	デバッグ中のプログラムの実行を再開する
detach	アタッチしているプロセスやファイルをデタッチする
finish	指定したフレームを完了するまで実行する
handle	シグナル処理を指定する
interrupt	プログラムの実行に割り込む
jump	指定した行、アドレスから実行を再開する
kill	デバッグ中のプログラムを終了する
next	プログラムの次の行（ソースコード単位）を実行する
nexti	プログラムの次の命令（アセンブリ単位）を実行する
run	デバッグするプログラムを開始する
signal	指定したシグナルを送信し、プログラムの実行を再開する
start	main 関数の始まりまでプログラムを実行する。main 関数前にコンストラクタを実行する C++ プログラムの場合に有用。
step	ソースコードの行が変わるまでプログラムを実行する。呼び出した関数内に入る。
stepi	1 命令を実行する
thread	スレッドを切替える
thread apply	複数のスレッドにコマンドを適用する
thread apply all	すべてのスレッドにコマンドを適用する
tty	今後実行するデバッギーに tty（端末）を設定する
unset environment	デバッギーの環境から環境変数を削除する
until	ソースコードのカレント行を終えるまでプログラムを実行する

7.6 スタックの操作

コマンド	説明
backtrace	すべてのフレームのバックトレースを表示する
down	カレントフレームが呼び出しているフレームを選択し表示する
frame	フレームを選択し表示する
return	選択したフレームを破棄する
select-frame	フレームを選択するが表示はしない
up	カレントフレームを呼び出しているフレームを選択し表示する

7.7 状態の確認

コマンド	説明
info	デバッギーに関する情報を表示する汎用のコマンド
macro	コマンドの対象をCプリプロセッサのマクロにする場合に、コマンドの前に挿入するプレフィックス
show	デバッガに関する情報を表示する汎用のコマンド

7.8 その他機能

コマンド	説明
apropos	コマンド名を正規表現で検索する
complete	コマンド名の補完候補を表示する
define	新たにコマンドを定義する
document	ユーザ定義コマンドに説明を加える
dont-repeat	ユーザ定義コマンドでのみ使用する
down-silently	表示しない点を除き down と同義
echo	文字列定数を表示する
else	if とともに使用し、コマンドを実行する
end	コマンド列(または動作)を終了する
help	コマンドの一覧を表示する
if	与えられた条件が非 0 の場合にコマンドを実行する
make	引数を与え make コマンドを実行する
quit	GDB を終了する
shell	シェルを起動しコマンドを実行する
source	指定したファイルからコマンドを実行する
up-silently	表示しない点を除き up と同義
while	与えられた条件が非 0 の間はコマンドを実行する

7.9 テキストユーザインタフェースコマンド

コマンド	説明
focus	キーボードフォーカスを与えるウィンドウを変更する
layout	使用中のウィンドウのレイアウトを変更する
refresh	画面を一度消去し再表示する
tui reg	レジスタウィンドウに表示するレジスタを変更する

コマンド	説明
update	ソースコードウィンドウを更新する
winheight	指定したウィンドウの高さを変更する

7.10 よく使用するコマンド

GDBは戸惑うほど多くのコマンドを備えていますが、ほとんどのユーザは一部のコマンドだけでデバッグ作業をなんなくこなしています。**表1**にもっともよく使用されるコマンドをまとめておきます。

表1 GDB コマンドトップ 12(1 ダース)

コマンド名	機能	使用例
backtrace	コールトレースを表示する	ba
break	関数または行番号にブレークポイントを設定する	b main、b parser.c:723
continue	実行を再開する	cont
delete	ブレークポイントを削除する	d3
finish	関数を終了するまで実行する	fin
info breakpoints	ブレークポイントの一覧を表示する	i br
next	次の1行まで実行する。関数呼び出しも1行と扱う	ne
print	式を表示する	print 1.0/3.0
run	プログラムを(再)実行する。引数も渡せる。	ru、ru -u -o foo < data
step	次の1行まで実行する。関数呼び出しの場合は関数内に入る	s
x	メモリ内容を表示する	x/s *environ
until	指定行まで実行する	until、until 2367

8章 set、showコマンド

setコマンドはGDBの動作を操作するもので、膨大な種類のパラメータがあります。しかしよく使用するパラメータはそれほど多くありません。showコマンドはsetコマンドのパラメータの内容を表示します。このセクションではパラメータを一覧にまとめ、GDBに与える影響を解説します。ほとんどのパラメータではset optionとset option onは同義です。双方ともにパラメータを有効化します。無効化するにはset option offを使用します。

annotate

```
set annotate level
show annotate
```

annotation_level変数に*level*を設定する。
内部でGDBを使用するGUIではこの変数を使用している。

architecture

```
set architecture architecture
show architecture
```

ターゲットのアーキテクチャに*architecture*を設定する。クロスデバッグ以外では使用しない。

args

```
set args
show args
```

デバッギーの実行開始時に引数を渡す。runコマンドに引数を与えない場合に使用される。

「10章 GDBコマンド(アルファベット順)」を参照。

auto-solib-add

```
set auto-solib-add
show auto-solib-add
```

必要に応じて共有ライブラリからシンボルを自動的にロードする。offを設定した場合は、sharedlibraryコマンドを使用し、手動でロードする必要がある。

auto-solib-limit

```
set auto-solib-limit megs
show auto-solib-limit
```

共有ライブラリから自動的にロードするシンボルのサイズを、*megs*メガバイトに制限する。使用できないシステムも存在する。

backtrace

```
set backtrace limit count
show backtrace limit
set backtrace past-main
show backtrace past-main
```

1番目の構文では表示するフレーム数を*count*に制限する。デフォルトでは無制限。2番目の構文ではmain関数以前のフレームをGDBが表示するか否かを変更する。通常スタートアップルーチンはデバッグ対象ではないため、デフォルト値はoff。

breakpoint

```
set breakpoint pending val
show breakpoint pending
```

ブレークポイントの位置を特定出来ない場合の動作を変更する(例えば共有ライブラリをロードする前など)。
*val*にはon、off、autoを指定する。onの場合は、GDBは自動的に遅延ブレークポイントを作成する。autoはユーザに問い合わせる。offは遅延ブレークポイントは作成されない。

can-use-hw-watchpoints

```
set can-use-hw-watchpoints value
show can-use-hw-watchpoints
```

非0の場合は、GDBはハードウェアの機能を利用したウォッチポイントを使用する(ハードウェアが対応している場合)。それ以外の場合はハードウェアの機能を利用しない。

case-sensitive

```
set case-sensitive
show case-sensitive
```

GDBがシンボルを検索する場合に大文字／小文字を区別するか否かを設定する。on、off、autoが設定可能。autoの場合はプログラムを記述した言語の仕様にしたがう。

coerce-float-to-double

```
set coerce-float-to-double
show coerce-float-to-double
```

値がonの場合は、プロトタイプ宣言されていない関数を呼び出す場合に、GDBはfloat型をdouble型に強制的に変換する。offの場合は、float型を変換せず、プロトタイプ宣言された関数はfloat型のまま値を受け取る。

commands

```
show commands [cmdnum]
show commands +
```

デフォルトではコマンドヒストリは最後のコマンドを10個表示する。*cmdnum*を指定すると、*cmdnum*番目を中心とした10個のコマンドを表示する。2番目の構文では直前に表示したコマンドに続く10個のコマンドを表示する。

complaints

```
set complaints limit
show complaints
```

シンボルテーブルロード時に問題があっても、通常は何も表示しないが、complaints変数を設定すると、発見された問題の種類ごとに最大で*limit*回数まで問題を表示する。デフォルト値は0になっており、何も表示しない。「無制限」を指定する場合は巨大な数値を指定する。

confirm

```
set confirm
show confirm
```

ブレークポイントの削除など特定の操作の前には通常はユーザに確認をするが、confirm変数にoffを設定すると確認しない。offにする影響を充分把握している場合にのみ使用する。

convenience

```
show convenience
```

今までに使用されたGDBの変数(コンビニエンス変数、便利変数)の一覧を値とともに表示する。show convと短縮可能。

copying

```
show copying
```

GNU General Public License(GPL)を表示する。

cp-abi

```
set cp-abi
show cp-abi
```

C++オブジェクトのアプリケーションバイナリインタフェース(Application Binary Interface、ABI)を設定する。デフォルト値はautoで、GDBがABIを決定する。バージョン3未満のg++

の場合には値にgnu-v2を、g++バージョン3.0以降の場合にはgnu-v3をそれぞれ設定する。HP ANSI C++ コンパイラの場合には hpaCC を設定する。

debug-file-directory

```
set debug-file-directory dir
show debug-file-directory
```

デバッグ情報のファイルを検索するディレクトリを指定する。実行ファイル内にデバッグ情報を持たないシステム上でデバッグする際に使用する。

demangle-style

```
set demangle-style style
show demangle-style
```

コンパイラが mangle（マングル）した名前を元の Objective C、C++ の名前に復元する方式を指定する。指定できる値は次のとおり。

arm	『The Annotated C++ Reference Manual』† に記述されているアルゴリズムを使用する。
auto	GDB 付属文書では、auto 指定だけでは cfront †† により生成されたコードをデバッグできないと記述されている。 GDB が demangle-style を判断する。
gnu	GNU C++ コンパイラ（g++）と同じ方式を使用する。デフォルト値。
hp	HP 社の ANSI C++ コンパイラ（aCC）方式を使用する。
lucid	Lucid 社の C++ コンパイラ（lcc）方式を使用する。

directories

```
show directories
```

ソースファイルを検索するディレクトリのサーチパスを表示する。

† 訳者注：『注解 C++ リファレンス・マニュアル』（足立高徳、小山裕司訳、シイエム・シイ出版部）

†† 実際に問題になることはないでしょう。現在では cfront ベースの C++ コンパイラは一般的とは言えません。

disassembly-flavor

```
set disassembly-flavor flavor
show disassembly-flavor
```

機械語の命令を表示する際に使用する命令セット。本書執筆時点ではIntel x86アーキテクチャのみ使用可能。指定可能な値(*flavor*)はintelまたはattで、デフォルト値はatt。

editing

```
set editing
show editing
```

入力されたコマンドラインの編集機能を有効にする。

environment

```
set environment variable[=value]
show environment [variable]
```

環境変数*variable*に新たな値(*value*)や空文字列を設定する。*variable*を指定しない場合はすべての環境変数を表示する。指定した場合は*variable*の値を表示する。

exec-done-display

```
set exec-done-display
show exec-done-display
```

コマンドの非同期実行時に完了通知を有効にする。

extension-language

```
set extension-language .ext lang
show extension-language
```

拡張子*.ext*のファイル名とプログラミング言語*lang*を関連付ける。

follow-fork-mode

set follow-fork-mode *mode*
show follow-fork-mode

デバッギーが子プロセスを生成した場合に、以降GDBが親子どちらのプロセスをデバッグ対象にするかを設定する。親プロセスをデバッグする場合は*mode*にparentを、子プロセスの場合は child をそれぞれ指定する。

gnutarget

set gnutarget *format*
show gnutarget

デバッギー(コアファイル、実行ファイル、オブジェクトファイル(.o))のファイルフォーマット。デフォルト値は auto で、通常はこのままにしておくのが良い。

height

set height *count*
show height

画面(1ページ)の行数を設定する。0を設定すると、1画面分以上の量の情報を表示する場合に GDB が一時停止しない。

history

set history *feature*
show history *feature*

GDB のコマンドヒストリの動作を制御する。設定する値と内容は次のとおり。

set history expansion
show history expansion
　csh スタイルの！文字を使用する。デフォルトでは off。

set history filename *file*
show history filename
　コマンドヒストリをファイルに保存し、GDB起動時にはこのファイルからロードし、復元する。デフォルトのファイル名は./.gdb_history。環境変数GDBHISTFILEによりデフォルトのファイル名は変更可能。

set history save
show history save
　コマンドヒストリのファイルへの保存 / 復元を有効にする。

set history size *amount*
show history size
　コマンドヒストリに保存するコマンド数を *amount* に制限する。

input-radix

set input-radix *base*
show input-radix

数値を入力する際に基底のデフォルト値を設定する。設定可能な値は8、10、16。入力する数値は設定された底の値にしたがって解釈されるが、8進数の前に'0'を付加する、16進数の前に'0x'や'0X'を付加するなどの明示的な指定も可能。

language

set language *lang*
show language

ソースファイルのプログラミング言語を *lang* に設定する。通常はGDBが実行ファイル内の情報から言語を判断する。

listsize

set listsize *count*
show listsize

listコマンドで表示するソース行数。

logging

```
set logging
set logging option value
show logging
```

通常はon、offを設定し、コマンド出力のログを有効化/無効化する。*option*と*value*を指定すると、特殊なログ出力が設定可能。

ログオプション

file	コマンド出力ログをファイルに書き出す。デフォルトファイル名はgdb.txt。
overwrite	ログファイルを上書きする。overwriteを設定しない場合は追加書きする。
redirect	コマンド出力ログをファイルにのみ書き出す。デフォルトでは端末とファイルの両方に書き出す。

max-user-call-depth

```
set max-user-call-depth limit
show max-user-call-depth
```

ユーザ定義コマンドの再帰呼び出しの上限値を*limit*に設定する。上限に達するとGDBはコマンドが無限に再帰していると判断し、エラー終了する。

opaque-type-resolution

```
set opaque-type-resolution
show opaque-type-resolution
```

シンボルロード時に構造体/クラス/共用体のオペーク(opaque)なシンボルを解決する。ある型(struct foo *)の定義と参照が別ファイルで行われている場合に、型を定義しているファイルを検索する。

osabi

```
set osabi os-abi-type
show osabi
```

デバッギーのオペレーティングシステムアプリケーションバイナリインタフェース。デフォルト値はautoで、GDBが自動的に判断する。GDBの自動判断を変更する必要がある場合に使用する。

output-radix

```
set output-radix base
show output-radix
```

数値を表示する際のデフォルトの底を設定する。設定できる値は8、10、16。入力する数値は設定された底の値にしたがって解釈されるが、8進数の前に'0'を付加する、16進数の前に'0x'や'0X'を付加するなどの明示的な指定も可能。

overload-resolution

```
set overload-resolution
show overload-resolution
```

オーバロードされた関数をGDBから呼び出す際に、関数のシグネチャが引数の型にマッチするものを検索する。

pagination

```
set pagination
show pagination
```

表示のページ制御を有効化／無効化する。デフォルトではon。

paths

show paths

実行ファイルの検索パスを表示する(環境変数PATH)。この検索パスはオブジェクトファイルの検索にも使用される。

print

set print *print-opt*
show print *print-opt*

GDBではデバッギーのさまざまな情報を表示でき、また表示も細かく制御可能になっている。表示を制御するオプションのほとんどは set print オプション名、set print オプション名 on のどちらでも有効化できる。onの代わりにoffを指定することで無効化できるオプションもある。オプションが有効/無効のどちらになっているかを確認するには、show print オプション名を実行する。*print-opt* の値と GDB の動作の変化を次の表にまとめる。

set print address, show print address
: フレーム内のプログラムカウンタ値を含める。

set print array, show print array
: 配列の表示形式を変更し、読みやすくする。しかし表示により多くの行数、桁数が必要になる。デフォルトでは off。

set print asm-demangle, show print asm-demangle
: 逆アセンブル表示時でもC++、Objective Cの名前をdemangle(デマングル、復元)する。

set print demangle, show print demangle
: 表示時に C++、Objective C の名前を demangle (デマングル、復元)する。

set print elements *count*, show print elements
: 配列表示時に *count* 以上の要素を表示しない。デフォルト値は 200。0 を指定すると無制限に表示する。

set print null-stop, show print null-stop
値が0(文字配列の場合のASCII NUL)の要素を見つけたら、配列の表示を停止する。デフォルトではoff。

set print object, show print object
ポインタ変数の表示時にポイントしているオブジェクトの実際の型、すなわち宣言された型ではなく仮想関数テーブルの情報から導き出した型を表示する。
デフォルト値はoffで、宣言された型を表示する。

set print pascal_static-members, show print pascal_static-members
Pascalのスタティックメンバを表示する。

set print pretty, show print pretty
構造体の表示形式を変更し、メンバを1行ずつインデント付きで表示する。

set print sevenbit-strings, show print sevenbit-strings
文字列中の8ビット文字を\nnn形式で表示する。

set print static-members, show print static-members
C++オブジェクト表示時にスタティックメンバを表示する。

set print symbol-filename, show print symbol-filename
アドレスをシンボルへ変換し表示する際に、ソースファイル名と行番号も表示する。

set print union, show print union
構造体内の共用体を表示する。

set print vtbl, show print vtbl
C++の仮想関数テーブルを見やすく表示する。デフォルト値はoff。

set print max-symbolic-offset *max*, show print max-symbolic-offset
アドレス表示時に、オフセットが*max*未満の場合にのみシンボル+オフセット形式で表示する。デフォルト値は0で無制限をあらわす。

prompt

```
set prompt string
show prompt
```

プロンプト文字列を設定／表示する。デフォルトは(gdb)。

radix

```
set radix base
show radix
```

数値の入力、表示の際に使用する底を両方とも設定する。設定可能な値は8、10、16。入力する数値は設定された底の値にしたがって解釈されるが、8進数の前に'0'を付加する、16進数の前に '0x' や '0X' を付加するなどの明示的な指定も可能。☞input-radix、output-radix

scheduler-locking

```
set scheduler-locking
show scheduler-locking
```

デバッギー内のスレッドのうち、デバッグ対象外のスレッドのスケジューリングを制御する(オペレーティングシステムによっては使用できない)。設定できる値は、on、off、step。offの場合はすべてのスレッドが実行され、他のスレッドがデバッガに割り込める(ブレークポイントに達した、シグナルが送信されたなど)。onの場合は、カレントスレッドだけが実行される。step の場合は、シングルステップ時のみスケジューラがロックされる。

solib-absolute-prefix

```
set solib-absolute-prefix path
show solib-absolute-prefix
```

共有ライブラリの絶対パスの前に *path* を挿入する。
主にクロスデバッグ時に使用し、ターゲットの共有ライブラリの検索時に使用する。

solib-search-path

```
set solib-search-path path
show solib-search-path
```

共有ライブラリの検索パスをコロンで区切って設定する。solib-absolute-prefixを加えた検索でも発見できなかった場合に、このパスを検索する。主にクロスデバッグ時に使用する。

step-mode

```
set step-mode
show step-mode
```

stepコマンドのモードを設定する。デフォルトではstepコマンドはデバッグ情報が存在しない関数内には入らないが、この変数を設定すると関数内に入り、機械語(命令)レベルでのデバッグが可能になる。

stop-on-solib-events

```
set stop-on-solib-events
show stop-on-solib-events
```

共有ライブラリイベント発生時に停止する。一般的なイベントは共有ライブラリのロード／アンロード。

symbol-reloading

```
set symbol-reloading
show symbol-reloading
```

VxWorksなど自動再リンク機能を備えたシステムでは、オブジェクトファイルが変更された場合に、シンボルテーブルを再ロードする。

trust-readonly-sections

set trust-readonly-sections
show trust-readonly-sections

リードオンリーなセクションはリードオンリーなまま変化しないと想定する。この設定によりGDBはリモートデバッギーからではなく、オブジェクトファイルから内容を取り出せる。リモートデバッグ以外では基本的に使用しない。

tui

set tui *feature value*
show tui *feature*

TUI 機能の *feature* に *value* を設定する。

TUI 機能

set tui active-border-mode *mode*
show tui active-border-mode

アクティブウィンドウのボーダ(枠線)に使用するcursesライブラリの属性を設定／参照する。設定可能な値は、normal、standout、half、half-standout、bold、bold-standout。

set tui border-kind *kind*
show tui border-kind

ボーダの描画に使用する文字を設定／参照する。設定可能な値は次のとおり。

acs	代替文字セット(線描画用文字セット、Alternate Character Set)を使用する(端末が対応している場合)。
ascii	通常の +、-、| を使用する。
space	空白文字を使用する。

set tui border-mode *mode*
show tui border-mode

非アクティブウィンドウのボーダ(枠線)に使用するcursesライブラリの属性を設定／参照する。設定可能な値は、normal、standout、half、half-standout、bold、bold-standout。

values

```
show values [valnum]
show values +
```

引数がない場合は、値ヒストリ（value history）内の最後の10個を表示する（値ヒストリの詳細については「5.1 **値ヒストリ**」を参照）。引数*valnum*はヒストリ番号として解釈され、*valnum*を中心とした連続する10個の値を表示する。+を指定すると、最後に表示した値の次の10個を表示する。

variable

```
set variable assignment
```

GDBの変数ではなく、プログラムの変数へ代入する。

verbose

```
set verbose
show verbose
```

時間のかかるコマンド実行中に情報を表示する。この表示によりGDBが異常な状態になっていないことが確認できる。

version

```
show version
```

GDBのバージョンを表示する。

warranty

```
show warranty
```

GNU General Public License（GPL）の「無保証（no warranty）」の記述を表示する。

watchdog

```
set watchdog seconds
show watchdog
```

リモートターゲットの実行完了を*seconds*秒以上待たない。タイムアウトになった場合は、エラーを表示する。

width

```
set width numchars
show width
```

1 行内の文字数を設定する。0 を設定すると行を折り返さない。

write

```
set write
show write
```

GDB が実行ファイル、コアファイルへ書き込めるようにする。デフォルトでは off。

9章 info コマンド

infoコマンドはデバッギーに関するさまざまな情報を表示します(GDB内部の機能、変数、オプションなどの情報を表示するのはshowコマンドです)。引数がない場合は、参照可能な情報の一覧を表示します。

引数	表示する情報
address *sym*	シンボル*sym*の格納場所。アドレスかまたはレジスタ名のどちらかになる。
all-registers	浮動小数点レジスタも含めたすべてのレジスタ
args	現在の関数(フレーム)の引数
break [*bpnum*]	ブレークポイント*bpnum*の情報。*bpnum*を指定しない場合は全ブレークポイントの情報
breakpoints [*bpnum*]	info breakコマンドと同義
catch	カレントフレーム内で発生した例外
classes [*regexp*]	指定された正規表現にマッチするObjective-Cのクラスの情報。正規表現が指定されない場合は全クラスの情報。
display	自動表示の設定内容
extensions	プログラミング言語とファイル名の拡張子の関連付け
f [*address*]	info frameコマンドと同義
files	実行ファイル、コアファイル、シンボルファイルなど、デバッグ対象
float	浮動小数点を処理するハードウェアの情報
frame [*address*]	引数がない場合はカレントフレーム情報。引数がある場合は指定されたアドレスを含むフレームの情報のみ。カレントフレームは含まない。
functions [*regexp*]	引数がない場合は全関数の名前と型。引数がある場合は正規表現にマッチする関数のみ。

引数	表示する情報
handle	GDB が処理するシグナル一覧
line *line-spec*	*line-spec* で指定されたソース行を含むアドレス範囲。*line-spec* については「**10章 GDB コマンド(アルファベット順)**」の list を参照。このコマンドにより指定されたソース行の開始アドレスが設定され、x/i コマンドで命令を確認できる。
locals	カレントフレームからアクセス可能な局所変数(スタティック、またはオート変数)
macro *macroname*	マクロ *macroname* の定義位置
mem	メモリリージョンとその属性
proc [*item*]	実行中のデバッギー。*/proc*ファイルシステムが利用可能なシステムでのみ使用可能。引数 *item* には次のものが指定可能。 mappings(利用可能なメモリ範囲とアクセス可否)、times(開始時刻とuser、systemのCPU時間)、id(プロセスID)、status(プロセスの一般的な情報)、all(すべての情報)
program	実行中、停止中、プロセス ID などデバッギーに関する情報
registers [*reg* …]	引数がない場合は、浮動小数点レジスタ以外の全レジスタ。または引数で指定されたレジスタ。
s	info stack コマンドと同義(backtrace コマンドと同義)
scope *address*	指定されたアドレスを含むスコープ内のローカルな変数。アドレスは関数名、ソース行、* から始まる絶対アドレスで指定する。
selectors [*regexp*]	指定された正規表現にマッチする Objective-C のセレクタ。指定されない場合は全セレクタ。
set	引数がない show コマンドと同義
share	info sharedlibrary コマンドと同義
sharedlibrary	ロード済みの共有ライブラリ
signal	info handle コマンドと同義
source	コンパイルしたディレクトリ、プログラミング言語、デバッグ情報などソースファイルの情報
sources	デバッグ情報を持つ全ソースファイルの情報。出力はロード済みのものと今後必要に応じてロードされるものの、2つのリストになる。
stack	backtrace コマンドと同義
symbol *address*	*address* に格納されているシンボル(関数、変数など)の名前

引数	表示する情報
target	info filesコマンドと同義
terminal	現在の端末モード
threads	全スレッド
types [*regexp*]	指定された正規表現にマッチする型の情報。正規表現が指定されなければ、すべての型の情報。
variables [*regexp*]	正規表現がない場合は、局所変数以外のすべての変数の名前と型。正規表現が指定された場合は、名前がマッチする変数。
watchpoints [*wpnum*]	ウォッチポイント*wpnum*の情報。*wpnum*が指定されない場合は、全ウォッチポイントの情報。
win	表示しているすべてのTUIウィンドウの名前とサイズ

10章 GDBコマンド（アルファベット順）

このアルファベット順にまとめたGDBコマンドの一覧には、デバッグ作業に必要なものをすべて含めました。GDBのメンテナが使用するコマンドや、シリアルポートやネットワーク経由でリモートシステムをクロスデバッグする際に使用するコマンドなどは、一般ユーザはほとんど使用しないため、割愛してあります。

ほとんどのコマンドは短縮形で使用されます。短縮形の一覧は「7.1　コマンドのエイリアス」で前述しました。

add-symbol-file

```
add-symbol-file file addr [-mapped] [-readnow]
add-symbol-file file [-s section address ...]
```

*file*から追加のシンボルテーブル情報を読み取る。デバッギーが自身で動的にロードした場合に使用する。デバッギーがみずからロードした場合は、GDBがアドレスを自動的に決定できないため、アドレスを明示的に指定し、シンボルテーブルをロードする必要がある。-mappedと-readnowオプションはfileコマンドと同様の意味を持つ(☞file コマンド)。-sオプションを使用し、*address*から始まるセクションに名前を付けることも可能。-sオプションを複数使用し、複数の *section/address* ペアを指定可能。

advance

```
advance bp-spec
```

プログラムが *bp-spec* に達するまで実行する。*bp-spec* にはbreakコマンドと同様の指定が可能(☞breakコマンド)。untilコマンドとは関数の再帰呼び出しをスキップしない点、指定位置がカレントフレーム外でも良い点が異なる。

apropos

apropos *regex*

正規表現*regex*にマッチするコマンド名の説明（ドキュメンテーション）を検索する。1つの正規表現で複数の単語を指定可能。GDBではgrepコマンドと同じ正規表現（Basic Regular Expressions）を使用しているが、検索時には大文字、小文字を区別しない。

attach

attach *pid*

実行中のプロセス*pid*にアタッチし、メモリ中のデータにアクセスする。アタッチするには適切な権限が必要。

awatch

awatch *expression*

指定した*expression*が読み取られるか、または書き込まれた場合に実行を停止するウォッチポイントを設定する（☞rwatch コマンド、watch コマンド）。

backtrace

backtrace [*count*]

全フレームを表示する。*count*の値が正の場合はフレームを内側から*count*個表示する。負の場合は外側から*count*個表示する。

break

break [*bp-spec*]
break *bp-spec* if *condition*
break *bp-spec* thread *threadnum*
break *bp-spec* thread *threadnum* if *condition*

ブレークポイントを設定する。1番目の形式では、デバッガーが常に実行を停止する、無条

件ブレークポイントを設定する。2番目の形式は、GDBが与えられた条件を評価し真の場合に実行を停止する、条件付きブレークポイントを設定する。偽の場合には停止しない。どちらの形式でも*bp-spec*には、以下の指定方法が使用できる。3番目と4番目の形式は、デバッギー内で実行しているスレッドに対して作用する点を除き、それぞれ1番目と2番目の形式と同様である。スレッド番号*threadnum*で指定したスレッドが*bp-spec*に達した場合に実行を停止する。

ブレークポイントの形式

break コマンドに指定できる形式を以下にまとめる。

break
: カレントフレーム内の次の命令にブレークポイントを設定する。カレントフレームがもっとも内側ではない場合は、そのフレームに戻って来た時に実行が停止する finish コマンドのような動作をする。ただし finish コマンドではブレークポイントは自動的に削除される。もっとも内側のフレームの場合は、実行がブレークポイントに達した時に停止する。ループ内で停止させる場合に有用。

break *function*
: 関数名 *function* で指定した関数の先頭にブレークポイントを設定する。

break *linenumber*
: カレントファイル内の*linenumber*で指定した行番号にブレークポイントを設定する。

break *file*:*line*
: *file*で指定したファイル内の*line*で指定した行番号にブレークポイントを設定する。

break *file*:*function*
: *file* で指定したファイル内の *function* で指定した関数にブレークポイントを設定する。

break +*offset*
break -*offset*
: カレントフレーム内の停止している位置から、前方へ*offset*行の位置(+*offset*)、もしくは後方へ *offset* 行の位置(-*offset*)にブレークポイントを設定する。

break *address*

*address*にブレークポイントを設定する。共有ライブラリ内など、シンボルでデバッグできないオブジェクトファイルの場合に有用である。

行や文に対してブレークポイントを設定すると、その文の最初の命令に達した時に停止する。

call

call *expression*

デバッギー内の関数を呼び出す。*expression*には関数名とその引数を指定する。非void型の場合は戻り値が表示され、値ヒストリに記憶される。

catch

catch *event*

キャッチポイントを設定する。指定した*event*が発生すると停止する。

キャッチポイントのイベント

catch

C++の例外をキャッチした。

exec

プログラムがexecve()を呼び出した（この機能が実装されていないシステムもある）。

fork

プログラムがfork()を呼び出した（この機能が実装されていないシステムもある）。

throw

C++の例外がスローされた。

vfork

プログラムがvfork()を呼び出した（この機能が実装されていないシステムもある）。

cd

cd *dir*

GDB のカレントディレクトリを *dir* へ変更する。

clear

clear [*bp-spec*]

ブレークポイントを削除する。引数はbreakコマンドと同じ形式で指定する(☞breakコマンド)。

commands

commands [*bp*]
... *commands* ...
end

ブレークポイントで停止した時に自動的に実行するコマンドを設定する。*bp*を指定しない場合は、最後に設定したブレークポイント、ウォッチポイント、キャッチポイントのいずれかにコマンドを設定する。最後に実行したものではない点に注意。コマンドを削除するには、commands キーワード入力後に end を入力する。

complete

complete *prefix*

prefix から始まるコマンド名を表示する(補完候補)。Emacs バッファ内で GDB を実行している場合に、Emacs から使用される。

condition

condition *bp*
condition *bp expression*

指定したブレークポイントに条件を追加、削除する。1 番目の構文はブレークポイント番号

*bp*に対応する条件をすべて削除する。2番目の構文はブレークポイント番号*bp*に対応する条件として*expression*を追加する。break … ifコマンドと同義。☞break コマンド

continue

continue [*count*]

ブレークポイントで停止後に実行を再開する。*count*を指定するとignoreカウントとして使用される。☞ignore コマンド

core-file

core-file [*filename*]

引数がない場合は、コアファイルが存在しないことを示す。引数がある場合は、*filename*をコアファイルとして使用する。コアファイルとは実行中のプログラムのメモリ空間をダンプしたファイル。

define

define *commandname*
... *commands* ...
end

*commandname*という名前のユーザ定義コマンドを定義する。コマンド列が*commandname*の定義内容になる。*commandname*を入力すると、GDBがコマンド列を実行する。通常のプログラミング言語の関数、プロシージャに相当する。☞document コマンド

フック

*commandname*がhook-*command*という形式をしており、*command*がGDBが内蔵しているコマンド名と一致する場合では、*command*の実行直前に*commandname*を自動的に実行する。同様に*commandname*がhookpost-*command*という形式をしている場合は、*command*終了直後に*commandname*を実行する。

GDBはデバッギー停止時に必ずstopという名前の疑似コマンドを「実行」している。フック機能を利用し、hook-stopという名前のフックを定義すると、プログラムが停止するたびに

特定のコマンドを実行することも可能。

delete

```
delete [breakpoints] [range ...]
delete display dnums ...
delete mem mnums ...
```

1番目の構文では指定された範囲のブレークポイント、ウォッチポイント、キャッチポイントを削除する。引数がない場合は、すべてのブレークポイントを削除する(set confirm設定に応じ、GDBは削除確認を問い合わせる)。2番目の構文では(displayコマンドで設定した)自動表示設定を削除する(☞displayコマンド)。3番目の構文はmemコマンドで作成したメモリリージョンを削除する。☞memコマンド

detach

```
detach
```

アタッチした実行中のプロセスをデタッチする。

directory

```
directory [dirname ...]
```

GDBがソースファイルを検索する際に参照するディレクトリリサーチパスの先頭へ*dirname*を追加する。引数がない場合は、ディレクトリサーチパスを削除する。

disable

```
disable [breakpoints] [range ...]
disable display dnums ...
disable mem mnums ...
```

1番目の構文では指定された*range*内のブレークポイントを無効化する。*range*が指定されなければすべてのブレークポイントを無効化する。ブレークポイントは無効化されても削除はされない。しかしデバッギーの実行には影響を与えない。2番目の構文では自動表示設定*dnums*(複数指定可)を無効化する(☞displayコマンド)。3番目の構文はメモリリージョン

mnums（複数指定可）を無効化する。☞mem コマンド

disassemble

```
disassemble
disassemble pc-val
disassemble start end
```

メモリの内容をアセンブリコードとして表示する。引数がない場合は現在の関数全体を逆アセンブルする。引数が1つの場合は、プログラムカウンタをあらわし、この値を含む関数を逆アセンブルする。引数が2つの場合は、逆アセンブルするアドレス範囲をあらわし、*start* 以上 *end* 未満の範囲を逆アセンブルする。

display

```
display
display/format expression
```

デバッギーが停止するたびに自動的に表示する *expression*（通常は変数やアドレス）を追加する。*format* にはxコマンドの書式を1つ指定する（☞x コマンド）。/*format* はdisplayコマンドの直後に書き、空白を置かない。引数がない場合は、設定されている式の現在の値を表示する。

document

```
document commandname
... text ...
end
```

ユーザ定義コマンド*commandname*にドキュメンテーションを追加する。ドキュメンテーションはテキスト（複数行可）で記述する。documentコマンド実行後に、help *commandname* と実行すると *text* を表示する。☞define コマンド

dont-repeat

dont-repeat

ユーザ定義コマンド内で使用するコマンド(☞defineコマンド)。ユーザ定義コマンド実行後に ENTER だけが入力された際に、ユーザ定義コマンドを繰り返さない。

down

down *count*

フレームを1段下(新しいフレーム)に移動する。*count*に正の値を指定すると、複数段移動する。☞frame コマンド、up コマンド

down-silently

down-silently *count*

メッセージを表示しない点を除き、down コマンドと同義。GDB スクリプト内で使用する。

echo

echo *strings* ...

文字列を表示する。C言語の標準的なエスケープシーケンスを使用し制御文字も表示可能。改行する場合には\nを指定する。シェルのechoコマンドとは異なり、GDBのechoコマンドは自動的に改行文字を付加しない。改行する場合には明示的に指定する。

edit

edit [*line-spec*]

line-spec で指定したソースファイルを編集する。*line-spec* の指定については list コマンドを参照。引数がない場合は、最後に表示した行を含むファイルを編集する。使用するエディタは環境変数EDITORで指定可能。環境変数が設定されていない場合は、exコマンドを使用する。

else

```
else
```

if コマンドに与えられた式が偽の場合に実行するコマンド列を指定する。コマンド列は end で終了する。☞if コマンド

enable

```
enable [breakpoints] [range ...]
enable [breakpoints] delete range ...
enable [breakpoints] once range ...
enable display dnums ...
enable mem mnums ...
```

1番目の構文はブレークポイントを有効化する。*range*を指定しない場合は全ブレークポイントを、ブレークポイントが指定された場合はそのブレークポイントだけを有効化する。2番目の構文は指定されたブレークポイントを有効化し、プログラムが実行を停止した際にブレークポイントを削除する。3番目の構文も指定されたブレークポイントを有効化し、プログラムが実行を停止した際にブレークポイントを無効化する。4番目の構文は、disableコマンドで無効化された自動表示設定を有効化する（☞displayコマンド）。5番目の構文は、メモリリージョンを有効化する。☞mem コマンド

end

```
end
```

commands、define、document、else、if、whileの各コマンドに与えたコマンド列（また文字列）を終了する。

exec-file

```
exec-file [filename]
```

引数がない場合は、実行ファイルの情報をすべて削除する。引数がある場合は、*filename*を実行ファイルとする。必要に応じて環境変数 PATH を検索する。

fg

fg [*count*]

continue コマンドの別名。☞continue コマンド

file

file
file *filename* [-mapped] [-readnow]

1 番目の構文は、シンボルファイルと実行ファイルの情報をすべて削除する。2 番目の構文は、引数の *filename* をデバッグ対象とし、シンボルテーブルにも使用する。run コマンドで実行するプログラムになる。-mapped オプションを使用すると、program.syms というファイル名でシンボルテーブルを書き出す。このファイルは以降のデバッグ時に参照可能であり、プログラムが変更されない限りは、実行ファイルからシンボルテーブルを読み取るよりも高速に読み取れる。-readnow オプションはシンボルテーブルを即座に読み取る。通常は必要になるまで読み取らない。

finish

finish

カレントフレーム(関数)がリターンするまで実行するデバッグ情報をもたない関数(ライブラリなど)に意図せず入ってしまった(シングルステップ)場合に有用。

focus

focus *window*

TUIウィンドウ*window*へフォーカスを移動する。*window*に指定できるのは、next、prev、src、asm、regs、cmd である。

forward-search

forward-search *regex*

カレント行から前方へ正規表現 *regex* にマッチする行を検索し、表示する。

frame

frame
frame *frame-num*
frame *address*

カレントフレーム（関数呼び出し）を選択または表示する。フレーム0がもっとも内側（最後に呼び出した）フレームである。引数がない場合はカレントフレームを表示する。フレーム番号 *frame-num* を指定すると、そのフレームへ移動する。よく使用される引数である。引数 *address* はそのアドレスにあるフレームを選択する際に使用する。この方法はバグによりフレームを破壊した場合に必要となる。アーキテクチャによってはアドレスを複数指定する必要がある。

generate-core-file

generate-core-file [*file*]

デバッギーの現在のメモリ内容をコアファイルに書き出す。*file* を指定すると、指定されたファイルに生成する。指定されなければ core.PID というファイル名で生成する。

handle

handle *signal keywords* ...

GDBが1つまたは複数のシグナルを処理するように設定する。*signal* にはシグナル番号、シグナル名（前に「SIG」を付けなくとも良い）、low-high 形式の範囲、all を使用する。*keywords* には以下から 1 つまたは複数を指定する。

ignore	シグナルを無視する。プログラムに渡さない。
noignore	pass と同義。
nopass	ignore と同義。

noprint	シグナルを受信した際にメッセージを表示しない。
nostop	シグナルを受信した際に停止せず、デバッギーへ渡す。
pass	シグナルをプログラムへ渡す。
print	シグナルを受信した際にメッセージを表示する。
stop	シグナルを受信した際に停止する。通常はSIGSEGVなど「エラー」のシグナルだけがプログラムを停止する。

hbreak

hbreak *bp-spec*

ハードウェアを機能を利用したハードウェアブレークポイントを設定する。引数はbreakコマンド(☞breakコマンド)と同様。hbreakコマンドはEEPROM/ROMコードのデバッグ時に使用する。位置が変更されないブレークポイントを設定可能。しかし、この機能を備えていないハードウェアもある。

help

help [*command*]

引数がない場合は、helpコマンドを使用できるサブトピックの一覧を表示する。引数*command*が指定された場合は、そのコマンドまたはコマンドグループのヘルプを表示する。

if

```
if expression
... commands1 ...
[ else
 ... commands2 ... ]
end
```

コマンド列を条件付きで実行する。*expression*が真の場合は、*commands1*を実行する。else以降が記述されており、かつ*expression*が偽の場合は*commands2*を実行する。

ignore

ignore *bp count*

ブレークポイント、ウォッチポイント、キャッチポイント*bp*にignoreカウント*count*を設定する。ignoreカウントの値が正の間は条件を評価しない。

inspect

inspect *print-expressions*

printコマンドの昔の別名。現在では使用されていない。☞printコマンド

info

info [*feature*]

デバッギーの状態*feature*の情報を表示する。引数がない場合は、使用可能な*feature*の一覧を表示する。☞infoコマンド

jump

jump *location*

*location*から実行を再開する。*location*にはlistコマンドと同様の*line-spec*か、または先頭に*を加えた16進数のアドレスを指定する。continueコマンドが停止している位置から実行を再開するのに対し、jumpコマンドは異なる位置へ移動する。*location*がカレントフレーム外を指す場合は、GDBはスタックポインタ、フレームポインタなどのレジスタを保護するため、ユーザへ確認する。

kill

kill

実行中のデバッギーのプロセスを終了する。デバッグに使用するコアファイルを生成する良

い方法である。

layout

layout *layout*

TUI ウィンドウのレイアウトを *layout* へ変更する。*layout* に指定できる値は次のとおり。

asm	アセンブリウィンドウのみ。
next	次のレイアウト。
prev	前のレイアウト。
regs	レジスタウィンドウのみ。
split	ソースウィンドウとアセンブリウィンドウ。
src	ソースウィンドウのみ。

コマンドウィンドウは常に表示される。

list

list *function*
list *line-spec*

関数 *function* の先頭から(1番目の形式)、または *line-spec* を中心とした(2番目の形式)ソースコードを表示する。GDBはENTERのみを入力すると直前のコマンドを繰り返し実行するが、list コマンドの場合は、ソースコードの先を表示する。*line-spec* には次の値を指定できる。

行の指定

list *number*

行番号 *number* を中心に表示する。

list *+offset*
list *-offset*

最後に表示した行を *offset* 行移動し、その行を中心に表示する。1番目の形式では前方へ、2 番目の形式では後方へそれぞれ移動する。

list *file*:*line*
　ソースファイル *file* の *line* 行を中心に表示する。

list *file*:*function*
　ソースファイル *file* の関数 *function* を中心に表示する。同名の関数が複数のファイルで定義されている場合に有用。

list **address*
　address を含む行を中心に表示する。*address* には式を記述できる。

list *first*,*last*
　first から *last* までの行を表示する。*first*、*last* は *line-spec* の形式で指定する。

list *first*,
　first から行を表示する。

list ,*last*
　last まで行を表示する。

list +
list -
　最後に表示した行の直後（1 番目の形式）、または直前（2 番目の形式）を表示する。

macro

macro expand *expression*
macro expand-once *expression*
macro define *macro body*
macro define *macro(args) body*
macro undefine *macro*

C プロプロセッサのマクロを処理する。GDB 6.3 ではすべては実装されていない。

macro expand *expression*
　マクロを展開した結果を表示する。展開結果は実行、評価されないため、文法的に誤っていても構わない。expand を exp と短縮も可能。

macro expand-once *expression*
　expression に指定されたマクロのみを展開する。展開したマクロ内で使用されている

マクロは展開しない。expand-once を exp1 と短縮も可能。GDB 6.3 では未実装。

macro define *macro body*
macro define *macro(args) body*

body と言う実体に展開される、*macro* という名前のマクロを定義する。C と C++ の場合は、1番目の形式では定数を定義し、2番目の形式は引数をとるマクロを定義する。GDB 6.3 では未実装。

macro undefine *macro*

macro という名前のマクロを削除する。削除できるのは macro define コマンドで定義したマクロのみ。デバッギー内のマクロは削除できない。GDB 6.3 では未実装。

make

make [*args*]

make プログラムを起動する。*args* が指定されていれば引数として渡す。shell make args コマンドと同義。GDB を終了せずにプログラムを再コンパイルする場合に有用。

mem

mem *start-addr end-addr attributes ...*

メモリリージョンを定義する。すなわち *start-addr* から始まり *end-addr* で終了する特定の属性を持ったアドレス空間。

メモリ属性

ro	リードオンリー。
rw	リード／ライト。
wo	ライトオンリー。
8、16、32、64	指定されたビット幅でメモリへアクセスする。メモリにマッピングしたデバイスレジスタへアクセスする際によく使用される。

next

next [*count*]

次のソース行を実行する。stepコマンドと異なり、関数呼び出しも1行と見なし、呼び出した関数内には入らない。*count*を指定した場合は、次の*count*数分ソース行を実行する。どちらの実行でもブレークポイントに達した場合や、シグナルを受信した場合は停止する。☞step コマンド

nexti

nexti [*count*]

実行する単位が命令(機械語)である点を除き、next コマンドと同義。

nosharedlibrary

デバッギーからすべての共有ライブラリをアンロードする。

output

output *expression*
output/*format expression*

expression のみを表示する。改行も付加されず、$n = も表示されない。また値もヒストリに記憶されない。/*format* を指定した場合は、表示に*format* を使用する。*format* 指定はprint コマンドと同様である。☞print コマンド

path

path *dir*

環境変数 PATH の先頭にディレクトリ *dir* を挿入する。

print

print [/*format*] [*expression*]

expression の値を表示する。1 番目の引数が /*format* の場合は、指定された書式を使用する。*expression*を省略した場合は直前に使用した*expression*を再度使用する。この機能により、同じ *expression* を別の書式で表示することが容易になる。*format* に使用できる書式は x コマンドのサブセットになっており、x コマンドも参照のこと。

表示書式

a	アドレスとして表示する。16 進数の絶対アドレスと、もっとも近いシンボルからのオフセットを表示する。
c	文字定数として表示する。
d	符号付き 10 進整数として表示する。
f	実数として表示する。
o	8 進整数として表示する。
t	2 進整数として表示する(t は「two」をあらわす)。
u	符号なし 10 進整数として表示する。
x	16 進整数として表示する。

print-object

print-object *object*

Objective-C のオブジェクト *object* に自身の情報を表示させる。
フック関数 _NSPrintForDebugger()を定義している Objective-C のライブラリでのみ使用可能。

printf

printf *format-string*, *expressions* ...

C ライブラリの printf(3)のように、書式を指定し *expressions* を表示する。\t、\n などの単純な 1 文字エスケープシーケンスも使用可能。

ptype

```
ptype
ptype expression
ptype type-name
```

型の定義を表示する。whatisコマンドが型の名前しか表示しないのに対し、ptypeコマンドは型のすべての情報を表示する。引数がない場合（1番目の形式）では、値ヒストリ内の最後の値の型を表示する。ptype $と同義である。*expression*を指定した場合（2番目の形式）では、*expression*の型を表示する。*expression*は評価されない点に注意。++演算や関数呼び出しは実行されない。3番目の形式は*type-name*の型を表示する。*type-name*は型名、またはタグを伴ったクラス、列挙子、構造体、共用体である。☞whatis コマンド

pwd

```
pwd
```

GDBのカレントディレクトリを表示する。

quit

```
quit
```

GDBを終了する。

rbreak

```
rbreak regexp
```

正規表現*regexp*にマッチするすべての関数にブレークポイントを設定する。正規表現はgrepコマンドと同じ構文を使用する（Basic Regular Expressions）。C++のオーバロードされた関数の場合に有用。

refresh

refresh

TUIの画面を再表示する。詳細は「**6章　テキストユーザインタフェース**」を参照。

return

return [*expression*]

カレントフレームから呼び出し元へリターンする。*expression*が指定されている場合は、戻り値として使用する。カレントフレームとカレント以降のフレーム(カレントフレームが呼び出した関数)を実行スタックからポップし、呼び出し元がカレントフレームになる。実行は再開されず、continueコマンドを実行しなければ停止したままである。

reverse-search

reverse-search *regex*

正規表現*regex*にマッチする行をカレント行から後方へ検索し、表示する。

run

run [*arguments*]

デバッギーを実行する。*arguments*が指定された場合はコマンドラインの引数として渡す。単純なI/Oリダイレクション(<、>、>>)も使用可能。パイプには対応していない。最後に使用した*arguments*は記憶されており、引数を伴わないrunコマンドは同じ*arguments*を使用し、プログラムを再実行する。プログラムの引数を消去、変更するにはset argsコマンドを使用する。デバッギーが受け取る環境は次のとおり。runコマンドの引数、GDBが継承しset environmentコマンドにより変更した環境変数、カレントディレクトリ、標準入力、標準出力、標準エラー(リダイレクトされていない場合)。

rwatch

rwatch *expression*

*expression*が読み取られた場合に停止するウォッチポイントを設定する（☞awatchコマンド、watch コマンド）。

search

search *regex*

forward-search コマンドの別名。☞forward-search コマンド

section

section *sectname address*

*sectname*のベースアドレスを*address*へ変更する。実行ファイルがセクションアドレスのデータを持っていない場合や、ファイル内のデータが狂っている場合に使用する最後の手段。

select-frame

select-frame
select-frame *frame-num*
select-frame *address*

メッセージを表示しない点を除き、frame コマンドと同義。☞frame コマンド

set

set [*variable*]

GDBの変数、またはデバッギー内の変数を設定、変更する。詳細は「**8章　setコマンド、showコマンド**」を参照。

sharedlibrary

sharedlibrary [*regexp*]

引数がない場合は、プログラムやコアファイルが必要とする共有ライブラリをすべてロードする。正規表現 *regexp* がある場合は、マッチするファイルのみをロードする。

shell

shell [*command args*]

GDBを終了せずシェルを実行し、引数*command*を実行する。引数がない場合はインタラクティブシェルを起動する。

show

show [*variable*]

GDB 内部の変数を表示する。詳細は「8 章　set、show コマンド」を参照。

signal

signal *sig*

実行を再開し、シグナル *sig* を送信する。*sig* には、シグナル番号かシグナル名を指定する。シグナル番号0は特殊で、シグナルを受信しプログラムが実行を停止した場合に、シグナル番号 0 を送信すると GDB は元のシグナルをプログラムへ渡さずに実行を再開する。

silent

silent

ブレークポイントに達した際のメッセージを表示しない。コマンド列内で使用する。

☞commands コマンド

source

source *file*

*file*からコマンドを読み取り、実行する。コマンドは表示されず、また*file*内のいずれかのコマンドでエラーが発生すれば、*file*全体の実行を中止する。通常はユーザへ問い合わせるコマンドは問い合わせない。また、メッセージを明示的に表示するコマンド以外は何も表示しない。

step

step [*count*]

次のソース行を実行する。nextコマンドとは異なり、次の行が関数呼び出しの場合に関数内に入り、呼び出された関数内でシングルステップする。nextコマンドは関数内に入らず呼び出した関数を1行とみなす。*count*を指定した場合は、stepコマンドは*count*行数分実行する。どちらの場合でもブレークポイントに達するかまたはシグナルを受信すると実行を停止する。

☞next コマンド

stepi

stepi [*count*]

次の命令（機械語）を実行する以外は関数内に入りシングルステップするstepコマンドと同様。*count*を指定した場合は、stepコマンドは*count*命令数分実行する。

symbol-file

symbol-file *filename* [-mapped] [-readnow]

引数がない場合は、すべてのシンボル情報を削除する。引数がある場合は、指定された*filename*からシンボル情報を読み取り、また*filename*を実行ファイルとみなす。ファイルの検索には環境変数PATHを使用する。-mapped、-readnowオプションはfileコマンドと同様。

☞file コマンド

tbreak

tbreak *bp-spec*

一時ブレークポイントを設定する。引数はbreakコマンドと同様(☞breakコマンド)だが、ブレークポイントに達すると停止後にブレークポイントを削除する点が異なる。

tcatch

tcatch event

一時キャッチポイントを設定する。引数は catch コマンドと同様(☞catch コマンド)だが、キャッチポイントに達すると停止後にキャッチポイントを削除する点が異なる。

thbreak

thbreak *bp-spec*

ハードウェアの機能を利用した一時ブレークポイントを設定する。引数はhbreakコマンドと同様(☞hbreak コマンド)。

thread

thread *threadnum*
thread apply [*threadnum* | all] *command*

1 番目の形式はカレントスレッド(GDB が処理対象としているスレッド)に番号 *threadnum* を割り振る。2番目の形式は、*threadnum*で指定したスレッドまたはすべてのスレッドに*command* を適用する。

tty

tty *device*

デバッギーの入力、出力を *device* から行う(通常は端末に対応するデバイスファイル)。

tui

tui reg *regkind*

TUIを使用している場合に、レジスタセット*regkind*を表示し、レジスタウィンドウを再表示する。

レジスタセット

regkind には次の値が使用可能。

float	浮動小数点レジスタ。
general	汎用レジスタ。
next	「next」レジスタグループ。既定義のレジスタグループは、all、float、general、restore、save、system、vector。
system	システムレジスタ。

undisplay

undisplay *dnums* ...

自動表示設定から *dnums* を削除する。☞display コマンド

unset

unset environment *variable*

デバッギーの環境から環境変数 *variable* を削除する。

until

until [*location*]

カレント行の次のソース行に達するまで実行する。ループの次の行まで実行する場合に有用。*location*を指定しない場合は、untilコマンドは次のソース行に達するまでシングルステップを繰り返す。*location* を指定した場合は、次のソース行に内部的にブレークポイントを設定

する。後者の方が高速に動作する。*location*はbreakコマンドと同様に指定する。☞breakコマンド

up

up *count*

フレームを1段上(古いフレーム)に移動する。*count*に正の値を指定すると、複数段移動する。☞frame コマンド、down コマンド

up-silently

up-silently *count*

メッセージを表示しない点を除き、up コマンドと同義。GDB スクリプト内で使用する。

update

update

TUI の場合にソースウィンドウとカレント実行ポイントを再表示する。

watch

watch *expression*

expression が変更された場合に停止するウォッチポイントを設定する(☞awatch コマンド、rwatch コマンド)。

whatis

whatis [*expression*]

引数がない場合は、値ヒストリ内の最後の値の型を表示する。whatis $と同義。*expression*を指定した場合は、*expression* の型を表示する。*expression* を評価しない点に注意。++ 演算や

関数呼び出しなどは実行されない。☞ptype コマンド

where

where [*count*]

backtrace コマンドと同義。☞backtrace コマンド

while

```
while expression
... commands ...
end
```

コマンド列を繰り返し実行する。*expression* が真の間は *commands* を実行する。

winheight

winheight *win* ± *amount*

TUIの場合に、ウィンドウ *win* の高さを amount に変更する。+ を指定すると高さは伸び、- の場合は縮む。ウィンドウ *win* には asm、cmd、regs、src のいずれかを指定する。

x

x [[/*NFU*] *addr*]

addr にある値を表示する。アドレスを指定せず連続的にxコマンドを使用すると、*N*、*F*、*U*の値にしたがいメモリ位置を前方へ進める。*N*は繰り返し回数であり、例えば表示する命令数などを指定する。*F*は書式であり、表示するデータ形式を指定する。*U*は表示するデータの単位バイト数を指定する。xコマンドで表示したアドレスは$_変数に、またアドレスの内容は$__変数に、それぞれ記憶される。

書式

a	アドレスとして表示する。16 進数の絶対アドレスと、もっとも近いシンボルからのオフセットを表示する。
c	文字定数として表示する。
d	符号付き 10 進整数として表示する。
f	実数として表示する。
i	機械語(命令)として表示する。
o	8 進整数として表示する。
s	NUL 文字で終端する文字列として表示する。
t	2 進整数として表示する(t は「two」をあらわす)。
u	符号なし 10 進整数として表示する。
x	16 進整数として表示する。

単位バイト数

b	バイト。
g	ジャイアントワード。8 バイト。
h	ハーフワード。2 バイト。
w	ワード。4 バイト。

付録 A
GCCを利用したデバッグ方法

デバッガ以外にもデバッグ方法はいくつもあります。

初めにGCC、GNU C Libraryに備わっているデバッグやテスト用の機能を紹介し、次にプログラムの実行を外部からトレースするためのツールを紹介します。

A.1 メモリ割り当て

アプリケーション開発では、メモリの割り当てと解放に起因する問題に手を焼くことがあります。解放済みの領域を誤って参照したり、または二重に解放するなどのバグは一般的でしょう。メモリ内に常駐するデーモンなどでは、解放忘れもメモリ枯渇など致命的な問題になるでしょう。

手を焼く理由には問題が常に発症するとは限らない、時間をかけないと発症しないなどがあります。GDB だけでは発見しにくい問題の 1 つです。

この問題の発見、解決に特化したソフトウェア製品も市販されていますし、フリーに公開されている専用ライブラリもあります。現在のGCCではコンパイル時にも多くの問題を発見できますし、さらにGNU C Libraryでは単純ながらもこの問題の原因を突き止めるための方法が提供されています。

GNU C Library には標準的なメモリ割り当て／解放以外に mtrace、mcheck というデバッグ用のライブラリ関数、MALLOC_CHECK_という環境変数、メモリ割り当て／解放時に呼び出されるフック関数、mallinfo というメモリ割り当て／解放の統計情報を保持する構造体が追加されています。

ここではmcheckをGDBから使用する方法と環境変数MALLOC_CHECK_を紹介します。それ以外の点についてはGNU C Libraryのオンラインマニュアルやinfoページを参照してください。

mcheckライブラリ関数を使用すると、mallocによりメモリを動的に割り当てる際に、アプリケーションがこれまでに割り当てたメモリ領域を越えて使用していないことを確認できます。

異常を発見すると、デフォルトではabortライブラリ関数を呼び出し、アプリケーションを異常終了させるため、問題箇所を早期に突き止められます。

mcheckは、アプリケーション実行内の最初のmallocよりも先に呼び出す必要があります。コンパイル／リンク時に'-lmcheck'を追加する方法もありますが、GDBを使用する方法もあります。次のようにGDBを使用するとGDB内でデバッギーを実行する際に自動的にmcheckを呼び出します。ただし、main関数以前に実行されるスタティックオブジェクトのコンストラクタなどがある場合には有効ではありません。

また、mcheckを使用しても実行時間節約のため、mallocは毎回はチェックしません。時間がかかっても毎回チェックしたい場合は、mcheckの代わりに、明文化されていないmcheck_pedanticを使用します。使用方法はmcheckと同様です。

```
(gdb) break main
Breakpoint 1, main (argc=2, argv=0xbffff964) at whatever.c:10
(gdb) command 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>call mcheck(0)
>continue
>end
(gdb) ...
```

環境変数MALLOC_CHECK_はもっと簡単に利用できます。リンクするライブラリを追加する必要もありません。MALLOC_CHECK_に2を設定し、(GDBを使用せず)通常通りアプリケーションを実行します。GNU C LibraryがMALLOC_CHECK_を認識し、mcheckと同様に異常発見時にabortします。

SUIDまたはSGIDが設定されたアプリケーションの場合は、デフォルトでは環境変数MALLOC_CHECK_は機能しません。詳細はGNU C Libraryのオンラインマニュアルやinfoページを参照してください。

A.2 プロファイル、カバレッジ

GDB同様にアプリケーション開発のテスト、デバッグ時に使用されるツールに、プロファイラ(GNU gprof)、テストカバレッジ(gcov)があります。どちらもGCCとGNU C Libraryを組み合わせて使用します。

プロファイラはアプリケーション実行時に情報を収集し、呼び出し回数の多い関数や、処理時間の長い関数などを特定するのに使用できます。GDBを使用するにはコンパイル時に-gオプションを付加しますが、プロファイラを使用する場合は-pgオプションを加えます。

通常通りアプリケーションを実行すると、終了時にカレントディレクトリにgmon.outというファイルが生成されます。

アプリケーションが内部でカレントディレクトリを移動した場合は、起動したディレクトリとは異なるディレクトリに生成されます。gmon.outにはアプリケーションの実行をプロファイリングした情報が収められており、gprof コマンドにより内容を解析、表示します。

通常は関数単位でのプロファイリングしますが、コンパイル時に-pgオプションと-gオプションを両方とも使用し、gprof コマンドに-l オプションを使用するとソース行単位の情報収集、表示が可能になります。

テストカバレッジはアプリケーションテスト時に使用し、ソースコードのどの部分が実行されたかを確認できます。使用方法はコンパイル時に -fprofile-arcs オプションと -ftest-coverage オプションを与え、その後通常通りアプリケーションを実行します。前述のプロファイル情報同様に実行時に収集された実行したソースコードの行単位の情報がファイルとして生成されます。

gcovコマンドにソースファイル名を与え起動すると、生成されたファイルを参照し、実行したソースコード行の割合い(テスト網羅率)などを表示し、解析結果をファイルに出力します。

付録 B
strace、ltrace

strace、ltraceはいずれもアプリケーションの実行を外部からトレースする、GNU/Linuxのコマンドです。Solarisではtrussコマンド、BSDではktraceコマンドが同等の機能を持ちます。

GDBのように引数にコマンド名を与えると新たにプロセスを生成し、与えられたコマンドを実行し、発行されたシステムコールや受信したシグナルを表示します。またプロセスIDを与え、実行中のプロセスにアタッチすることも可能です。システムコール以外にライブラリもトレース対象にする場合は、strace コマンドの代わりに ltrace コマンドを使用します。

表示内容はシステムコールの引数、戻り値以外にも、時間情報も加えられ、またトレース対象とするシステムコールも指定できます。

インタラクティブ性が高いGDBとは異なり、バッチ的にアプリケーションの動作を観察する場合に簡単に利用できるでしょう。リコンパイルも不要ですし、デバッグ情報を含んでいない既存のアプリケーションの動作も観察できる点も有用です。

索　引

記号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

あ行

か行

さ行

た行

な行

は行

ま行

や行

ら行

●著者紹介

Arnold Robbins（アーノルド・ロビンス）

アトランタ生まれ。プログラマ兼テクニカルライター。PDP-11 上の UNIX Version 6 と出会った1980年からUNIX関連の仕事に従事。1987年からawkのヘビーユーザであり、awkのGNUバージョンであるgawkの開発にもかかわり、現在はメンテナンスと文書を担当している。POSIX 1003.2の選考委員の一人として、awkのPOSIX準拠にも一役買っている。入門 vi 第 6 版、sed & awk プログラミング改訂版、Linux クイックリファレンス、Unix クイックリファレンスなど、著書多数。イスラエル在住。

●訳者紹介

千住 治郎（せんじゅ じろう）

獨協大学前田ゼミ卒。普及しているプログラミング言語以外にもAPLなど少数派の言語も経験する。昭和63年からUNIXを使用し始め、ソフトウェア開発を行っている。訳書に『PDF Hacks』（オライリー・ジャパン）ほか。

GDBハンドブック

2005年9月2日　初版第1刷発行
2009年6月17日　初版第4刷発行

著　者　Arnold Robbins（アーノルド・ロビンス）
訳　者　千住 治郎（せんじゅ じろう）
発行人　ティム・オライリー
印刷　株式会社ルナテック
製本　株式会社越後堂製本
発行所　株式会社オライリー・ジャパン
〒160-0002　東京都新宿区坂町26番地27　インテリジェントプラザビル1F
Tel　（03）3356-5227
Fax　（03）3356-5263
電子メール　japan@oreilly.co.jp
発売元　株式会社オーム社
〒101-8460　東京都千代田区神田錦町3-1
Tel　（03）3233-0641（代表）
Fax　（03）3233-3440

Printed in Japan（ISBN4-87311-246-X）
乱丁、落丁の際はお取り替えいたします。