

# Debugging with GDB

---

The GNU Source-Level Debugger

Seventh Edition, for GDB version 4.18  
February 1999

Richard M. Stallman and Roland H. Pesch

---

(Send bugs and comments on GDB to [bug-gdb@prep.ai.mit.edu](mailto:bug-gdb@prep.ai.mit.edu).)

*Debugging with GDB*

T<sub>E</sub>Xinfo 1999-03-15.17

Copyright © 1988-1999 Free Software Foundation, Inc.

Published by the Free Software Foundation

59 Temple Place - Suite 330,

Boston, MA 02111-1307 USA

Printed copies are available for \$20 each.

ISBN 1-882114-11-6

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

## GDB の要約

GDB のようなデバッガの目的は、実行中のプログラムの内部において何が起きているのか、あるいは、プログラムがクラッシュしたときに何をしていたのかを知ることができるようにすることにあります。

GDB は、実際のバグを発見できるようにするために 4 つのこと（さらに、これらを支援するために他のことも）を行います。

- ユーザ・プログラムを、その動作に影響を与える可能性のある様々なことを指定して起動する
- 指定された条件が成立したときにユーザ・プログラムを停止する
- 停止したときにユーザ・プログラムが何を行っていたかを調べる
- ユーザ・プログラムの内部を変更することによって、1 つのバグの影響を試験的に修正して、ほかのバグについて調べる

GDB を使って C および C++ で記述されたプログラムをデバッグすることができます。詳細については、See Section 9.4.1 [C and C++], page 80。

Modula-2 と Chill のサポートはまだ部分的なものです。Modula-2 に関する情報については、See Section 9.4.2 [Modula-2], page 85。Chill に関するドキュメントはまだありません。

集合、サブ範囲 (subrange)、ファイル変数、入れ子関数を使っている Pascal プログラムをデバッグすることは、現時点ではできません。Pascal の構文を使って、式の入力、変数の値の表示、およびそれに類することを実行することを、GDB はサポートしていません。

GDB は、Fortran で記述されたプログラムのデバッグに使うことができます。しかし、Fortran の構文を使って、式の入力、変数の値の表示、およびそれに類する機能を実行することは、まだサポートされていません。変数によっては、末尾にアンダースコアを付けて参照する必要のある場合があります。

### フリー・ソフトウェア

GDB はフリー・ソフトウェアであり、GNU General Public License (GPL) により保護されています。あなたは、GPL によって、ライセンスされたプログラムをコピーしたり改造したりする自由を与えられます。しかし、コピーを入手した人は誰でも、そのコピーとともに、そのコピーを修正する自由を手に入れます（つまりソース・コードを入手することができなければならないということです）。また、さらにそのコピーを配布する自由も手に入れます。通常、ソフトウェア会社は、著作権によりユーザの自由を妨げます。Free Software Foundation は GPL を使ってこれらの自由を保護します。

基本的には、GPL は、「あなたはこれらの自由を与えられるが、これらの自由をほかの誰からも奪うことはできない」と主張するライセンスです。

### GDB に貢献した人々

Richard Stallman は、GDB、および、その他の多くの GNU プログラムの最初の開発者です。ほかにも多くの人々が GDB の開発に貢献してきました。この節では、主要な貢献者を紹介したいと思います。フリー・ソフトウェアの素晴らしい点の 1 つは、誰もがそれに貢献する自由があるということです。残念ながら、ここですべての人を紹介することはできません。GDB ディストリビューションに含まれる ‘ChangeLog’ というファイルにおおまかな紹介を載せてあります。

バージョン 2.0 よりもずっと前の変更内容は、いつのまにか紛失してしまいました。

お願い: このセクションへの追加は大歓迎です。あなたやあなたの友人( 公平を期するため、あなたの敵も加えておきましょう )が不当にもこのリストから除外されているのであれば、喜んで名前を付け加えます。

彼らの多大な労働が感謝されていないと思われぬように、最初に、GDB の主要なリリースを通じて GDB の面倒を見てきた人々に特に感謝します。その人々とは、Jim Blandy ( リリース 4.18 ) Jason Molenda ( リリース 4.17 ) Stan Shebs ( リリース 4.14 ) Fred Fish ( リリース 4.16, 4.15, 4.13, 4.12, 4.11, 4.10, 4.9 ) Stu Grossman と John Gilmore ( リリース 4.8, 4.7, 4.6, 4.5, 4.4 ) John Gilmore ( リリース 4.3, 4.2, 4.1, 4.0, 3.9 ) Jim Kingdon ( リリース 3.5, 3.4, 3.3 ) Randy Smith ( リリース 3.2, 3.1, 3.0 ) です。

Richard Stallman は、様々な機会に Peter TerMaat、Chris Hanson、Richard Mlynarik の支援を受けながら、2.8 までのリリースを担当しました。

Michael Tiemann は、GDB における GNU C++ サポートのほとんどを開発してくれました。C++ のサポートについては、Per Bothner から重要な貢献がありました。James Clark は GNU C++ のデマングラ ( demangler ) を開発してくれました。C++ についての初期の仕事は Peter TerMaat によるものです ( 彼はまた、リリース 3.0 までの一般的なアップデート作業の多くを担当してくれました )。

GDB 4 は、複数のオブジェクト・ファイル・フォーマットを調べるのに BFD サブルーチン・ライブラリを使用しています。BFD は、David V. Henkel-Wallace、Rich Pixley、Steve Chamberlain、John Gilmore による共同プロジェクトです。

David Johnson は、最初の COFF サポートを開発してくれました。Pace Willison は最初のカプセル化された COFF ( encapsulated COFF ) のサポートを開発してくれました。

Harris Computer Systems 社の Brent Benson は、DWARF 2 のサポート部分を提供してくれました。

Adam de Boor と Bradley Davis は ISI Optimum V のサポート部分を提供してくれました。Per Bothner、引地信之、Alessandro Forin は、MIPS のサポート部分を提供してくれました。Jean-Daniel Fekete は Sun 386i のサポート部分を提供してくれました。Chris Hanson は HP9000 サポートを改善してくれました。引地信之と長谷井智之は、Sony/News OS 3 のサポート部分を提供してくれました。David Johnson は Encore Umax のサポート部分を提供してくれました。Jyrki Kuoppala は Altos 3068 のサポート部分を提供してくれました。Jeff Law は HP PA と SOM のサポート部分を提供してくれました。Keith Packard は NS32K のサポート部分を提供してくれました。Doug Rabson は Acorn Risc Machine のサポート部分を提供してくれました。Bob Rusk は Harris Nighthawk CX-UX のサポート部分を提供してくれました。Chris Smith は Convex のサポート ( および、Fortran デバッグのサポート ) 部分を提供してくれました。Jonathan Stone は Pyramid のサポート部分を提供してくれました。Michael Tiemann は SPARC のサポート部分を提供してくれました。Tim Tucker は Gould NP1 と Gould Pownode のサポート部分を提供してくれました。Pace Willison は Intel 386 のサポート部分を提供してくれました。Jay Vosburgh は Symmetry のサポート部分を提供してくれました。

Andreas Schwab は M68K Linux のサポート部分を提供してくれました。

Rich Schaefer と Peter Schauer は SunOS 共用ライブラリのサポートを手伝ってくれました。

Jay Fenlason と Roland McGrath は、GDB と GAS がいくつかのマシン命令セットに関して共通の認識を持つようにしてくれました。

Patrick Duval、Ted Goldstein、Vikram Koka、Glenn Engel はリモート・デバッグ機能の開発を手伝ってくれました。Intel 社、Wind River Systems 社、AMD 社、ARM 社はそれぞれ、i960、VxWorks、A29K UDI、RDI ターゲット用のリモート・デバッグ・モジュールを提供してくれました。

Brian Fox は、コマンドライン編集やコマンドライン・ヒストリを提供する readline ライブラリの開発者です。

SUNY Buffalo の Andrew Beers は言語切り替えのソース・コードと Modula-2 サポートを開発し、このマニュアルのプログラミング言語関連 ( Languages ) の章を提供してくれました。

Fred Fish は Unix System Vr4 サポートのほとんどを開発してくれました。彼はまた、C++ のオーバーロードされたシンボルを扱うようコマンド補完機能を拡張してくれました。

Hitachi America, Ltd. は、H8/300 プロセッサ、H8/500 プロセッサ、および、Super-H プロセッサのサポートを後援してくれました。

NEC は、v850 プロセッサ、Vr4xxx プロセッサ、および、Vr5xxx プロセッサのサポートを後援してくれました。

Mitsubishi ( 三菱 ) は、D10V プロセッサ、D30V プロセッサ、および、M32R/D プロセッサのサポートを後援してくれました。

Toshiba ( 東芝 ) は、TX39 Mips プロセッサのサポートを後援してくれました。

Matsushita ( 松下 ) は、MN10200 プロセッサと MN10300 プロセッサのサポートを後援してくれました。

Fujitsu ( 富士通 ) は、SPARClike プロセッサと FR30 プロセッサのサポートを後援してくれました。

Kung Hsu, Jeff Law, Rick Sladkey はハードウェア・ウォッチポイントのサポートを追加してくれました。

Michael Snyder はトレースポイントのサポートを追加してくれました。

Stu Grossman は gdbserver を開発してくれました。

Jim Kingdon, Peter Schauer, Ian Taylor, Stu Grossman は GDB 全体にわたって、ほとんど数えることができないほどのバグ・フィックスとソース・コードの整理を行ってくれました。

Hewlett-Packard 社の Ben Krepp, Richard Title, John Bishop, Susan Macchia, Kathy Mann, Satish Pai, India Paul, Steve Rehrauer, Elena Zannoni は、PA-RISC 2.0 アーキテクチャ、HP-UX 10.20, 10.30, 11.0(narrow mode)、HP によるカーネル・スレッドの実装、HP aC++コンパイラ、および、端末ユーザ・インターフェイスの各サポート部分を提供してくれました。また、このマニュアルの中の HP 固有の情報は、Kim Haase により提供されたものです。

Cygnus Solutions 社は、1991 年以降、GDB の保守作業と GDB の多くの開発作業を後援しています。フルタイムで GDB に関わる仕事をした Cygnus のエンジニアは、Mark Alexander, Jim Blandy, Per Bothner, Edith Epstein, Chris Faylor, Fred Fish, Martin Hunt, Jim Ingham, John Gilmore, Stu Grossman, Kung Hsu, Jim Kingdon, John Metzler, Fernando Nasser, Geoffrey Noer, Dawn Perchik, Rich Pixley, Zdenek Radouch, Keith Seitz, Stan Shebs, David Taylor, Elena Zannoni です。さらに、Dave Brolley, Ian Carmichael, Steve Chamberlain, Nick Clifton, JT Conklin, Stan Cox, DJ Delorie, Ulrich Drepper, Frank Eigler, Doug Evans, Sean Fagan, David Henkel-Wallace, Richard Henderson, Jeff Holcomb, Jeff Law, Jim Lemke, Tom Lord, Bob Manson, Michael Meissner, Jason Merrill, Catherine Moore, Drew Moseley, Ken Raeburn, Gavin Romig-Koch, Rob Savoye, Jamie Smith, Mike Stump, Ian Taylor, Angela Thomas, Michael Tiemann, Tom Tromey, Ron Unrau, Jim Wilson, David Zuhn は、大小様々な貢献をしてくれました。



## 1 GDB セッションのサンプル

その気になれば、このマニュアルを使って GDB のすべてを学習することももちろん可能ですが、GDB を使い始めるには、いくつかのコマンドを知っていれば十分です。本章では、そのようなコマンドについて説明します。

GDB の出力情報との区別が容易につくように、このサンプル・セッションでは、ユーザの入力を `input` のように太字で表わします。

汎用的なマクロ・プロセッサである GNU m4 には、かつて、まだ正式なバージョンがリリースされる以前に、次のような不具合がありました。引用を表わす文字列をデフォルトとは異なるものに変更すると、あるマクロ定義の内部に入れ子状態になっている他のマクロ定義を取り出すために使われるコマンドが、正しく動作しなくなることがある、という不具合です。以下の短い m4 セッションでは、0000 に展開されるマクロ `foo` を定義しています。さらに、m4 の組み込みコマンド `defn` を使って、マクロ `bar` に同一の定義を与えています。ところが、引用の開始文字列を `<QUOTE>` に、引用の終了文字列を `<UNQUOTE>` にそれぞれ変更すると、全く同一の手順で新しい同義語 `baz` を定義しようとしても、うまくいかないのです。

```
$ cd gnu/m4
$ ./m4
define(foo,0000)

foo
0000
define(bar,defn('foo'))

bar
0000
changequote(<QUOTE>,<UNQUOTE>)

define(baz,defn(<QUOTE>foo<UNQUOTE>))
baz
C-d
m4: End of input: 0: fatal error: EOF in string
```

ここで GDB を使って、何が起きているのか調べてみましょう。

```
$ gdb m4
GDB is free software and you are welcome to distribute copies
of it under certain conditions; type "show copying" to see
the conditions.
There is absolutely no warranty for GDB; type "show warranty"
for details.

GDB 4.18, Copyright 1999 Free Software Foundation, Inc...
(gdb)
```

GDB は、必要なときに他のシンボルを見つけるのに最低限必要となるシンボル情報しか読み込みません。その結果、最初のプロンプトが表示されるまでの時間が極めて短いのです。ここで、出力情報がこのマニュアルの紙幅に収まるようにするために、GDB に対して表示幅を通常よりも狭くするよう指示を出してみましょう。

```
(gdb) set width 70
```

m4の組み込みコマンドである `changequote` がどのように動作するのかを調べてみる必要があります。ソースを見ると、関連するサブルーチンが `m4_changequote` であることがわかります。そこで、GDBの `break` コマンドでブレイクポイントを設定してみます。

```
(gdb) break m4_changequote
Breakpoint 1 at 0x62f4: file builtin.c, line 879.
```

`run` コマンドを使って、GDBの管理下で m4 を走らせます。m4\_changequote サブルーチンに到達するまでは、プログラムは通常どおりの動作をします。

```
(gdb) run
Starting program: /work/Editorial/gdb/gnu/m4/m4
define(foo,0000)
```

```
foo
0000
```

ブレイクポイントでプログラムを停止させるために `changequote` を実行すると、GDBは m4の実行を停止し、停止した箇所のコンテキスト情報を表示します。

```
changequote(<QUOTE>,<UNQUOTE>)

Breakpoint 1, m4_changequote (argc=3, argv=0x33c70)
  at builtin.c:879
879         if (bad_argc(TOKEN_DATA_TEXT(argv[0]),argc,1,3))
```

次に `n (next)` コマンドを実行すると、現在停止している関数の中で 1 行だけ処理が実行されます。

```
(gdb) n
882         set_quotes((argc >= 2) ? TOKEN_DATA_TEXT(argv[1])\
: nil,
```

`set_quotes` というのは、いわくありげなサブルーチンです。`next` コマンドの代わりに `s (step)` コマンドを使うことで、このサブルーチンの中に入ることができます。`step` コマンドは、それがどのサブルーチンの中にあるかということにかかわりなく、次の 1 行に移動します。この場合、次の 1 行は `set_quotes` の中ですから、そこへ移動することになります。

```
(gdb) s
set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
  at input.c:530
530         if (lquote != def_lquote)
```

m4がその中で現在停止しているサブルーチン（および、そのサブルーチンへの引数）が表示されています。これをスタック・フレーム表示と呼びます。それは、スタックの状態を要約した情報を表示しています。`backtrace` コマンド（あるいは、`bt` と省略することもできます）を使って、現在、スタック全体の中のどこにいるかを知ることができます。`backtrace` コマンドは、アクティブなサブルーチンのスタック・フレームを表示します。

```
(gdb) bt
#0  set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
#1  0x6344 in m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:882
#2  0x8174 in expand_macro (sym=0x33320) at macro.c:242
#3  0x7a88 in expand_token (obs=0x0, t=209696, td=0xf7fffa30)
    at macro.c:71
#4  0x79dc in expand_input () at macro.c:40
```



```
#5 0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

次に、2、3 行先に進んで、何が起きているのかを見てみましょう。最初の 2 回は、‘s’ コマンドを使います。続く 2 回は、xstrdup サブルーチンの中に入ってしまうのを防ぐために、n コマンドを使います。

```
(gdb) s
0x3b5c 532          if (rquote != def_rquote)
(gdb) s
0x3b80 535          lquote = (lq == nil || *lq == '\0') ? \
def_lquote : xstrdup(lq);
(gdb) n
536          rquote = (rq == nil || *rq == '\0') ? def_rquote\
: xstrdup(rq);
(gdb) n
538          len_lquote = strlen(rquote);
```

最後に表示された行は、少し妙な感じがします。2 つの変数 lquote、rquote を調べて、本当にそれが、新たに指定された引用開始文字列、引用終了文字列であるかどうか確認することができます。値を調べるには p ( print ) コマンドを使用します。

```
(gdb) p lquote
$1 = 0x35d40 "<QUOTE>"
(gdb) p rquote
$2 = 0x35d50 "<UNQUOTE>"
```

lquote と rquote は確かに引用開始文字列、引用終了文字列のようです。前後関係を調べるには、l ( list ) コマンドを使って、現在停止している行を中心にその前後 10 行を表示します。

```
(gdb) l
533          xfree(rquote);
534
535          lquote = (lq == nil || *lq == '\0') ? def_lquote\
: xstrdup (lq);
536          rquote = (rq == nil || *rq == '\0') ? def_rquote\
: xstrdup (rq);
537
538          len_lquote = strlen(rquote);
539          len_rquote = strlen(lquote);
540      }
541
542      void
```

len\_lquote と len\_rquote に値を設定している行を実行させてから、それらの値を調べてみましょう。

```
(gdb) n
539          len_rquote = strlen(lquote);
(gdb) n
540      }
(gdb) p len_lquote
$3 = 9
(gdb) p len_rquote
$4 = 7
```

len\_lquote と len\_rquote が、それぞれ lquote と rquote の長さであるとなると、ここに表示されている値は明らかに誤りです。p コマンドを使って、正しい値を設定することができます。p コマン

ドによって任意の式の値を表示することができますが、ここでいう「式」には、サブルーチンの呼び出しや、値の割り当ても含まれます。

```
(gdb) p len_lquote=strlen(lquote)
$5 = 7
(gdb) p len_rquote=strlen(rquote)
$6 = 9
```

新しい引用文字列をセットした状態で、m4の組み込みコマンド `defn` を使用しようとするると発生する問題を修正するには、これだけで十分でしょうか？ `c (continue)` コマンドを使えば、m4に処理を継続させて、実際に問題を発生させていた例を実行することができます。

```
(gdb) c
Continuing.
```

```
define(baz,defn(<QUOTE>foo<UNQUOTE>))
```

```
baz
0000
```

今度はうまくいきました。新たにセットされた引用文字列は、デフォルトの引用文字列と同じように機能しました。問題の原因は、プログラム内の2箇所のタイプ・ミスで、長さの設定が正しく行われていないことにあったようです。EOFを入力して、m4を終了させましょう。

**C-d**

```
Program exited normally.
```

‘Program exited normally.’というメッセージは、GDBが出力したもので、m4の実行が終了したことを意味しています。GDBの `quit` コマンドで、GDBセッションを終了することができます。

```
(gdb) quit
```

## 2 GDB の起動・終了

本章では、GDB の起動方法、終了方法を説明します。基本は、以下の 2 つです。

- ‘gdb’ と入力して GDB を起動する
- *quit* または *C-d* を入力して GDB を終了する

### 2.1 GDB の起動

*gdb* というプログラムを実行することで、GDB が起動されます。ひとたび起動されると、GDB は終了を指示されるまで、端末からのコマンド入力を受け付けます。

あるいは、最初から GDB のデバッグ環境を指定するために、様々な引数やオプションを指定して *gdb* プログラムを実行することもできます。

ここで説明するコマンドライン・オプションは、様々な状況に対応するために設計されたものです。環境によっては、ここで説明するオプションのいくつかは、事実上使用できない場合もあります。

GDB の最も基本的な起動方法は、デバッグされる実行プログラムの名前を引数に指定することです。

```
gdb program
```

起動時に、実行プログラム名とともに、コア・ファイルの名前を指定することもできます。

```
gdb program core
```

あるいは、既に実行中のプロセスをデバッグする場合には、そのプロセス ID を第 2 引数に指定することもできます。

```
gdb program 1234
```

ここでは、GDB はプロセス ID 1234 のプロセスにアタッチします（ただし、‘1234’ という名前のファイルが存在しないというのが条件です。GDB は、まずコア・ファイルの存在を確認します）。

このような第 2 引数の利用が可能であるためには、かなり完成されたオペレーティング・システムが必要になります。ボード・コンピュータに接続して、リモート・デバッガとして GDB を使用する場合には、そもそも「プロセス」という概念がないかもしれませんし、多くの場合、コア・ダンプというものもないでしょう。

*gdb* を起動すると、GDB の無保証性を説明する文章が表示されますが、*-silent* オプションを指定することで、これを表示しないようにすることもできます。

```
gdb -silent
```

コマンドライン・オプションを指定することで、GDB の起動方法をさらに制御することができます。GDB 自身に、使用可能なオプションを表示させることができます。

```
gdb -help
```

のように *gdb* プログラムを実行することで、使用可能なオプションがすべて、その使用方法についての簡単な説明付きで表示されます（短縮して、‘*gdb -h*’ という形で実行しても同じ結果が得られます）。

ユーザの指定したすべてのオプションと引数は、順番に処理されます。‘*-x*’ オプションが指定されている場合は特別で、順序の違いに意味がでてきます。

### 2.1.1 ファイルの選択

起動された GDB は、指定された引数のうちオプション以外のものは、実行ファイル名およびコア・ファイル名（あるいはプロセス ID）であると解釈します。これは、`-se` オプションと `-c` オプションが指定されたのと同じことです（GDB は、対応するオプション・フラグを持たない最初の引数を `-se` オプション付きと同等とみなし、同じく対応するオプション・フラグを持たない第 2 の引数があれば、これを `-c` オプション付きと同等とみなします）。

多くのオプションには、完全形と短縮形があります。以下の一覧では、その両方を示します。オプション名は、他のオプションと区別がつけば、最後まで記述しなくても、GDB によって正しく認識されます（オプション名には `-` ではなく `--` を使うことも可能ですが、ここでは一般的な慣例にしたがうこととします）。

- `-symbols file`
- `-s file`      *file* で指定されるファイルからシンボル・テーブルを読み込みます。
- `-exec file`
- `-e file`      可能であれば、*file* で指定されるファイルを、実行ファイルとして使います。また、このファイルを、コア・ダンプとともにデータを解析するために使います。
- `-se file`      *file* で指定されるファイルからシンボル・テーブルを読み込み、かつ、このファイルを実行ファイルとして使います。
- `-core file`
- `-c file`      *file* で指定されるファイルを解析すべきコア・ダンプとして使います。
- `-c number`      *number* で指定されるプロセス ID を持つプロセスに接続します。これは、`attach` コマンドを実行するのと同様です（ただし、*number* で指定される名前のコア・ダンプ形式のファイルが存在する場合は、そのファイルをコア・ダンプとして読み込みます）。
- `-command file`
- `-x file`      *file* で指定されるファイル内に記述された GDB コマンドを実行します。See Section 15.3 [Command files], page 139。
- `-directory directory`
- `-d directory`      ソース・ファイルを検索するパスに *directory* で指定されるディレクトリを追加します。
- `-m`
- `-mapped`      注意: このオプションは、すべてのシステムでサポートされているわけではない、オペレーティング・システムのある機能に依存しています。  
システム上で、`mmap` システム・コールによるファイルのメモリへのマッピングが使用可能な場合、このオプションを使うことで、プログラムのシンボル情報を再利用可能なファイルとしてカレント・ディレクトリに書き出させることができます。仮にデバッグ中のプログラム名が `/tmp/fred` であるとする、マップされたシンボル・ファイルは `./fred.syms` となります。この後の GDB デバッグ・セッションは、このファイルの存在を検出し、そこから迅速にシンボル情報をマップします。この場合、実行プログラムからシンボル情報を読み込むことはありません。  
  
`.syms` ファイルは、GDB が実行されるホスト・マシンに固有のもので、このファイルは、内部の GDB シンボル・テーブルのイメージをそのまま保存したものです。これを、複数のホスト・プラットフォーム上において、共有することはできません。

-r

-readnow シンボル・ファイル内のシンボル・テーブル全体をただちに読み込みます。デフォルトの動作では、シンボル情報は必要になるたびに徐々に読み込まれます。このオプションを使うと起動までに時間がかかるようになりますが、その後の処理は速くなります。

-mappedオプションと-readnowオプションは、完全なシンボル情報を含む‘.syms’ファイルを作成するために、通常は一緒に指定されます(‘.syms’ファイルに関する詳細については、See Section 12.1 [Commands to specify files], page 99)。後に使用する目的で‘.syms’を作成するだけで、それ以外には何もしないようにするための GDB の単純な起動方法は、以下のとおりです。

```
gdb -batch -nx -mapped -readnow programname
```

### 2.1.2 モードの選択

GDB を様々なモードで実行することが可能です。例えば、batch モードや quiet モードなどがあります。

-nx

-n 初期化ファイルに記述されたコマンドを実行しません(通常、初期化ファイルは‘.gdbinit’という名前です。ただし、PC 上では‘gdb.ini’となります)。通常は、すべてのコマンド・オプションと引数が処理された後に、初期化ファイル内のコマンドが実行されます。See Section 15.3 [Command files], page 139。

-quiet

-q 紹介メッセージおよびコピーライト・メッセージを表示しません。これらのメッセージは、batch モードでも表示されません。

-batch

batch モードで実行されます。‘-x’オプションで指定されたすべてのコマンド・ファイルを処理した後、終了コード 0 で終了します(‘-n’オプションによって禁止されていなければ、初期化ファイル内に記述されているすべてのコマンドも実行されます)。コマンド・ファイルに記述された GDB コマンドの実行中にエラーが発生した場合には、0 以外の終了コードで終了します。

batch モードは GDB をフィルタとして実行する場合に便利です。例えば、あるプログラムを別のコンピュータ上にダウンロードして実行する場合などです。このような使い方の邪魔にならないよう、

```
Program exited normally.
```

というメッセージは、batch モードでは表示されません(通常このメッセージは、GDB の管理下で実行中のプログラムが終了するときに、必ず表示されます)。

-cd *directory*

カレント・ディレクトリではなく、*directory* で指定されたディレクトリを作業ディレクトリとして、GDB を実行します。

-fullname

-f GNU Emacs が GDB をサブ・プロセスとして起動するとき、このオプションを指定します。このオプションは、スタック・フレームを表示するときには、必ず完全なファイル名と行番号を標準的な認識可能な書式で出力するよう GDB に対して指示するものです(スタック・フレームは、例えば、プログラムの実行が停止されたときに必ず表示されます)。認識可能な書式とは、先頭に 2 つの‘\032’文字、続いてコロンの区切られたファイル名、行番号、桁位置、最後に改行、というものです。Emacs-GDB インターフェイ

ス・プログラムは、フレームに対応するソース・コードを表示させる命令として、2つの‘\032’文字を使用します。

`-b bps` GDBによってリモート・デバッグ用に使用されるシリアル・インターフェイスの回線速度(ボーレートあるいはBPS)を設定します。

`-tty device` プログラムの標準入力および標準出力として *device* を使用して実行します。

## 2.2 GDB の終了

`quit` GDBを終了するためには、`quit`コマンド(省略形は`q`)を使用するか、あるいは、ファイルの終端文字(通常は`C-d`)を入力します。*expression*を指定しない場合、GDBは正常終了します。*expression*が指定された場合、*expression*の評価結果をエラー・コードとして終了します。

割り込み(多くの場合`C-c`)はGDBを終了させません。割り込みは通常、実行中のGDBコマンドを終了させ、GDBのコマンド・レベルに復帰させます。割り込み文字は、いつ入力しても安全です。というのは、割り込みの発生が危険である間は、GDBが割り込みの発生を抑止するからです。

アタッチされたプロセスやデバイスを制御するためにGDBを使用していた場合、`detach`コマンドでそれを解放することができます(see Section 4.7 [Debugging an already-running process], page 23)。

## 2.3 シェル・コマンド

デバッグ・セッションの途中でシェル・コマンドを実行する必要がある場合、GDBを終了したり一時停止させたりする必要はありません。`shell`コマンドを使用することができます。

`shell command string`  
*command string*で指定されるコマンド文字列を実行するために標準シェルを起動します。SHELL環境変数が設定されていれば、その値が実行されるべきシェルを決定します。SHELL環境変数が設定されていなければ、GDBは`/bin/sh`を実行します。

開発環境ではしばしば`make`ユーティリティが必要とされます。GDB内部で`make`ユーティリティを使用する場合は、`shell`コマンドを使用する必要はありません。

`make make-args`  
*make-args*で指定される引数とともに`make`プログラムを実行します。これは、`'shell make make-args'`を実行するのと同じことです。

## 3 GDB コマンド

GDB コマンドの名前は、最初の 2、3 文字に省略することができます。ただし、省略されたコマンド名があいまいであってはなりません。さらに、同じ GDB コマンドを連続して使用する場合には、`(RET)` キーを押すだけで十分です。また、`(TAB)` キーを押すことで、途中まで入力されたコマンド名を補完させることができます（複数の補完候補がある場合には、その一覧を表示します）。

### 3.1 コマンドの構文

GDB コマンドは 1 行で入力されます。1 行の長さには上限がありません。行は、コマンド名で始まり、コマンド名によって意味が決まる引数がそれに続きます。例えば、`step` コマンドは `step` を実行する回数を引数に取ります。例えば、`'step 5'` のようになります。`step` コマンドは引数なしでも実行可能です。コマンドによっては、全く引数を受け付けないものもあります。

GDB コマンド名は省略可能です。ただし、省略された名前があいまいなものではあってはなりません。省略形は、それぞれのコマンドのドキュメント内に記載されています。場合によっては、あいまいな省略形も許されることがあります。例えば、`s` は、文字 `s` で始まるコマンドがほかにも存在するにもかかわらず、`step` コマンドの省略形として特別に定義されています。ある省略形が使用可能か否かは、それを `help` コマンドへの引数として使用することで判定可能です。

GDB への入力として空行を与える（`(RET)` キーだけを押す）ことは、1 つ前に実行したコマンドを繰り返すということを意味します。ただし、いくつかのコマンド（例えば、`run` コマンド）は、この方法で実行を繰り返すことはできません。意図に反して再実行してしまうと問題を引き起こす可能性があるため、繰り返し実行してほしくないようなコマンドの場合です。

`list` コマンドと `x` コマンドは、`(RET)` キーにより繰り返し実行すると、新たに引数が生成されて実行されるので、前回実行されたときと全く同様の状態で繰り返し実行されるわけではありません。こうすることで、ソース・コードの内容やメモリの内容を容易に調べることができます。

GDB は、別の用途でも `(RET)` キーを使用します。`more` ユーティリティと同様の方法で、長い出力を分割して表示する場合です（see Section 14.4 [Screen size], page 133）。このような場合、`(RET)` キーを余分に押してしまうことは往々にしてありえるので、GDB はこのような表示方法を使用しているコマンドについては、`(RET)` キーによる繰り返し実行を行いません。

テキストの中に `#` 記号があると、そこから行末まではコメントになります。コメントの部分は実行されません。これは、特にコマンド・ファイルの中で便利です（see Section 15.3 [Command files], page 139）。

### 3.2 コマンド名の補完

途中まで入力されたコマンド名は、それがあいまいでなければ、GDB が残りの部分を補完してくれます。また、いつでも、コマンド名の補完候補の一覧を表示してくれます。この機能は、GDB コマンド名、GDB サブ・コマンド名、ユーザ・プログラムのシンボル名に対して有効です。

GDB に単語の残りの部分を補完させたい場合には、`(TAB)` キーを押します。補完候補が 1 つしか存在しない場合、GDB は残りの部分を補完し、ユーザがコマンドを（`(RET)` キーを押すことで）完結させるのを待ちます。例えば、ユーザが以下のように入力したとしましょう。

```
(gdb) info bre (TAB)
```

GDB は `'breakpoints'` という単語の残りの部分を補完します。なぜなら、`info` コマンドのサブ・コマンドのうち、`'bre'` で始まるのはこの単語だけだからです。

```
(gdb) info breakpoints
```

この時点で、ユーザは`(RET)`キーを押して `info breakpoints` コマンドを実行するか、あるいは `'breakpoints'` コマンドが実行したいコマンドではなかった場合には、バックスペース・キーを押してこれを消去してから、他の文字を入力することができます (最初から `info breakpoints` コマンドを実行するつもりであれば、コマンド名補完機能ではなくコマンド名の省略形を利用して、`'info bre'` と入力した後、ただちに `(RET)` キーを押してもいいでしょう)。

`(TAB)` キーが押されたときに、2 つ以上の補完候補が存在する場合、GDB はベル音を鳴らします。さらにいくつか文字を入力してから補完を再度試みることも可能ですし、単に続けて `(TAB)` キーを押すことも可能です。後者の場合、GDB は補完候補の全一覧を表示します。例えば、`'make_'` で始まる名前を持つサブルーチンにブレイクポイントを設定したいような場合に、`b make_` まで入力して `(TAB)` キーを入力したところベル音が鳴ったとしましょう。ここで続けて `(TAB)` キーを入力すると、プログラム内の `'make_'` で始まるすべてのサブルーチン名が表示されます。例えば、以下のように入力したとします。

```
(gdb) b make_ (TAB)
```

ここで GDB はベル音を鳴らします。もう一度 `(TAB)` キーを入力すると、以下のように表示されます。

```
make_a_section_from_file      make_envron
make_abs_section              make_function_type
make_blockvector              make_pointer_type
make_cleanup                   make_reference_type
make_command                   make_symbol_completion_list
(gdb) b make_
```

補完候補を表示した後、ユーザが続きを入力できるよう、GDB は途中まで入力された文字列 (ここでは `'b make_'`) を再表示します。

最初から補完候補の一覧を表示したいのであれば、`(TAB)` キーを 2 回押す代わりに `M-?` を入力することもできます。ここで、`M-?` というのは `(META)` ? を意味します。これを入力するには、キーボード上に `(META)` シフト・キーとして指定されたキーがあれば、それを押しながら ? を入力します。`(META)` シフト・キーがない場合には、`(ESC)` キーを押した後、? を入力します。

ときには、入力したい文字列が、論理的には『単語』であっても、GDB が通常は単語の一部に含めない括弧のような文字を含む場合があります。このような場合に単語の補完機能を使用するためには、GDB コマンド内において、そのような単語を `'` (単一引用符) で囲みます。

このようなことが必要になる可能性が最も高いのは、C++ 関数名を入力するときでしょう。これは、C++ が関数のオーバーローディング (引数の型の違いによって識別される、同一の名前を持つ関数の複数の定義) をサポートしているからです。例えば、関数 `name` にブレイクポイントを設定する場合、それが `int` 型のパラメータを取る `name(int)` なのか、それとも `float` 型のパラメータを取る `name(float)` なのかをはっきりさせる必要があります。このような場合に単語の補完機能を使用するには、単一引用符 `'` を関数名の前に入力します。こうすることによって、`(TAB)` キーまたは `M-?` キーが押されて単語補完が要求されたときに、補完候補の決定には通常よりも多くのことを検討する必要があることが GDB に通知されます。

```
(gdb) b 'bubble( (M-?)
bubble(double,double)      bubble(int,int)
(gdb) b 'bubble(
```

場合によっては、名前の補完をするには引用符を使用する必要があるということを、GDB が自分で認識できることもあります。このような場合、ユーザが引用符を入力していなくても、GDB が (可能な限り補完を行いつつ) 引用符を挿入してくれます。



```
(gdb) b bub TAB
GDB は入力された 1 行を以下のように変更し、
ベル音を鳴らします。
(gdb) b 'bubble(
```

一般的には、オーバーロードされたシンボルに対して補完が要求された際に引数リストがまだ入力されていないと、GDB は、引用符が必要であると判断します（そして実際に挿入します）。

オーバーロードされた関数に関する情報については、see Section 9.4.1.3 [C++ expressions], page 82。コマンド `set overload-resolution off` を使用すれば、オーバーロードの解決を無効化することができます。see Section 9.4.1.7 [GDB features for C++], page 84。

### 3.3 ヘルプの表示

`help` コマンドを使うことで、GDB コマンドに関するヘルプ情報を GDB 自身に表示させることができます。

`help`

`h` `help` コマンド（省略形は `h`）を引数なしで実行することで、コマンドのクラス名の簡単な一覧を表示させることができます。

```
(gdb) help
List of classes of commands:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features

Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

`help class` 一般的なクラス名を引数に指定することで、そのクラスに属するコマンドの一覧を表示させることができます。`status` クラスを指定した場合の表示例を以下に示します。

```
(gdb) help status
Status inquiries.

List of commands:

show -- Generic command for showing things set
with "set"
```

```
info -- Generic command for printing status

Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

#### help command

helpの引数にコマンド名を指定することで、そのコマンドの使用法に関する簡単な説明が表示されます。

#### complete args

complete argsコマンドにコマンド名の先頭の部分を指定すると、コマンド名の補完候補の一覧を表示します。args には、補完されるべきコマンド名の先頭の文字列を指定します。例えば、

```
complete i
```

は、以下のような結果を表示します。

```
info
inspect
ignore
```

これは、GNU Emacs での使用を想定したものです。

helpコマンドに加えて、GDBのinfoコマンドおよびshowコマンドを使用することで、ユーザ・プログラムの状態やGDBの状態を問い合わせることができます。どちらのコマンドも、多くの観点からの問い合わせをサポートしています。このマニュアルでは、それぞれを適切と思われる箇所で紹介しています。索引のinfoやshowの部分に、それぞれのサブ・コマンドの紹介されているページが示されています。See [Index], page 175。

- |      |   |
|------|---|
| info | このコマンド（省略形はi）は、ユーザ・プログラムの状態を表わす情報を表示するものです。例えば、info argsによってユーザ・プログラムに与えられた引数を、info registersによって現在使用中のレジスタの一覧を、info breakpointsによってユーザが設定したブレイクポイントの一覧を、それぞれ表示することができます。help infoによって、infoコマンドのサブ・コマンドの完全な一覧が表示されます。   |
| set  | setコマンドによって、ある式の評価結果を環境変数に割り当てることができます。例えば、GDBのプロンプト文字列を\$記号に変更するには、set prompt \$を実行します。  |
| show | infoコマンドとは異なり、showコマンドはGDB自身の状態を表わす情報を表示するものです。showコマンドで表示可能な状態はすべて、対応するsetコマンドで変更可能です。例えば、数値の表示に使用する基数はset radixコマンドで制御できます。現在どの基数が使用されているかを単に知るためには、show radixコマンドを使用します。変更可能なすべてのパラメータとそれらの現在の値を表示するためには、showコマンドを引数なしで実行します。また、info setコマンドを使用することもできます。どちらのコマンドも、同じ情報を出力します。 |

以下に、対応するsetコマンドを持たないという意味で例外的である、3つのshowサブ・コマンドを示します。

#### show version

実行中のGDBのバージョンを表示します。GDBに関する障害レポートには、この情報を含める必要があります。もしも異なるバージョンのGDBを複数使用しているので

あれば、ときには現在実行している GDB のバージョンをはっきりさせたいこともあるでしょう。GDB のバージョンが上がるにつれ、新しいコマンドが導入され、古いコマンドはサポートされなくなるかもしれません。バージョン番号は、GDB の起動の際にも表示されます。

`show copying`

GDB のコピー作成許可に関する情報が表示されます。

`show warranty`

GNU の『無保証 ( NO WARRANTY )』声明文が表示されます。



## 4 GDB 配下でのプログラムの実行

プログラムを GDB 配下で実行するには、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。ユーザが選択した環境で、必要に応じて引数を指定して、GDB を起動することができます。プログラムの入力元と出力先をリダイレクトすること、既に実行中のプロセスをデバッグすること、子プロセスを終了させることもできます。

### 4.1 デバッグのためのコンパイル

プログラムを効率的にデバッグするためには、そのプログラムの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。このデバッグ情報はオブジェクト・ファイルに格納されます。この情報は、個々の変数や関数の型、ソース・コード内の行番号と実行形式コードのアドレスとの対応などを含みます。

デバッグ情報の生成を要求するには、コンパイラの実行時に `-g` オプションを指定します。

多くの C コンパイラでは、`-g` オプションと `-O` オプションを同時に指定することができません。このようなコンパイラでは、デバッグ情報付きの最適化された実行ファイルを生成することができません。

GNU の C コンパイラである GCC は、`-O` オプションの有無にかかわらず、`-g` オプションが指定できます。したがって、最適化されたコードをデバッグすることが可能です。プログラムをコンパイルするときには、常に `-g` オプションを指定することをお勧めします。自分のプログラムは正しいと思うかもしれませんが、自分の幸運を信じて疑わないというのは無意味なことです。

`-g -O` オプションを指定してコンパイルされたプログラムをデバッグするときには、オプティマイザがコードを再調整していることを忘れないでください。デバッグは、実際に存在するコードの情報を表示します。実行されるパスがソース・ファイルの記述と一致していなくても、あまり驚かないでください。これは極端な例ですが、定義されているが実際には使われていない変数を、GDB は認識しません。なぜなら、コンパイラの最適化処理により、そのような変数は削除されるからです。

命令スケジューリング機能を持つマシンなどでは、`-g` を指定してコンパイルされたプログラムでは正しく動作することが、`-g -O` を指定してコンパイルされたプログラムでは正しく動作しないということがあります。`-g -O` を指定してコンパイルされたプログラムのデバッグで何かおかしい点があれば、`-g` だけを指定してコンパイルしてみてください。これで問題が解決するようであれば、(再現環境と一緒に) 障害として私たちに報告してください。

古いバージョンの GNU C コンパイラは、デバッグ情報の生成のためのオプションの 1 つとして `-gg` をサポートしていました。現在の GDB はこのオプションをサポートしていません。お手元の GNU C コンパイラにこのオプションがあるようであれば、それは使わないでください。

### 4.2 ユーザ・プログラムの起動

run

r      GDB 配下でユーザ・プログラムの実行を開始するには `run` コマンドを使用してください。(VxWorks 以外の環境では) 最初にプログラム名を指定する必要があります。これには、GDB への引数を使用する方法 (see Chapter 2 [Getting In and Out of GDB], page 9) と、`file` コマンドまたは `exec-file` コマンドを使用する方法 (see Section 12.1 [Commands to specify files], page 99) とがあります。

プロセスをサポートする環境でプログラムを実行している場合、`run` コマンドは下位プロセスを生成し、そのプロセスにプログラムを実行させます（プロセスをサポートしていない環境では、`run` コマンドはプログラムの先頭アドレスにジャンプします）。

プログラムの実行は、上位プロセスから受け取る情報によって影響されます。GDB はこの情報を指定する手段を提供しています。これは、ユーザ・プログラムが起動される前に実行されていなければなりません（ユーザ・プログラムの実行後にその情報を変更することも可能ですが、その変更結果は、次にプログラムを実行したときに初めて有効になります）。この情報は、4 つに分類することができます。

**引数** ユーザ・プログラムに与える引数を、`run` コマンドへの引数として指定します。ターゲット上でシェルが使用可能であれば、引数を表現するのに通常使用する手法（例えば、ワイルドカード拡張や変数による代替など）が利用できるよう、シェルを経由して引数を渡します。UNIX システムでは、SHELL 環境変数によって、使用されるシェルを選択することができます。See Section 4.3 [Your program's arguments], page 20。

**環境** ユーザ・プログラムは通常、GDB の環境を継承します。GDB の `set environment` コマンドと `unset environment` コマンドを使用して、ユーザ・プログラムの実行に影響する環境の一部を変更することができます。See Section 4.4 [Your program's environment], page 21。

#### 作業ディレクトリ

ユーザ・プログラムは GDB の作業ディレクトリを継承します。GDB の作業ディレクトリは、GDB の `cd` コマンドで設定可能です。See Section 4.5 [Your program's working directory], page 22。

#### 標準入力、標準出力

ユーザ・プログラムは通常、GDB が標準入力、標準出力として使用しているのと同じのデバイスを、標準入力、標準出力として使用します。`run` コマンドのコマンド・ライン上で、標準入力、標準出力をリダイレクトすることも可能です。また、`tty` コマンドによって別のデバイスを割り当てることも可能です。See Section 4.6 [Your program's input and output], page 22。

注意：入出力のリダイレクトは機能しますが、デバッグ中のプログラムの出力を、パイプを使用して他のプログラムに渡すことはできません。このようなことをすると、GDB は誤って、別のプログラムのデバッグを開始してしまうでしょう。

`run` コマンドを実行すると、ユーザ・プログラムはすぐに実行を始めます。プログラムを停止させる方法については、See Chapter 5 [Stopping and continuing], page 29。プログラムが停止すると、`print` コマンドまたは `call` コマンドを使用して、プログラム内の関数を呼び出すことができます。See Chapter 8 [Examining Data], page 59。

GDB が最後にシンボル情報を読み込んだ後に、シンボル・ファイルの修正タイムスタンプが変更されている場合、GDB はシンボル・テーブルを破棄し再読み込みを行います。この場合、GDB は、その時点におけるブレイクポイントの設定を保持しようと試みます。

### 4.3 ユーザ・プログラムの引数

ユーザ・プログラムへの引数は、`run` コマンドへの引数によって指定可能です。それはまずシェルに渡され、ワイルドカードの展開や I/O のリダイレクトの後、プログラムに渡されます。SHELL 環境変数によって、GDB の使用するシェルが指定されます。SHELL 環境変数が定義されていないと、GDB は `/bin/sh` を使用します。

引数を指定せずに `run` コマンドを実行すると、前回 `run` コマンドを実行したときの引数、または、`set args` コマンドでセットされた引数が使われます。

`set args` ユーザ・プログラムが次に実行されるときに使用される引数を指定します。`set args` が引数なしで実行された場合、`run` コマンドは、ユーザ・プログラムを引数なしで実行します。一度プログラムに引数を指定して実行すると、次にプログラムを引数なしで実行する唯一の方法は、`run` コマンドを実行する前に `set args` コマンドを実行することです。

`show args` ユーザ・プログラムが実行されるときに渡される引数を表示します。

#### 4.4 ユーザ・プログラムの環境

環境とは、環境変数とその値の集合のことです。環境変数は、慣例として、ユーザ名、ユーザのホーム・ディレクトリ、端末タイプ、実行プログラムのサーチ・パスなどを記録します。通常、環境変数はシェル上で設定され、ユーザの実行するすべてのプログラムによって継承されます。デバッグ時には、GDB を終了・再起動せずに環境を変更して、ユーザ・プログラムを実行できると便利でしょう。

`path directory`

`directory` で指定されるディレクトリを環境変数 `PATH` (実行ファイルのサーチ・パス) の先頭に追加します。これは、GDB とユーザ・プログラムの両方に対して有効です。‘.’ (コロン) またはスペースで区切られた複数のディレクトリを指定することもできます。環境変数 `PATH` の中に既に `directory` が含まれている場合には、`directory` は環境変数 `PATH` の先頭に移動されます。これにより、`directory` はより早く検索されることになります。

文字列 ‘\$cwd’ によって、GDB がパスを検索する時点における作業ディレクトリを参照することができます。‘.’ (ピリオド) を使用すると、`path` コマンドを実行したディレクトリを参照することになります。`directory` 引数に ‘.’ (ピリオド) が含まれていると、GDB はまずそれを (カレント・ディレクトリに) 置き換えてから、サーチ・パスに追加します。

`show paths`

実行ファイルを検索するパスの一覧 (環境変数 `PATH` の値) を表示します。

`show environment [varname]`

ユーザ・プログラム起動時に渡される環境変数 `varname` の値を表示します。`varname` が指定されない場合は、プログラムに渡されるすべての環境変数の名前と値が表示されます。`environment` は `env` に省略可能です。

`set environment varname [=] value`

環境変数 `varname` の値として `value` をセットします。値の変更はユーザ・プログラムに対してのみ有効で、GDB に対しては無効です。`value` には任意の文字列が指定可能です。環境変数の値は単なる文字列であり、その解釈はユーザ・プログラムに委ねられています。`value` は必須パラメータではありません。省略された場合には、変数には空文字列がセットされます。

例えば、以下のコマンドは、後に UNIX プログラムが実行されるときにユーザ名として ‘foo’ をセットします (‘=’ の前後のスペースは見やすくするためのもので、実際には必要ありません)。

```
set env USER = foo
```

`unset environment varname`

ユーザ・プログラムに渡される環境から、環境変数 *varname* を削除します。これは、`'set env varname ='`とは異なります。`unset environment`は、環境変数の値として空文字列をセットするのではなく、環境変数そのものを環境から削除します。

注意: GDB は、環境変数 SHELLにより指定されるシェル(環境変数 SHELLが設定されていない場合には `/bin/sh`)を使用してプログラムを実行します。SHELL環境変数の指定するシェルが初期化ファイルを実行するものである場合(例えば、C-shellの`'.cshrc'`、BASHの`'.bashrc'`)、初期化ファイルの中で設定された環境変数はユーザ・プログラムに影響を与えます。環境変数の設定は、`'.login'`や`'.profile'`のように、ユーザがシステム内に入るときに実行されるファイルに移したほうがよいでしょう。

#### 4.5 ユーザ・プログラムの作業ディレクトリ

`run`コマンドで実行されるユーザ・プログラムは、実行時の GDB の作業ディレクトリを継承します。GDB の作業ディレクトリは、もともと親プロセス(通常はシェル)から継承したのですが、`cd` コマンドによって、GDB の中から新しい作業ディレクトリを指定することができます。

GDB の作業ディレクトリは、GDB によって操作されるファイルを指定するコマンドに対して、デフォルト・ディレクトリとして機能します。See Section 12.1 [Commands to specify files], page 99.

`cd directory`

GDB の作業ディレクトリを *directory* にします。

`pwd`

GDB の作業ディレクトリを表示します。

#### 4.6 ユーザ・プログラムの入出力

GDB 配下で実行されるプログラムは、デフォルトでは、GDB と同一の端末に対して入出力を行います。GDB は、ユーザとのやりとりのために、端末モードを GDB 用に変更します。このとき、ユーザ・プログラムが使用していた端末モードは記録され、ユーザ・プログラムを継続実行すると、そのモードに戻ります。

`info terminal`

ユーザ・プログラムが使用している端末モードに関して GDB が記録している情報を表示します。

`run`コマンドにおいてシェルのリダイレクト機能を使用することによって、ユーザ・プログラムの入出力をリダイレクトすることが可能です。例えば、

```
run > outfile
```

はユーザ・プログラムの実行を開始し、その出力をファイル `'outfile'`に書き込みます。

ユーザ・プログラムの入出力先を指定する別の方法に、`tty`コマンドがあります。このコマンドはファイル名を引数として取り、そのファイルを後に実行される `run`コマンドのデフォルトの入出力先とします。このコマンドはまた、後の `run`コマンドにより生成される子プロセスを制御する端末を変更します。例えば、

```
tty /dev/ttyb
```

は、それ以降に実行される `run`コマンドによって起動されるプロセスのデフォルトの入出力先および制御端末を `'/dev/ttyb'`端末とします。



runコマンド実行時に明示的にリダイレクト先を指定することで、ttyコマンドで指定された入力装置を変更することができますが、制御端末の設定は変更できません。

ttyコマンドを使用した場合も、runコマンドで入力をリダイレクトした場合も、ユーザ・プログラムの入力元だけが変更されます。これらのコマンドを実行しても、GDB の入力元は、ユーザの使用している端末のままです。

## 4.7 既に実行中のプロセスのデバッグ

### attach process-id

GDB の外で起動され、既に実行中のプロセスにアタッチします ( info files コマンドで、現在デバッグ対象となっているプログラムの情報が表示されます )。このコマンドは、プロセス ID を引数に取ります。UNIX プロセスのプロセス ID を知るのに通常使用する方法は、psユーティリティ、または、シェル・コマンドの 'jobs -l' の実行です。attachコマンドを実行後(RET)キーを押しても、コマンドは再実行されません。

attachコマンドを使用するには、プロセスをサポートする環境でユーザ・プログラムを実行する必要があります。例えば、オペレーティング・システムの存在しないボード・コンピュータのような環境で動作するプログラムに対して、attachコマンドを使うことはできません。さらに、ユーザは、プロセスに対してシグナルを送信する権利を持っている必要があります。

attachコマンドを使用すると、デバッグは、まずカレントな作業ディレクトリの中で、プロセスにより実行されているプログラムを見つけようとします。(プログラムが見つからなければ)次に、ソース・ファイルのサーチ・パス ( see Section 7.3 [Specifying source directories], page 55 ) を使用して、プログラムを見つけようとします。fileコマンドを使用して、プログラムをロードすることも可能です。See Section 12.1 [Commands to Specify Files], page 99。

指定されたプロセスをデバッグする準備が整った後に、GDB が最初にすることは、そのプロセスを停止することです。runコマンドを使用してプロセスを起動した場合は、通常使用可能なすべての GDB コマンドを使用して、アタッチされたプロセスの状態を調べたり変更したりすることができます。ブレイクポイントの設定、ステップ実行、継続実行、記憶域の内容の変更が可能です。プロセスの実行を継続したいのであれば、GDB がプロセスにアタッチした後に、continueコマンドを使用することができます。

detach      アタッチされたプロセスのデバッグが終了した場合には、detachコマンドを使用してそのプロセスを GDB の管理から解放することができます。プロセスからディタッチしても、そのプロセスは実行を継続します。detachコマンド実行後は、ディタッチされたプロセスと GDB は互いに完全に依存関係がなくなり、attachコマンドによる別のプロセスへのアタッチや、runコマンドによる別のプロセスの起動が可能になります。detachコマンドを実行後(RET)キーを押しても、detachコマンドは再実行されません。

プロセスがアタッチされている状態で、GDB を終了したり runコマンドを使用したりすると、アタッチされたプロセスを終了させてしまいます。デフォルトの状態では、このようなことを実行しようとする、GDB が確認を求めてきます。この確認処理を行うか否かは、set confirmコマンドで設定可能です ( see Section 14.6 [Optional warnings and messages], page 134 )。

## 4.8 子プロセスの終了

kill          GDB 配下で実行しているユーザ・プログラムのプロセスを終了させます。

このコマンドは、実行中のプロセスではなく、コア・ダンプをデバッグしたいときに便利です。GDB は、ユーザ・プログラムの実行中は、コア・ダンプ・ファイルを無視します。

いくつかのオペレーティング・システム上では、GDB の管理下でブレイクポイントを設定されている状態のプログラムを、GDB の外で実行することができません。このような場合、kill コマンドを使用することで、デバッガの外でのプログラムの実行が可能になります。

kill コマンドは、プログラムを再コンパイル、再リンクしたい場合にも便利です。というのは、多くのシステムでは、プロセスとして実行中の実行ファイルを更新することはできないからです。次に run コマンドを実行したときに、GDB は、実行ファイルが変更されていることを認識し、シンボル・テーブルを再度読み込みます (この際、その時点でのブレイクポイントの設定を維持しようと試みます)。

## 4.9 プロセス情報

いくつかのオペレーティング・システムは、`/proc` と呼ばれる便利な機能を提供しています。これは、ファイル・システム関連のサブルーチンを使用して、実行中プロセスのイメージを調べるのに使用することができます。GDB が、この機能を持つオペレーティング・システム用に構成されていれば、info proc コマンドを使用することで、ユーザ・プログラムを実行しているプロセスに関するいくつかの情報を知ることができます。info proc は、procfs をサポートする SVR4 システム上でのみ機能します。

info proc プロセスに関して入手可能な情報を要約して出力します。

info proc mappings

プログラムがアクセスすることのできるアドレス範囲を表示します。出力情報には、それぞれのアドレス範囲に対してユーザ・プログラムが持つ読み込み権、書き込み権、実行権の情報が含まれます。

info proc times

ユーザ・プログラムおよびその子 (プロセス) の起動時刻、ユーザ・レベルの CPU 消費時間、システム・レベルの CPU 消費時間を表示します。

info proc id

ユーザ・プログラムに関連のあるプロセスの ID 情報を表示します。ユーザ・プログラムのプロセス ID、親 (プロセス) のプロセス ID、プロセス・グループ ID、セッション ID を出力します。

info proc status

プロセスの状態に関する一般的な情報を出力します。プロセスが停止している場合は、停止した理由、( シグナルを受信した場合には ) 受信したシグナルが出力情報に含まれます。

info proc all

プロセスに関する上記の情報をすべて表示します。

## 4.10 マルチスレッド・プログラムのデバッグ

HP-UX や Solaris のようなオペレーティング・システムにおいては、1 つのプログラムが複数のスレッドを実行することができます。「スレッド」の正確な意味は、オペレーティング・システムによって異なります。しかし、一般的には、1 つのアドレス空間を共有するという点を除けば、プログラム内

のマルチスレッドは、マルチプロセスと類似しています（アドレス空間の共有とは、複数のスレッドが同一の変数の値を参照したり変更したりすることが可能であるということです）。その一方で、個々のスレッドは自分用のレジスタ、実行スタック、そしておそらくはプライベート・メモリを持ちます。

GDB は、マルチスレッド・プログラムのデバッグ用に、以下のような便利な機能を提供しています。

- 新規スレッド生成の自動的な通知
- スレッドを切り替えるコマンド `'thread threadno'`
- 既存のスレッドに関する情報を問い合わせるコマンド `'info threads'`
- 1 つのコマンドを複数のスレッドに対して実行するコマンド `'thread apply [threadno] [all] args'`
- スレッド固有のブレイクポイント

注意: これらの機能は、スレッドをサポートするオペレーティング・システム用に構成されたすべての GDB で使用可能なわけではありません。GDB がスレッドをサポートしていない環境では、これらのコマンドは無効です。例えば、スレッドをサポートしていないシステム上で GDB の `'info threads'` コマンドを実行しても何も表示されませんし、`thread` コマンドの実行は常に拒絶されます。

```
(gdb) info threads
(gdb) thread 1
Thread ID 1 not known. Use the "info threads" command to
see the IDs of currently known threads.
```

GDB のスレッド・デバッグ機能により、ユーザ・プログラムの実行中に、すべてのスレッドを観察することができます。ただし、GDB に制御権のある状態では、特定の 1 つのスレッドだけがデバッグの対象となります。このスレッドは、カレント・スレッドと呼ばれます。デバッグ用のコマンドは、カレント・スレッドの立場から見たプログラムの情報を表示します。

ユーザ・プログラム内部において新しいスレッドの存在を検出すると、GDB は、`'[New systag]'` という形式で、ターゲット・システム上におけるこのスレッドの ID を表示します。ここで `systag` とはスレッドの ID で、その形式はシステムによって異なります。例えば、LynxOS 上では、GDB が新しいスレッドを検出すると、

```
[New process 35 thread 27]
```

のように表示されます。一方、SGI のシステム上では、`systag` は単に `'process 368'` のような形式で、これ以外の情報は含まれません。

GDB は、ユーザ・プログラム内の個々のスレッドに対して、デバッグ用の整数値のスレッド番号を独自に割り当てます。

```
info threads
```

その時点においてユーザ・プログラム中に存在するすべてのスレッドに関する要約を表示します。個々のスレッドに関して、以下の情報が（列挙された順に）表示されます。

1. GDB により割り当てられたスレッド番号
2. ターゲット・システムのスレッド ID ( `systag` )
3. スレッドのカレントなスタック・フレームの要約

GDB により割り当てられたスレッド番号の左のアスタリスク '\*' は、そのスレッドがカレント・スレッドであることを意味しています。

以下に例を示します。

```
(gdb) info threads
3 process 35 thread 27  0x34e5 in sigpause ()
2 process 35 thread 23  0x34e5 in sigpause ()
* 1 process 35 thread 13  main (argc=1, argv=0x7fffffff8)
   at threadtest.c:68
```

`thread threadno`

スレッド番号 *threadno* を割り当てられたスレッドをカレント・スレッドとします。このコマンドの引数 *threadno* は、‘info threads’コマンドの出力の最初のフィールドに表示される、GDB 内部のスレッド番号です。GDB は、指定されたスレッドのシステム上の ID とカレントなスタック・フレームの要約を表示します。

```
(gdb) thread 2
[Switching to process 35 thread 23]
0x34e5 in sigpause ()
```

‘[New ...]’メッセージと同様、‘Switching to’の後ろに表示される情報の形式は、そのシステムにおけるスレッドの識別方法に依存します。

`thread apply [threadno] [all] args`

`thread apply` コマンドにより、1 つのコマンドを 1 つ以上のスレッドに対して実行することができます。実行対象となるスレッドのスレッド番号を、引数 *threadno* に指定します。*threadno* は、‘info threads’コマンドの出力の最初のフィールドに表示される、GDB 内部のスレッド番号です。すべてのスレッドに対してコマンドを実行するには、`thread apply all args` コマンドを使用してください。

GDB がユーザ・プログラムを停止させるとき、その理由がブレイクポイントであれシグナルの受信であれ、ブレイクポイントに到達したスレッド、または、シグナルを受信したスレッドが自動的に選択されます。GDB は、‘[Switching to systag]’という形式のメッセージでそのスレッドを示し、コンテキスト切り替えの発生に注意を促します。

複数スレッドを持つプログラムの停止時や起動時の GDB の動作の詳細については、See Section 5.4 [Stopping and starting multi-thread programs], page 45。

また、複数スレッドを持つプログラムの中におけるウォッチポイントについては、See Section 5.1.2 [Setting watchpoints], page 33。

## 4.11 マルチプロセス・プログラムのデバッグ

`fork` 関数を使用して新たにプロセスを生成するプログラムのデバッグに関しては、GDB は特別な機能を提供していません。プログラムが `fork` を実行するとき、GDB は引き続き親プロセスのデバッグを継続し、子プロセスは妨げられることなく実行を続けます。子プロセスが実行するコードにブレイクポイントを設定してあると、子プロセスは SIGTRAP シグナルを受信し、( そのシグナルをキャッチする処理がなければ ) 子プロセスは終了してしまいます。

しかし、子プロセスをデバッグしたい場合には、それほど困難ではない回避策があります。`fork` の呼び出し後に子プロセスが実行するソース・コードの中に、`sleep` 関数の呼び出しを加えてください。GDB に子プロセスのデバッグをさせる理由がないときに遅延が発生することのないように、特定の環境変数が設定されているときのみ、あるいは、特定のファイルが存在するときのみ、`sleep` 関数を呼び出すようにするとよいでしょう。子プロセスが `sleep` を呼び出している間に、`ps` コーティリティを使用して子プロセスのプロセス ID を獲得します。次に、GDB に対して ( 親プロセスもデバッグするのであれば、新たに GDB を起動して、その GDB に対して ) 子プロセスにアタッチするよう

指示してください ( see Section 4.7 [Attach], page 23 )。これ以降は、通常の方法でプロセスにアタッチした場合と全く同様に、子プロセスのデバッグが可能です。



## 5 停止と継続

デバッガを使用する主な目的は、プログラムが終了してしまう前に停止させたり、問題のあるプログラムを調査して何が悪いのかを調べたりすることにあります。

GDB 内部においてプログラムが停止する原因はいくつかあります。例えば、シグナルの受信、ブレイクポイントへの到達、stepコマンドのような GDB コマンドの実行後の新しい行への到達などです。プログラムが停止すると、変数の値の調査や設定、新しいブレイクポイントの設定、既存のブレイクポイントの削除などを行った後に、プログラムの実行を継続することができます。通常、GDB が表示するメッセージは、ユーザ・プログラムの状態について多くの情報を提供してくれます。ユーザはいつでも明示的にこれらの情報を要求することができます。

info program

ユーザ・プログラムの状態に関する情報を表示します。表示される情報は、そのプログラムの実行状態（実行中か否か）、そのプログラムのプロセス、プログラムが停止した理由です。

### 5.1 ブレイクポイント、ウォッチポイント、キャッチポイント

ブレイクポイントによって、プログラム内のある特定の箇所に到達するたびに、プログラムを停止することができます。個々のブレイクポイントについて、そのブレイクポイントにおいてプログラムを停止させるためには満足されなければならない、より詳細な条件を設定することができます。ブレイクポイントの設定は、いくつかある break コマンドのいずれかによって行います（see Section 5.1.1 [Setting breakpoints], page 30）。行番号、関数名、プログラム内における正確なアドレスを指定することで、プログラムのどこで停止するかを指定することができます。

HP-UX、SunOS 4.x、SVR4、Alpha OSF/1 上では、実行開始前に共用ライブラリ内にブレイクポイントを設定することもできます。HP-UX システムでは、ちょっとした制約があります。プログラムによって直接呼び出されるのではない共用ライブラリ・ルーチン（例えば、pthread\_create の呼び出しにおいて、引数として指定されるルーチン）にブレイクポイントをセットするためには、そのプログラムの実行が開始されるまで待たなければなりません。

ウォッチポイントは、ある式の値が変化したときにユーザ・プログラムを停止させる、特別なブレイクポイントです。ウォッチポイントは、他のブレイクポイントと同じように管理することができますが、設定だけは特別なコマンドで行います（see Section 5.1.2 [Setting watchpoints], page 33）。有効化、無効化、および削除を行うときに使用する各コマンドは、対象がブレイクポイントであってもウォッチポイントであっても同一です。

ブレイクポイントで GDB が停止するたびに、常に自動的にユーザ・プログラム内のある値を表示させるようにすることができます。See Section 8.6 [Automatic display], page 64。

キャッチポイントは、C++ の例外の発生やライブラリのローディングのようなある種のイベントが発生したときに、ユーザ・プログラムを停止させる、また別の特殊なブレイクポイントです。ウォッチポイントと同様、キャッチポイントを設定するために使用する特別なコマンドがあります。（see Section 5.1.3 [Setting catchpoints], page 34）。しかし、この点を除けば、キャッチポイントを他のブレイクポイントと同様に管理することができます。（ユーザ・プログラムがシグナルを受信したときに停止するようにするためには、handle コマンドを使用します。see Section 5.3 [Signals], page 43）。

ユーザが新規に作成した個々のブレイクポイント、ウォッチポイント、キャッチポイントに対して、GDB は番号を割り当てます。この番号は 1 から始まる連続する整数値です。ブレイクポイントの様々な側面を制御するコマンドの多くにおいて、変更を加えたいブレイクポイントを指定するのにこの番

号を使用します。個々のブレイクポイントを有効化、無効化することができます。無効化されたブレイクポイントは、再度有効化されるまで、ユーザ・プログラムの実行に影響を与えません。

### 5.1.1 ブレイクポイントの設定

ブレイクポイントは、`break`コマンド（省略形は `b`）によって設定されます。デバッガのコンビニエンス変数 `$bpnum` に、最後に設定されたブレイクポイントの番号が記録されます。コンビニエンス変数の使用方法については、See Section 8.9 [Convenience variables], page 71。

ブレイクポイントの設定箇所を指定する方法はいくつかあります。

#### `break function`

関数 *function* のエントリにブレイクポイントを設定します。ソース言語が（例えば C++ のように）シンボルのオーバーロード機能を持つ場合、*function* は、プログラムを停止させる可能性を持つ 1 つ以上の箇所を指すことがあります。このような状況に関する説明については、See Section 5.1.8 [Breakpoint menus], page 40。

#### `break +offset`

#### `break -offset`

その時点において選択されているフレームにおいて実行が停止している箇所から、指定された行数だけ先または手前にブレイクポイントを設定します。

#### `break linenum`

カレントなソース・ファイル内の *linenum* で指定される行番号を持つ行に、ブレイクポイントを設定します。ここで「カレントなソース・ファイル」とは、最後にソース・コードが表示されたファイルを指します。このブレイクポイントは、その行に対応するコードが実行される直前に、ユーザ・プログラムを停止させます。

#### `break filename:linenum`

*filename* で指定されるソース・ファイルの *linenum* で指定される番号の行に、ブレイクポイントを設定します。

#### `break filename:function`

*filename* で指定されるソース・ファイル内の *function* で指定される関数エントリにブレイクポイントを設定します。同じ名前の関数が複数のファイルに存在する場合以外は、ファイル名と関数名を同時に指定する必要はありません。

#### `break *address`

*address* で指定されるアドレスにブレイクポイントを設定します。これは、プログラムの中の、デバッグ情報やソース・ファイルが手に入らない部分にブレイクポイントを設定するのに使用できます。

#### `break`

引数なしで実行されると、`break`コマンドは、選択されたスタック・フレームにおいて次に実行される命令にブレイクポイントを設定します（see Chapter 6 [Examining the Stack], page 47）。最下位にあるスタック・フレーム以外のフレームが選択されていると、このブレイクポイントは、制御がそのフレームに戻ってきた時点で、ユーザ・プログラムを停止させます。これが持つ効果は、選択されたフレームの下位にあるフレームにおいて `finish` コマンドを実行するのと似ています。ただし、1 つ異なるのは、`finish` コマンドがアクティブなブレイクポイントを残さないという点です。最下位のスタック・フレームにおいて引数なしで `break` コマンドを実行した場合、そのときに停止していた箇所に次に到達したときに、GDB はユーザ・プログラムを停止させます。これは、ループの内部では便利でしょう。



GDBは通常、実行を再開したときに、最低でも1命令が実行されるまでの間は、ブレイクポイントの存在を無視します。そうでなければ、ブレイクポイントで停止した後、そのブレイクポイントを無効にしない限り、先へ進めないことになってしまいます。この規則は、ユーザ・プログラムが停止したときに、既にそのブレイクポイントが存在したか否かにかかわらず、適用されます。

`break ... if cond`

*cond* で指定される条件式付きでブレイクポイントを設定します。そのブレイクポイントに達すると、必ず条件式 *cond* が評価されます。評価結果がゼロでない場合、すなわち、評価結果が真である場合のみ、ユーザ・プログラムを停止します。‘...’の部分には、これまでに説明してきた停止箇所を指定するための引数のいずれかが入ります(‘...’は省略も可能です)。ブレイクポイントの条件式の詳細については、See Section 5.1.6 [Break conditions], page 37。

`tbreak args`

プログラムを1回だけ停止させるブレイクポイントを設定します。*args* の部分は `break` コマンドと同様であり、ブレイクポイントも同じように設定されますが、`tbreak`により設定されたブレイクポイントは、プログラムが最初にそこで停止した後に自動的に削除されます。See Section 5.1.5 [Disabling breakpoints], page 36。

`hbreak args`

ハードウェアの持つ機能を利用したブレイクポイントを設定します。*args* の部分は `break` コマンドと同様であり、ブレイクポイントも同じように設定されますが、`hbreak`により設定されるブレイクポイントは、ハードウェアによるサポートを必要とします。ターゲット・ハードウェアによっては、このような機能を持たないものもあるでしょう。これの主な目的は、EPROM/ROM コードのデバッグであり、ユーザはある命令にブレイクポイントを設定するのに、その命令を変更する必要がありません。これは、SPARClike DSU の提供するトラップ発生機能と組み合わせ使用することができます。DSU は、デバッグ・レジスタに割り当てられたデータ・アドレスまたは命令アドレスをプログラムがアクセスすると、トラップを発生させます。ハードウェアの提供するブレイクポイント・レジスタは、データ・ブレイクポイントを2つまでしか取れないので、3つ以上使用しようとする、GDBはそれを拒絶します。このような場合、不要になったハードウェア・ブレイクポイントを削除または無効化してから、新しいハードウェア・ブレイクポイントを設定してください。See Section 5.1.6 [Break conditions], page 37。

`thbreak args`

ハードウェアの機能を利用して、プログラムを1回だけ停止させるブレイクポイントを設定します。*args* の部分は `hbreak` コマンドと同様であり、ブレイクポイントも同じように設定されます。しかし、`tbreak` コマンドの場合と同様、最初にプログラムがそこで停止した後に、このブレイクポイントは自動的に削除されます。また、`hbreak` コマンドの場合と同様、このブレイクポイントはハードウェアによるサポートを必要とするものであり、ターゲット・ハードウェアによっては、そのような機能がないこともあるでしょう。See Section 5.1.5 [Disabling breakpoints], page 36。また、See Section 5.1.6 [Break conditions], page 37。

`rbreak regex`

*regex* で指定される正規表現にマッチするすべての関数にブレイクポイントを設定します。このコマンドは、正規表現にマッチしたすべての関数に無条件ブレイクポイントを設定し、設定されたすべてのブレイクポイントの一覧を表示します。設定されたブレイクポイントは、`break` コマンドで設定されたブレイクポイントと同様に扱われます。他

のすべてのブレイクポイントと同様の方法で、削除、無効化、および条件の設定が可能です。

C++プログラムのデバッグにおいて、あるオーバーロードされたメンバ関数が、特別なクラスだけが持つメンバ関数というわけではない場合、そのメンバ関数にブレイクポイントを設定するのに、`rbreak`コマンドは便利です。

```
info breakpoints [n]
info break [n]
info watchpoints [n]
```

設定された後、削除されていない、すべてのブレイクポイント、ウォッチポイント、キャッチポイントの一覧を表示します。個々のブレイクポイントについて、以下の情報が表示されます。

ブレイクポイント番号

タイプ      ブレイクポイント、ウォッチポイント、または、キャッチポイント

廃棄          ブレイクポイントに次に到達したときに、無効化または削除されるべくマークされているか否かを示します。

有効 / 無効   有効なブレイクポイントを ‘y’、有効でないブレイクポイントを ‘n’ で示します。

アドレス     ユーザ・プログラム内のブレイクポイントの位置をメモリ・アドレスとして示します。

対象          ユーザ・プログラムのソース内におけるブレイクポイントの位置を、ファイル名および行番号で示します。

ブレイクポイントが条件付きのものである場合、`info break`コマンドは、そのブレイクポイントに関する情報の次の行に、その条件を表示します。ブレイクポイント・コマンドがあれば、続いてそれが表示されます。

`info break`コマンドに引数としてブレイクポイント番号 *n* が指定されると、その番号に対応するブレイクポイントだけが表示されます。コンビニエンス変数 `$_`、および、`x` コマンドのデフォルトの参照アドレスには、一覧の中で最後に表示されたブレイクポイントのアドレスが設定されます ( see Section 8.5 [Examining memory], page 63 )。

`info break`コマンドは、ブレイクポイントに到達した回数を表示します。これは、`ignore`コマンドと組み合わせると便利です。まず、`ignore`コマンドによってブレイクポイントへの到達をかなりの回数無視するよう設定します。プログラムを実行し、`info break`コマンドの出力結果から何回ブレイクポイントに到達したかを調べます。再度プログラムを実行し、今度は前回の実行時に到達した回数より 1 だけ少ない回数だけ無視するように設定します。こうすることで、前回の実行時にそのブレイクポイントに最後に到達したときと同じ状態でプログラムを停止させることが簡単にできます。

GDB では、ユーザ・プログラム内の同一箇所にも何度でもブレイクポイントを設定することができます。これは、くだらないことでも、無意味なことでもありません。設定されるブレイクポイントが条件付きのものである場合、これはむしろ有用です ( see Section 5.1.6 [Break conditions], page 37 )。

GDB 自身が、特別な目的でユーザ・プログラム内部にブレイクポイントを設定することがあります。例えば、( C プログラムにおける ) `longjmp` を適切に処理するためなどです。これらの内部的なブレイクポイントには -1 から始まる負の番号が割り当てられます。‘`info breakpoints`’ コマンドは、このようなブレイクポイントを表示しません。

これらのブレイクポイントは、GDB の保守コマンド ‘maint info breakpoints’ で表示することができます。

maint info breakpoints

‘info breakpoints’ コマンドと同様の形式で呼び出され、ユーザが明示的に設定したブレイクポイントと、GDB が内部的な目的で使用しているブレイクポイントの両方を表示します。内部的なブレイクポイントは、負のブレイクポイント番号で示されます。タイプ欄にブレイクポイントの種類が表示されます。

breakpoint

明示的に設定された普通のブレイクポイント

watchpoint

明示的に設定された普通のウォッチポイント

longjmp

longjmp が呼び出されたときに正しくステップ処理ができるように、内部的に設定されたブレイクポイント

longjmp resume

longjmp のターゲットとなる箇所に内部的に設定されたブレイクポイント

until

GDB の until コマンドで一時的に使用される内部的なブレイクポイント

finish

GDB の finish コマンドで一時的に使用される内部的なブレイクポイント

### 5.1.2 ウォッチポイントの設定

ウォッチポイントを設定することで、ある式の値が変化したときに、プログラムの実行を停止させることができます。その値の変更が、プログラムのどの部分で行われるかをあらかじめ知っている必要はありません。

システムによって、ウォッチポイントがソフトウェアによって実装されていることもあれば、ハードウェアによって実装されていることもあります。GDB は、ユーザ・プログラムをシングル・ステップ実行して、そのたびに変数の値をテストすることによって、ソフトウェア・ウォッチポイントを実現しています。これは、通常の実行と比較すると、何百倍も遅くなります。(それでも、プログラムのどの部分が問題を発生させたのか全く手掛りのない誤りを見つけることができるのであれば、十分価値のあることかもしれません)。

HP-UX や Linux のようなシステム上の GDB には、ハードウェア・ウォッチポイントのサポートも組み込まれています。これを使用すれば、ユーザ・プログラムの実行が遅くなることはありません。

watch expr

expr で指定される式に対してウォッチポイントを設定します。プログラムが式の値を書き換えるときに、GDB はプログラムの実行を停止させます。

rwatch expr

expr で指定される対象が読み込みアクセスされるときにプログラムを停止させるウォッチポイントを設定します。2 つめのウォッチポイントとして設定するのであれば、1 つめのウォッチポイントも rwatch コマンドで設定されていなければなりません。

awatch expr

expr で指定される対象が読み込みアクセス、書き込みアクセスされるときにプログラムを停止させるウォッチポイントを設定します。2 つめのウォッチポイントとして設定するのであれば、1 つめのウォッチポイントも awatch コマンドで設定されていなければなりません。

`info watchpoints`

ウォッチポイント、ブレイクポイント、キャッチポイントの一覧を表示します。これは、`info break`と同じです。

GDB は、可能であれば、ハードウェア・ウォッチポイントを設定します。ハードウェア・ウォッチポイントをセットした場合は高速な実行が可能であり、デバッガは、変更を引き起こした命令のところで、値の変更を報告することができます。ハードウェア・ウォッチポイントを設定できない場合、GDB は、ソフトウェア・ウォッチポイントを設定します。これは、実行速度も遅く、値の変更は、その変更が実際に発生した後に、その変更を引き起こした命令のところではなく、1 つ後ろの文のところで報告されます。

`watch` コマンドを実行すると、ハードウェア・ウォッチポイントの設定が可能な場合には、GDB は、以下のような報告を行います。

Hardware watchpoint *num*: *expr*

SPARClike DSU は、デバッグ・レジスタに割り当てられたデータ・アドレスや命令アドレスにプログラムがアクセスすると、トラップを発生させます。データ・アドレスについては、DSU が `watch` コマンドを支援しています。しかし、ハードウェアの提供するブレイクポイント・レジスタは、データ・ウォッチポイントを 2 つまでしか取れず、その 2 つは同じ種類のウォッチポイントでなければなりません。例えば、2 つのウォッチポイントを、両方とも `watch` コマンドで設定すること、両方とも `rwatch` コマンドで設定すること、あるいは、両方とも `awatch` コマンドで設定することは可能ですが、それぞれを異なるコマンドで設定することはできません。異なる種類のウォッチポイントを同時に設定しようとしても、コマンドの実行を GDB が拒否します。このような場合、使用しないウォッチポイント・コマンドを削除または無効化してから、新しいウォッチポイント・コマンドを設定してください。

`print` や `call` を使用して関数を対話的に呼び出すと、それまでにセットされていたウォッチポイントはいずれも、GDB が別の種類のブレイクポイントに到達するか、あるいは、関数の呼び出しが終了するまでの間は、効果を持たなくなります。

注意: マルチスレッド・プログラムでは、ウォッチポイントの有用性は限定されます。現在のウォッチポイントの実装では、GDB は、単ースレッドの中でしか式の値を監視することができません。カレント・スレッドの処理の結果としてのみ、その式の値が変更されること（かつ、他のスレッドがカレント・スレッドにはならないこと）が確実にあれば、通常どおり、ウォッチポイントを使用することができます。しかし、カレント・スレッド以外のスレッドが式の値を変更することがあると、GDB は、その変更気付かないかもしれません。

### 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によって、C++ 例外や共用ライブラリのローディングのような、ある種のプログラム・イベントが発生したときに、デバッガを停止させることができます。キャッチポイントを設定するには、`catch` コマンドを使用します。

`catch event`

*event* で指定されるイベントが発生したときに停止します。*event* は、以下のいずれかです。

`throw`      C++ 例外の発生。

`catch`      C++ 例外のキャッチ。

`exec`      `exec` の呼び出し。現在これは、HP-UX においてのみ利用可能です。

```

fork      forkの呼び出し。現在これは、HP-UX においてのみ利用可能です。
vfork     vforkの呼び出し。現在これは、HP-UX においてのみ利用可能です。

load
load libname
          任意の共用ライブラリの動的なローディング、あるいは、libname で指定
          されるライブラリのローディング。現在これは、HP-UX においてのみ利用
          可能です。

unload
unload libname
          動的にロードされた任意の共用ライブラリのアンローディング、あるいは、
          libname で指定されるライブラリのアンローディング。現在これは、HP-
          UX においてのみ利用可能です。

```

#### tcatch event

1 回だけ停止させるキャッチポイントを設定します。最初にイベントが捕捉された後に、キャッチポイントは自動的に削除されます。

カレントなキャッチポイントの一覧を表示するには、`info break` コマンドを使用します。

現在、GDB における C++ の例外処理 ( `catch throw` と `catch catch` ) にはいくつかの制限があります。

- 関数に対話的に呼び出すと、GDB は通常、その関数が実行を終了したときに、ユーザに制御を戻します。しかし、その関数呼び出しが例外を発生させると、ユーザへ制御を戻すメカニズムが実行されないことがあります。この場合、ユーザ・プログラムは、アボートするか、あるいは、ブレイクポイントへの到達、GDB が監視しているシグナルの受信、そのプログラム自体の終了などのイベントが発生するまで、継続実行されることになります。これは、例外に対するキャッチポイントを設定してある場合にもあてはまります。対話的な関数呼び出しの間は、例外に対するキャッチポイントは無効化されています。
- 対話的に例外を発生させることはできません。
- 対話的に例外ハンドラを組み込むことはできません。

`catch` コマンドが、例外処理をデバッグする手段としては最適なものではないような場合もあります。どこで例外が発生したのかを正確に知りたい場合、例外ハンドラが呼び出される前にプログラムを停止させた方がよいでしょう。なぜなら、スタック・ポインタの調整が行われる前のスタックの状態を見ることができるからです。例外ハンドラの内部にブレイクポイントを設定してしまうと、どこで例外が発生したのかを調べるのは簡単ではないでしょう。

例外ハンドラが呼び出される直前で停止させるには、実装に関する知識が若干必要になります。GNU C++ の場合、以下のような ANSI C インターフェイスを持つ `__raise_exception` というライブラリ関数を呼び出すことで例外を発生させます。

```

/* addr は例外識別子が格納される領域
   id は例外識別子 */
void __raise_exception (void **addr, void *id);

```

スタック・ポインタの調整が行われる前に、すべての例外をデバッガにキャッチさせるには、`__raise_exception` にブレイクポイントを設定します ( see Section 5.1 [Breakpoints; watchpoints; and exceptions], page 29 )

*id* の値に依存する条件を付けたブレイクポイント ( see Section 5.1.6 [Conditions], page 37 ) を使用することで、特定の例外が発生したときにだけユーザ・プログラムを停止させることができます。

す。複数の条件付きブレイクポイントを設定することで、複数の例外の中のどれかが発生したときにユーザ・プログラムを停止させることもできます。

#### 5.1.4 ブレイクポイントの削除

ブレイクポイント、ウォッチポイント、キャッチポイントがプログラムを 1 回停止させた後、同じところで再びプログラムを停止させたくない場合、それらを取り除くことがしばしば必要になります。これが、ブレイクポイントの削除と呼ばれるものです。削除されたブレイクポイントはもはや存在しなくなり、それが存在したという記録も残りません。

`clear` コマンドを使用する場合、ブレイクポイントを、それがプログラム内部のどこに存在するかを指定することによって削除します。`delete` コマンドの場合は、ブレイクポイント番号を指定することで、個々のブレイクポイント、ウォッチポイント、キャッチポイントを削除することができます。

ブレイクポイントで停止した後、先へ進むために、そのブレイクポイントを削除する必要はありません。ユーザが実行アドレスを変更することなく継続実行する場合、最初に行われる命令に設定されているブレイクポイントを、GDB は自動的に無視します。

`clear`        選択されているスタック・フレーム内において次に実行される命令に設定されているブレイクポイントを削除します ( see Section 6.3 [Selecting a frame], page 49 )。最下位にあるフレームが選択されている場合、ユーザ・プログラムが停止した箇所に設定されているブレイクポイントを削除するのに便利な方法です。

`clear function`

`clear filename: function`

`function` で指定される関数のエントリに設定されているブレイクポイントを削除します。

`clear linenum`

`clear filename: linenum`

指定された行、または、その行内に記述されたコードに設定されたブレイクポイントを削除します。

`delete [breakpoints] [bnums...]`

引数で指定された番号を持つブレイクポイント、ウォッチポイント、キャッチポイントを削除します。引数が指定されない場合、すべてのブレイクポイントを削除します ( `set confirm off` コマンドが事前に実行されていない場合、GDB は、削除してもよいかどうか確認を求めてきます )。このコマンドの省略形は `d` です。

#### 5.1.5 ブレイクポイントの無効化

ブレイクポイント、ウォッチポイント、キャッチポイントを削除するのではなく、無効化したい場合もあるでしょう。無効化によって、ブレイクポイントは、それがあたかも削除されたかのように機能しなくなりますが、後に再度有効化することができるよう、そのブレイクポイントに関する情報は記憶されます。

ブレイクポイント、ウォッチポイント、キャッチポイントは、`enable` コマンドと `disable` コマンドによって有効化、無効化されます。これらのコマンドには、引数として 1 つ以上のブレイクポイント番号を指定することも可能です。指定すべき番号が分からない場合は、`info break` コマンド、または、`info watch` コマンドによってブレイクポイント、ウォッチポイント、キャッチポイントの一覧を表示させてください。

ブレイクポイント、ウォッチポイント、キャッチポイントは、有効 / 無効という観点から見て、4 つの異なる状態を持つことができます。

- 有効。ブレイクポイントはユーザ・プログラムを停止させます。breakコマンドで設定されたブレイクポイントの初期状態はこの状態です。
- 無効。ブレイクポイントはユーザ・プログラムの実行に影響を与えません。
- 1 回有効。ブレイクポイントはユーザ・プログラムを停止させますが、停止後、そのブレイクポイントは無効状態になります。tbreakコマンドで設定されたブレイクポイントの初期状態はこの状態です。
- 1 回有効 ( 削除 )。ブレイクポイントはユーザ・プログラムを停止させますが、停止直後に、そのブレイクポイントは完全に削除されます。

以下のコマンドを使用することで、ブレイクポイント、ウォッチポイント、キャッチポイントの有効化、無効化が可能です。

`disable [breakpoints] [bnums...]`

指定されたブレイクポイントを無効化します。番号が 1 つも指定されない場合は、すべてのブレイクポイントが無効化されます。無効化されたブレイクポイントは何ら影響力を持ちませんが、そのブレイクポイントに関する情報まで削除されるわけではありません。そのブレイクポイントを無視する回数、ブレイクポイント成立の条件、ブレイクポイント・コマンドなどのオプションは、後にそのブレイクポイントが有効化される場合に備えて、記憶されています。disable コマンドは `dis` と省略することができます。

`enable [breakpoints] [bnums...]`

指定されたブレイクポイント ( または、すべての定義済みブレイクポイント ) を有効化します。有効化されたブレイクポイントは、再びユーザ・プログラムを停止させることができるようになります。

`enable [breakpoints] once bnums...`

指定されたブレイクポイントを一時的に有効化します。このコマンドで有効化されたブレイクポイントはどれも、最初にプログラムを停止させた直後に、GDB によって無効化されます。

`enable [breakpoints] delete bnums...`

1 回だけプログラムを停止させ、その直後に削除されるような設定で、指定されたブレイクポイントを有効化します。このコマンドで有効化されたブレイクポイントはどれも、最初にプログラムを停止させた直後に、GDB によって削除されます。

tbreak コマンド ( see Section 5.1.1 [Setting breakpoints], page 30 ) で設定されたブレイクポイントを除き、ユーザによって設定されたブレイクポイントの初期状態は有効状態です。その後、ユーザが上記のコマンドのいずれかを使用した場合に限り、無効化されたり有効化されたりします ( `until` コマンドは、独自にブレイクポイントを設定、削除することができますが、ユーザの設定した他のブレイクポイントの状態は変更しません。see Section 5.2 [Continuing and stepping], page 41 )。

### 5.1.6 ブレイクポイントの成立条件

最も単純なブレイクポイントは、指定された箇所にプログラムが到達するたびに、プログラムの実行を停止させます。ブレイクポイントに対して条件を指定することも可能です。ここで、「条件」とは、プログラムが記述された言語で表現された真偽値を表す式のことです ( see Section 8.1 [Expressions], page 59 )。条件付きのブレイクポイントにプログラムが到達するたびに、その式が評価されます。そして、その結果が真であった場合だけ、プログラムは停止します。

これは、プログラムの正当性を検査するために診断式を使用するのとは逆になります。診断式の場合は、成立しないとき、すなわち条件が偽であるときに、プログラムを停止させます。C 言語で `assert`

という診断式をテストするためには、しかるべきブレイクポイントに `! assert` という条件を設定します。

ウォッチポイントに対して条件を設定することもできます。もともとウォッチポイントは、ある式の値を検査するものですから、これは必要ないかもしれませんが。しかし、ある変数の新しい値がある特定の値に等しいか否かを検査するのは条件式のほうに任せて、ウォッチポイントの対象そのものは単にその変数の名前にしてしまうという設定の方が簡単でしょう。

ブレイクポイントの成立条件に副作用を持たせたり、場合によってはプログラム内部の関数を呼び出させたりすることもできます。プログラムの進行状況をログに取る関数を呼び出したり、特別なデータ構造をフォーマットして表示するユーザ定義の関数を使用したい場合などに便利です。この効果は、同じアドレスに有効なブレイクポイントが別に設定されていない限り、完全に予測可能です（別のブレイクポイントが設定されていると、GDBはこのブレイクポイントを先に検出し、他のブレイクポイントで設定した条件式をチェックすることなくプログラムを停止させてしまうかもしれません）。あるブレイクポイントに到達したときに、副作用を持つ処理を実行させるためには、ブレイクポイント・コマンドの方がより便利であり、より柔軟でしょう（see Section 5.1.7 [Breakpoint command lists], page 39）。

ブレイクポイントの成立条件は、ブレイクポイントを設定する際に、`break` コマンドの引数に `if` を使用することによって、設定できます。See Section 5.1.1 [Setting breakpoints], page 30。ブレイクポイントの成立条件は、`condition` コマンドによっていつでも変更できます。`watch` コマンドは、`if` キーワードを認識しません。ウォッチポイントに対して条件を追加設定する唯一の方法は、`condition` コマンドを使うことです。

`condition bnum expression`

`bnum` で指定される番号のブレイクポイント、ウォッチポイント、キャッチポイントの成立条件として、`expression` を指定します。条件を設定した後、番号 `bnum` のブレイクポイントは、`expression` の値が真（C 言語の場合はゼロ以外の値）であるときのみ、ユーザ・プログラムを停止させます。`condition` コマンドを使用すると、GDB はただちに `expression` の構文の正当性、および、`expression` の中で使用されるシンボル参照の、ブレイクポイントのコンテキストにおける有効性をチェックします。しかし、`condition` コマンドが実行されるときに、`expression` の値が GDB によって実際に評価されるわけではありません。See Section 8.1 [Expressions], page 59。

`condition bnum`

`bnum` で指定される番号のブレイクポイントから条件を削除します。実行後、それは通常の無条件ブレイクポイントになります。

ブレイクポイント成立条件の特別なものに、ブレイクポイントに到達した回数がある数に達したときにプログラムを停止させるというものがあります。これは大変便利なので、それを実現するための特別な方法が提供されています。それは、ブレイクポイントの通過カウント（`ignore count`）を使用する方法です。すべてのブレイクポイントは、通過カウントと呼ばれる整数値を持っています。ほとんどの場合、この通過カウントの値はゼロであり、何ら影響力を持ちません。しかし、通過カウントとして正の値を持つブレイクポイントに到達すると、ユーザ・プログラムはそこで停止せず、単に通過カウントの値を 1 減少させて処理を継続します。したがって、通過カウントが `n` であると、ユーザ・プログラムがそのブレイクポイントに到達した回数が `n` 以下の間は、そのブレイクポイントにおいてプログラムは停止しません。

`ignore bnum count`

`bnum` で指定される番号のブレイクポイントの通過カウントを `count` で指定される値に設定します。ブレイクポイントへの到達回数が `count` 以下の間、ユーザ・プログラムは



停止しません。この間、GDB は、通過カウン트의値を 1 減少させる以外には何もしません。

次にブレイクポイントに到達したときにプログラムを停止させるには、*count* にゼロを指定してください。

ブレイクポイントで停止した後に *continue* コマンドを使用して実行を再開する場合、*ignore* コマンドを使用することなく、直接 *continue* コマンドの引数に通過カウンートを指定することができます。See Section 5.2 [Continuing and stepping], page 41。

ブレイクポイントが通過カウンートとして正の値を持ち、かつ、成立条件を持つ場合、成立条件はチェックされません。通過カウンートが 0 に達すると、GDB は成立条件のチェックを再開します。

‘\$foo-- <= 0’ のように、評価のたびに値の減少するコンビニエンス変数を使用した評価式によって、通過カウンートと同様の効果を達成することができます。See Section 8.9 [Convenience variables], page 71。

通過カウンートは、ブレイクポイント、ウォッチポイント、キャッチポイントに適用されます。

### 5.1.7 ブレイクポイント・コマンド・リスト

ブレイクポイント（あるいは、ウォッチポイント、キャッチポイント）に対して、それによってプログラムが停止したときに実行される一連のコマンドを指定することができます。例えば、ある特定の式の値を表示したり、他のブレイクポイントを有効化したりできると便利なこともあるでしょう。

```
commands [bnum]
... command-list ...
end
```

*bnum* で指定される番号を持つブレイクポイントに対して一連のコマンドを指定します。コマンド自体は、次の行以下に記述します。コマンドの記述を終了するには、*end* だけから成る 1 行を記述します。

ブレイクポイントからすべてのコマンドを削除するには、*commands* 行に続いて（コマンドを 1 つも指定せずに）*end* を記述します。

引数 *bnum* が指定されない場合、*commands* は、最後に設定されたブレイクポイント、ウォッチポイント、キャッチポイントを対象とします（最後に到達したブレイクポイントではありません）。

*command-list* の記述中は、**RET** キーを持つ、最後に実行されたコマンドを繰り返し実行する機能は無効です。

ブレイクポイント・コマンドを使用してプログラムの実行を再開することができます。*continue*、*step*、または、実行を再開させるその他の任意のコマンドを使用してください。

コマンド・リストの中で、実行を再開するコマンドの後に記述されているものは無視されます。というのは、プログラムが実行を再開すると（たとえそれが *next* コマンドや *step* コマンドによるものであっても）別のブレイクポイントに到達する可能性があり、そのブレイクポイントがコマンド・リストを持っていると、どちらのリストを実行するべきかあいまいになるからです。

コマンド・リストの先頭に指定されたコマンドが *silent* であると、ブレイクポイントで停止したときに通常出力されるメッセージは表示されません。これは、ある特定のメッセージを出力して実行を継続するようなブレイクポイントを設定するのに望ましいでしょう。コマンド・リスト中の後続のコマンドがどれもメッセージを出力しない場合、ブレイクポイントに到達したことをユーザに示す情報は何も表示されないこととなります。*silent* はブレイクポイント・コマンド・リストの先頭においてのみ意味を持ちます。

echo、output、printfの各コマンドを使用することで、細かく管理された出力を表示することができます。これらのコマンドは、silent指定のブレイクポイントで使うと便利です。See Section 15.4 [Commands for controlled output], page 139。

例えば、ブレイクポイント・コマンドを使用して、fooへのエントリにおいて xが正の値を持つときに、その値を表示するには以下のようにします。

```
break foo if x>0
commands
silent
printf "x is %d\n",x
cont
end
```

ブレイクポイント・コマンドの1つの応用として、あるバグの持つ影響を取り除いて、他のバグを見つけるためにテストを継続することができます。誤りのある行の次の行にブレイクポイントを設定し、その条件の中で誤りの発生を検査し、ブレイクポイント・コマンドの中で修正の必要な変数に正しい値を割り当てます。コマンド・リストの最後には continueコマンドを記述して、プログラムが停止しないようにします。また、プログラムの先頭には silentコマンドを記述し、何も出力されないようにします。以下に例を挙げます。

```
break 403
commands
silent
set x = y + 4
cont
end
```

### 5.1.8 ブレイクポイント・メニュー

プログラミング言語によっては（特に C++の場合）、異なるコンテキストにおいて使用するために、同一の関数名を複数回定義することが可能です。これは、オーバーローディングと呼ばれます。関数名がオーバーロードされている場合、‘break function’だけでは、どこにブレイクポイントを設定したいのかを GDB に正しく指定するのに十分ではありません。このような場合には、ブレイクポイントを設定したい関数がどれであることを正確に指定するために、‘break function(types)’のような形式を使用することができます。このような形式を使用しないと、GDB は候補となりえるブレイクポイントの一覧を番号付きのメニューとして表示し、プロンプト ‘>’によってユーザの選択を待ちます。先頭の2つの選択肢は常に、‘[0] cancel’と‘[1] all’です。1を入力すると、候補となるすべての関数のそれぞれの定義に対してブレイクポイントを設定します。また、0を入力すると、新たにブレイクポイントを設定することなく breakコマンドを終了します。

例えば、以下に示すセッションの抜粋は、オーバーロードされたシンボル String::afterに対してブレイクポイントを設定しようとした場合を示しています。ここでは、この関数名を持つ関数定義の中から3つを選択しています。

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
breakpoints.
(gdb)
```

## 5.2 継続実行とステップ実行

継続実行とは、ユーザ・プログラムの実行を再開して、それが正常に終了するまで実行させることを指します。一方、ステップ実行とは、ユーザ・プログラムを1「ステップ」だけ実行することを指します。ここで「ステップ」とは、(使用されるコマンドによって)1行のソース・コードを指すこともありますし、1マシン命令を指すこともあります。継続実行の場合でもステップ実行の場合でも、ブレークポイントやシグナルが原因となって、正常終了する前にユーザ・プログラムが停止することがあります(シグナルによってプログラムが停止した場合、実行を再開するには `handle` コマンドまたは `'signal 0'` コマンドを使用するとよいでしょう。See Section 5.3 [Signals], page 43)。

```
continue [ignore-count]
c [ignore-count]
fg [ignore-count]
```

ユーザ・プログラムが最後に停止した箇所から、プログラムの実行を再開します。停止箇所に設定されているブレークポイントは無視されます。オプションの引数 *ignore-count* によって、停止箇所のブレークポイントを無視する回数を指定することができます。これは `ignore` コマンドと似た効果を持ちます (see Section 5.1.6 [Break conditions], page 37)。

引数 *ignore-count* は、ユーザ・プログラムがブレークポイントによって停止した場合にのみ意味を持ちます。これ以外の場合には、`continue` コマンドへの引数は無視されます。

`c` および `fg` は、簡便さのためだけに提供されている同義コマンドで、`continue` コマンドと全く同様の動作をします。

別の箇所で実行を再開するには、呼び出し関数に戻る `return` コマンド (see Section 11.4 [Returning from a function], page 97) または、ユーザ・プログラム内の任意の箇所へ移動する `jump` コマンド (see Section 11.2 [Continuing at a different address], page 96) を使用することができます。

ステップ実行を使用する典型的なテクニックは、問題があると思われる関数やプログラム部分の先頭にブレークポイント (see Section 5.1 [Breakpoints; watchpoints; and catchpoints], page 29)

を設定し、ブレイクポイントで停止するまでプログラムを実行させた後、問題が再現するまで、関連しそうな変数の値を調べながら、疑わしい部分を 1 行ずつ実行することです。

**step** 異なるソース行に到達するまでユーザ・プログラムを継続実行した後、プログラムを停止させ、GDB に制御を戻します。このコマンドの省略形は `s` です。

注意: デバッグ情報なしでコンパイルされた関数の内部にいるときに `step` コマンドを使用すると、デバッグ情報付きの関数に達するまでプログラムの実行は継続されます。同様に、`step` コマンドがデバッグ情報なしでコンパイルされた関数の内部へ入って、停止することはありません。デバッグ情報を持たない関数の内部でステップ実行を行うには、後述の `stepi` コマンドを使用してください。

`step` コマンドは、ソース・コード行の最初の命令においてのみ停止するようになりました。これにより、以前のバージョンで発生していた、`switch` 文や `for` 文などにおいて複数回停止してしまうという問題が回避されています。同じ行の中にデバッグ情報を持つ関数への呼び出しがあると、`step` コマンドは続けて停止します。

さらに、`step` コマンドは、サブルーチンが行番号情報を持つ場合に限り、サブルーチンの内部に入り込むようになりました。サブルーチンが行番号情報を持たない場合、`step` コマンドは `next` コマンドと同様の動作をします。これにより、MIPS マシン上で `cc -gl` を使用した場合に発生していた問題が回避されています。以前のバージョンでは、サブルーチンが何らかのデバッグ情報を持っていれば、その内部に入り込んでいました。

**step count**

`step` コマンドによるステップ実行を `count` 回繰り返します。ステップ実行を `count` 回繰り返して終わる前に、ブレイクポイントに到達するか、あるいは、ステップ実行とは関連のないシグナルが発生した場合には、ただちにステップ実行を中断して停止します。

**next [count]**

カレントな ( 最下位の ) スタック・フレーム上において、ソース・コード上の次の行まで実行します。これは `step` コマンドと似ていますが、`next` コマンドは、ソース・コード上に関数呼び出しが存在すると、その関数を停止することなく最後まで実行します。プログラムが停止するのは、`next` コマンドを実行したときと同一のスタック・フレーム上において、ソース・コード上の異なる行まで実行が継続されたときです。このコマンドの省略形は `n` です。

引数 `count` は、`step` コマンドの場合と同様、繰り返し回数です。

`next` コマンドは、ソース・コード行の最初の命令においてのみ停止するようになりました。これにより、以前のバージョンで発生していた、`switch` 文や `for` 文などにおいて複数回停止してしまうという問題が回避されています。

**finish** 選択されているスタック・フレーム上の関数が復帰するまで、実行を継続します。戻り値があれば、それを表示します。

`return` コマンド ( see Section 11.4 [Returning], page 97 ) と比較してみてください。

**until**

**u**

カレントなスタック・フレーム上において、カレント行よりも後ろにある行に到達するまで実行を継続します。このコマンドは、ループ内において複数回ステップ実行をするのを回避するために使用されます。これは `next` コマンドに似ていますが、唯一の相違点は、`until` コマンドによってジャンプ命令が実行された場合、プログラム・カウンタの値がジャンプ命令のアドレスより大きくなるまで、プログラムが継続実行されるという点です。

これは、ステップ実行によってループ内の最後の行に到達した後に `until` コマンドを実行することで、ループから抜け出るまでプログラムを継続実行させることができるということを意味しています。これに対して、ループ内の最後の行で `next` コマンドを実行すると、プログラムはループの先頭に戻ってしまうので、ループ内の処理を繰り返すことを余儀なくされます。

`until` コマンドの実行により、プログラムがカレントなスタック・フレームから抜け出ようとする、そこで `until` コマンドはプログラムを停止します。

実行されるマシン・コードの順序がソース行の順序と一致しない場合、`until` コマンドは直観にいくらか反するような結果をもたらすかもしれません。例えば、以下に挙げるデバッグ・セッションからの抜粋では、`f (frame)` コマンドによって、プログラムが 206 行めにおいて停止していることが示されています。ところが、`until` コマンドを実行すると、195 行めで停止してしまいます。

```
(gdb) f
#0  main (argc=4, argv=0xf7fffae8) at m4.c:206
206          expand_input();
(gdb) until
195          for ( ; argc > 0; NEXTARG) {
```

これは、コンパイラが、実行の効率を高めるために、C 言語では `for` ループ本体の前に記述されているループ終了のための条件判定を、ループの先頭ではなく末尾で行うコードを生成したためです。この判定式にまで処理が進んだとき、`until` コマンドはあたかもループの先頭に戻ったかのように見えます。しかしながら、実際のマシン・コードのレベルでは、前の命令に戻ったわけではありません。

引数のない `until` コマンドは、1 命令ごとのステップ実行によって実現されるため、引数付きの `until` コマンドに比べて処理性能が劣ります。

#### `until location`

`u location` `location` で指定される箇所に到達するか、カレントなスタック・フレームを抜け出るまで、ユーザ・プログラムを継続実行します。`location` は `break` コマンドの受け付ける形式の引数です (see Section 5.1.1 [Setting breakpoints], page 30)。この形式による `until` コマンドはブレイクポイントを使用するため、引数のない `until` コマンドより処理性能が優れています。

#### `stepi`

`si` 1 マシン命令を実行した後、停止してデバッガに戻ります。  
マシン命令単位でステップ実行する場合、`'display/i $pc'` を使用すると便利ながしばしばあります。これは、ユーザ・プログラムが停止するたびに、次に実行される命令を GDB に自動的に表示させます。See Section 8.6 [Automatic display], page 64。  
引数として、`step` コマンドと同様、繰り返し回数を取ります。

#### `nexti`

`ni` 1 マシン命令を実行しますが、それが関数の呼び出しである場合は、関数から復帰するまで実行を継続します。  
引数として、`next` コマンドと同様、繰り返し回数を取ります。

## 5.3 シグナル

シグナルは、プログラム内で発生する非同期イベントです。オペレーティング・システムによって、使用可能なシグナルの種類が定義され、それぞれに名前と番号が割り当てられます。例えば、UNIX

においては、割り込み（通常は、`Ctrl`キーを押しながら `C`を押す）を入力したときにプログラムが受信する `SIGINT`、その使用領域からかけ離れたメモリ域を参照したときにプログラムが受信する `SIGSEGV`、アラームのタイムアウト時に発生する（プログラムからアラームを要求した場合にのみ発生する）`SIGALRM`シグナルなどがあります。

`SIGALRM`など、いくつかのシグナルは、プログラムの正常な機能の一部です。`SIGSEGV`などの他のシグナルは、エラーを意味します。これらのシグナルは、プログラムが事前にそれを処理する何らかの方法を指定しないと、致命的な（プログラムを即座に終了させる）ものとなります。`SIGINT`はユーザ・プログラム内部のエラーを意味するものではありませんが、通常は致命的なものであり、割り込みの目的であるプログラムの終了を実現することができます。

GDB は、ユーザ・プログラム内部における任意のシグナル発生を検出することができます。ユーザは、個々のシグナルの発生時に何を実行するかを、GDB に対して事前に指定することができます。

通常 GDB は、`SIGALRM`のようなエラーではないシグナルを無視するよう（これらのシグナルがユーザ・プログラムの中で持っている役割を妨害することのないよう）設定されています。その一方で、エラーのシグナルが発生した場合にはすぐにユーザ・プログラムを停止させるよう設定されています。これらの設定は `handle` コマンドによって変更することができます。

#### `info signals`

すべてのシグナルを一覧にして表示します。また、個々のシグナルについて、GDB がそれをどのように処理するよう設定されているかを表示します。このコマンドを使用して、定義済みのすべてのシグナルのシグナル番号を知ることができます。

`info handle`は、`info signals`に対して設定された新しい別名です。

#### `handle signal keywords...`

GDB が `signal` で指定されるシグナルを処理する方法を変更します。`signal` には、シグナル番号またはシグナル名称（先頭の '`SIG`'は省略可能）を指定します。キーワード `keywords` によって、どのように変更するかを指定します。

`handle` コマンドが受け付けるキーワードには省略形を使用することができます。省略しない場合、キーワードは以下のようになります。

<code>nostop</code>	GDB に対して、このシグナルが発生してもユーザ・プログラムを停止しないよう指示します。GDB は、シグナルを受信したことをメッセージ出力によってユーザに通知することができます。
<code>stop</code>	GDB に対して、このシグナルが発生するとユーザ・プログラムを停止するよう指示します。これは、 <code>print</code> キーワードを暗黙のうちに含みます。
<code>print</code>	GDB に対して、このシグナルが発生するとメッセージを表示するよう指示します。
<code>noprint</code>	GDB に対して、このシグナルが発生したことを知らせないよう指示します。これは、 <code>nostop</code> キーワードを暗黙のうちに含みます。
<code>pass</code>	GDB に対して、このシグナルの発生をユーザ・プログラムが検出できるようにするよう指示します。ユーザ・プログラムはシグナルを処理することができます。致命的で処理できないシグナルが発生した場合、ユーザ・プログラムは停止するかもしれません。
<code>nopass</code>	GDB に対して、このシグナルの発生をユーザ・プログラムが検出できないようにするよう指示します。

シグナルによってユーザ・プログラムが停止した場合、実行を継続するまでそのシグナルは検出されません。その時点において、そのシグナルに対して `pass` キーワードが有効であれば、ユーザ・プロ

グラムは、実行継続時にシグナルを検出します。言い換えれば、GDB がシグナルの発生を報告してきたとき、`handle` コマンドに `pass` キーワードまたは `nopass` キーワードを指定することで、実行を継続したときにプログラムにそのシグナルを検出させるか否かを制御することができます。

また、`signal` コマンドを使用することによって、ユーザ・プログラムがシグナルを検出できないようにしたり、通常は検出できないシグナルを検出できるようにしたり、あるいは任意の時点で任意のシグナルをユーザ・プログラムに検出させたりすることができます。例えば、ユーザ・プログラムが何らかのメモリ参照エラーによって停止した場合、ユーザは、さらに実行を継続しようとして、問題のある変数に正しい値を設定して継続実行しようとするかもしれません。しかし、実行継続直後に検出される致命的なシグナルのために、おそらくユーザ・プログラムはすぐに終了してしまうでしょう。このようなことを回避したければ、`'signal 0'` コマンドによって実行を継続することができます。See Section 11.3 [Giving your program a signal], page 96。

## 5.4 マルチスレッド・プログラムの停止と起動

ユーザ・プログラムが複数のスレッド ( see Section 4.10 [Debugging programs with multiple threads], page 24 ) を持つ場合、すべてのスレッドにブレイクポイントを設定するか、特定のスレッドにブレイクポイントを設定するかを選択することができます。

```
break linespec thread threadno
```

```
break linespec thread threadno if ...
```

`linespec` はソース行を指定します。記述方法はいくつかありますが、どの方法を使っても結果的にはソース行を指定することになります。

`break` コマンドに修飾子 `'thread threadno'` を使用することで、ある特定のスレッドがこのブレイクポイントに到達したときだけ GDB がプログラムを停止するよう、指定することができます。ここで `threadno` は、GDB によって割り当てられるスレッド識別番号で、`'info threads'` コマンドによる出力の最初の欄に表示されるものです。

ブレイクポイントを設定する際に `'thread threadno'` を指定しなければ、そのブレイクポイントはユーザ・プログラム内部のすべてのスレッドに適用されます。

条件付きのブレイクポイントに対しても `thread` 識別子を使用することができます。この場合、以下のように `'thread threadno'` をブレイクポイント成立条件の前に記述してください。

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

いかなる理由によるのであれ GDB 配下においてユーザ・プログラムが停止した場合、カレント・スレッドだけではなく、すべての実行スレッドが停止します。これにより、知らないうちに状態の変化が発生することを心配することなく、スレッドの切り替えも含めて、プログラム全体の状態を検査することができます。

逆に、プログラムの実行を再開したときには、すべてのスレッドが実行を開始します。これは、`step` コマンドや `next` コマンドによるシングル・ステップ実行の場合でも同様です。

特に GDB は、すべてのスレッドの歩調を合わせてシングル・ステップ実行することはできません。スレッドのスケジューリングは、デバッグ対象のマシンのオペレーティング・システムに依存する ( GDB が管理するわけではない ) ので、カレント・スレッドがシングル・ステップの実行を完了する前に、他のスレッドは複数の文を実行してしまうかもしれません。また、プログラムが停止するとき、他のスレッドは 2 つの文の間の境界のところまでびったり停止するよりも、文の途中で停止してしまう方が一般的です。

また、継続実行やステップ実行の結果、プログラムが別のスレッド内で停止してしまうこともあります。最初のスレッドがユーザの要求した処理を完了する前に、他のスレッドがブレイクポイントに到達した場合、シグナルを受信した場合、例外が発生した場合には、常にこのようなことが発生します。

OSによっては、OS スケジューラをロックすることによって、ただ1つのスレッドだけが実行されるようにすることができます。

`set scheduler-locking mode`

スケジューラのロッキング・モード ( `locking mode` ) を設定します。offの場合は、ロックのメカニズムは機能せず、任意の時点において、どのスレッドも実行される可能性を持ちます。onの場合は、再始動 ( `resume` ) されるスレッドの優先順位が低い場合には、カレント・スレッドだけが実行を継続することができます。stepモードでは、シングル・ステップ実行のための最適化が行われます。ステップ実行をしている間、他のスレッドが「プロンプトを横取りする」ことがないよう、カレント・スレッドに占有権が与えられます。また、ステップ実行をしている間、他のスレッドはきわめて稀にしか (あるいは、まったく) 実行するチャンスを与えられません。nextコマンドによって関数呼び出しの次の行まで処理を進めると、他のスレッドが実行される可能性は高くなります。また、`continue`、`until`、`finish`のようなコマンドを使用すると、他のスレッドは完全に自由に実行されることができます。しかし、そのタイムスライスの中でブレイクポイントに到達しない限り、他のスレッドが、デバッグの対象となっているスレッドから、GDB プロンプトを横取りすることはありません。

`show scheduler-locking`

スケジューラの現在のロッキング・モードを表示します。



## 6 スタックの検査

ユーザ・プログラムが停止したとき、まず最初に、どこで停止したのか、そして、どのようにしてそこに到達したのかを知る必要があるでしょう。

ユーザ・プログラムが関数呼び出しを行うたびに、その呼び出しに関する情報が生成されます。その情報には、ユーザ・プログラム内においてその呼び出しが発生した場所、関数呼び出しの引数、呼び出された関数内部のローカル変数などが含まれます。その情報は、スタック・フレームと呼ばれるデータ・ブロックに保存されます。スタック・フレームは、呼び出しスタックと呼ばれるメモリ域に割り当てられます。

ユーザ・プログラムが停止すると、スタックを検査する GDB コマンドを使用して、この情報をすべて見るができます。

GDB は 1 つのスタック・フレームを選択していて、多くの GDB コマンドはこの選択されたフレームを暗黙のうちに参照します。特に、GDB に対してユーザ・プログラム内部の変数の値を問い合わせると、GDB は選択されたフレームの内部においてその値を探そうとします。関心のあるフレームを選択するための特別な GDB コマンドが提供されています。See Section 6.3 [Selecting a frame], page 49。

ユーザ・プログラムが停止すると、GDB はその時点において実行中のフレームを自動的に選択し、`frame` コマンド ( see Section 6.4 [Information about a frame], page 50 ) のように、そのフレームに関する情報を簡潔に表示します。

### 6.1 スタック・フレーム

呼び出しスタックは、スタック・フレーム、または短縮してフレームと呼ばれる、連続した小部分に分割されます。個々のフレームは、ある関数に対する 1 回の呼び出しに関連するデータです。フレームには、関数への引数、関数のローカル変数、関数の実行アドレスなどの情報が含まれます。

ユーザ・プログラムが起動されたとき、スタックには `main` 関数のフレームが 1 つ存在するだけです。これは、初期フレームまたは「最上位のフレーム」と呼ばれます。関数が呼び出されるたびに、新たにフレームが作成されます。関数が復帰すると、その関数を呼び出したときに生成されたフレームが取り除かれます。関数が再帰的に呼び出される場合、1 つの関数に対して多くのフレームが生成されるということもありえます。実際に実行中の関数に対応するフレームは、「最下位のフレーム」と呼ばれます。これは、存在するすべてのスタック・フレームの中で、最も新しく作成されたものです。

ユーザ・プログラムの内部においては、スタック・フレームはアドレスによって識別されます。スタック・フレームは多くのバイトから構成され、それぞれがそれ自身のアドレスを持っています。そのアドレスがフレームのアドレスとなるような 1 バイトを選択する慣習的な方法を、すべての種類のコンピュータが提供しています。通常、あるフレーム内部で実行中は、そのフレームのアドレスがフレーム・ポインタ・レジスタと呼ばれるレジスタに格納されています。

GDB は、既存のスタック・フレームのすべてに番号を割り当てます。最下位のフレームは 0 で、それを呼び出したフレームは 1 となります。以下、最下位のフレームを起点として、順番に値を割り当てていきます。これらの番号はユーザ・プログラム内部には実際には存在しません。これらの番号は、GDB コマンドでスタック・フレームを指定することができるように、GDB によって割り当てられたものです。

コンパイラによっては、スタック・フレームを使用しなくても実行可能なように関数をコンパイルする方法を提供しているものもあります (例えば、gcc のオプション `-fomit-frame-pointer` を指定すると、フレームを持たない関数が生成されます)。これは、フレームをセットアップする時間を節約するために、頻繁に利用されるライブラリ関数に対してしばしば適用されます。これらの関数の呼

び出しを処理するために GDB が提供する機能は限られています。最下位のフレームの関数呼び出しがスタック・フレームを持たない場合、GDB は、あたかもそれが通常どおりに番号 0 のフレームを持つものとみなして、関数呼び出しの連鎖を跡づけることができますようにします。しかしながら、最下位以外のスタック位置に存在する、フレームを持たない関数に対しては、GDB は特別な処置を取りません。

`frame args`

`frame` コマンドによって、あるスタック・フレームから別のスタック・フレームに移動し、選択したスタック・フレームを表示させることができます。`args` は、フレームのアドレスまたはスタック・フレーム番号です。引数なしで実行すると、`frame` コマンドはカレントなスタック・フレームを表示します。

`select-frame`

`select-frame` コマンドによって、フレームを表示することなく、あるスタック・フレームから別のスタック・フレームに移動することができます。これは、`frame` コマンドから、表示処理を取り除いたものです。

## 6.2 バックトレース

バックトレースとは、ユーザ・プログラムが現在いる箇所にどのようにして到達したかを示す要約情報です。複数のフレームが存在する場合、1 フレームの情報を 1 行に表示します。現在実行中のフレーム (番号 0 のフレーム) を先頭に、それを呼び出したフレーム (番号 1 のフレーム) を次行に、以降、同様にスタックをさかのぼって情報を表示します。

`backtrace`

`bt` 全スタックのバックトレースを表示します。スタック内のすべてのフレームが、1 行に 1 フレームずつ表示されます。

システムの割り込み文字 (通常は、`Ctrl` キーを押しながら `C` を押す) によって、いつでもバックトレースを停止することができます。

`backtrace n`

`bt n` 引数のない `backtrace` コマンドと似ていますが、最下位のフレームから数えて `n` 個のフレームだけが表示されます。

`backtrace -n`

`bt -n` 引数のない `backtrace` コマンドと似ていますが、最上位のフレームから数えて `n` 個のフレームだけが表示されます。

`backtrace` の別名としては、ほかに `where` や `info stack` (省略形は `info s`) があります。

`backtrace` コマンドの出力結果の各行に、フレーム番号と関数名が表示されます。`set print address off` コマンドを実行していなければ、プログラム・カウンタの値も表示されます。`backtrace` コマンドの出力結果では、関数への引数に加えて、ソース・ファイル名や行番号も表示されます。プログラム・カウンタが、行番号で指定される行の最初のコードを指している場合、その値は省略されます。

以下に `backtrace` の例を示します。これは、`'bt 3'` の出力であり、したがって最下位のフレームから 3 フレームが表示されています。

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
    (More stack frames follow...)
```

番号 0 のフレームを表示する行の先頭には、プログラム・カウンタの値がありません。これは、builtin.cの 993行めの最初のコードにおいてユーザ・プログラムが停止したことを表わしています。

### 6.3 フレームの選択

スタックやユーザ・プログラム内の他のデータを調べるためのほとんどのコマンドは、それが実行された時点において選択されているスタック・フレーム上で動作します。以下に、スタック・フレームを選択するためのコマンドを列挙します。どのコマンドも、それによって選択されたスタック・フレームに関する簡単な説明を最後に表示します。

frame *n*

*f n*      番号 *n* のフレームを選択します。最下位の ( 現在実行中の ) フレームが番号 0 のフレーム、最下位のフレームを呼び出したフレームが番号 1 のフレーム、以下同様となります。最も大きい番号を持つフレームは main のフレームです。

frame *addr*

*f addr*      アドレス *addr* のフレームを選択します。スタック・フレームの連鎖がバグのために破壊されてしまって、GDB がすべてのフレームに正しく番号を割り当てられないような場合に、この方法が役に立ちます。さらに、ユーザ・プログラムが複数のスタックを持ち、スタックの切り替えを行うような場合にも有効です。

SPARC アーキテクチャでは、フレームを任意に選択するには、フレーム・ポインタ、スタック・ポインタの 2 つのアドレスを frame に指定する必要があります。

MIPS、Alpha の両アーキテクチャでは、スタック・ポインタ、プログラム・カウンタの 2 つのアドレスが必要です。

29k アーキテクチャでは、レジスタ・スタック・ポインタ、プログラム・カウンタ、メモリ・スタック・ポインタの 3 つのアドレスが必要です。

*up n*      スタックを *n* フレームだけ上へ移動します。*n* が正の値の場合、最上位のフレームに向かって移動します。これは、より大きいフレーム番号を持ち、より長く存在しているフレームへの移動です。*n* のデフォルト値は、0 です。

*down n*      スタックを *n* フレームだけ下へ移動します。*n* が正の値の場合、最下位のフレームに向かって移動します。これは、より小さいフレーム番号を持ち、より最近作成されたフレームへの移動です。*n* のデフォルト値は 1 です。down の省略形は do です。

これらのコマンドはいずれも、最後にフレームに関する情報を 2 行で表示します。1 行めには、フレーム番号、関数名、引数、ソース・ファイル名、そのフレーム内において実行停止中の行番号が表示されます。2 行めには、実行停止中のソース行が表示されます。

以下に、例を示します。

```
(gdb) up
#1  0x22f0 in main (argc=1, argv=0xf7ffbf4, env=0xf7ffbf4)
    at env.c:10
10      read_input_file (argv[i]);
```

この情報が表示された後で、`list` コマンドを引数なしで実行すると、フレーム内で実行停止中の行を中心に 10 行のソース行が表示されます。See Section 7.1 [Printing source lines], page 53。

`up-silently n`

`down-silently n`

これら 2 つのコマンドは、それぞれ、`up` コマンド、`down` コマンドの変種です。相違点は、ここに挙げた 2 つのコマンドが、新しいフレームに関する情報を表示することなく実行されるという点にあります。これらは、情報の出力が不必要で邪魔ですらある、GDB のコマンド・スクリプトの中での使用を主に想定したものです。

## 6.4 フレームに関する情報

既に挙げたものの以外にも、選択されたフレームに関する情報を表示するコマンドがいくつかあります。

`frame`

`f`

このコマンドは、引数なしで実行されると、別のフレームを選択するのではなく、その時点において選択中のフレームに関する簡単な説明を表示します。このコマンドの省略形は `f` です。引数付きの場合、このコマンドはスタック・フレームを選択するのに使用されます。See Section 6.3 [Selecting a frame], page 49。

`info frame`

`info f`

このコマンドは、選択されたフレームに関する詳細な情報を表示します。表示される情報には、以下のようなものがあります。

- フレームのアドレス
- 1 つ下位の ( 選択されたフレームによって呼び出された ) フレームのアドレス
- 1 つ上位の ( 選択されたフレームを呼び出した ) フレームのアドレス
- 選択されたフレームに対応するソース・コードを記述した言語
- フレームの引数のアドレス
- 退避されているプログラム・カウンタ ( 呼び出し側フレームの実行アドレス )
- 退避されているレジスタ

これらの詳細な情報は、何か問題が発生して、スタックの形式が通常の慣習に合致しなくなった場合に、役に立ちます。

`info frame addr`

`info f addr`

アドレス `addr` のフレームに関する詳細な情報を、そのフレームを選択することなく表示します。このコマンドによって、その時点において選択されていたフレームとは異なるフレームが選択されてしまうことはありません。このコマンドでは、`frame` コマンドに指定するのと同様のアドレスを ( アーキテクチャによっては複数 ) 指定する必要があります。See Section 6.3 [Selecting a frame], page 49。

`info args` 選択中のフレームの引数を、1 行に 1 つずつ表示します。

`info locals`

選択中のフレームのローカル変数を、1 行に 1 つずつ表示します。これらはすべて、選択中のフレームの実行箇所においてアクセス可能な ( 静的変数または自動変数として宣言された ) 変数です。

`info catch`

選択中のスタック・フレームの実行箇所においてアクティブな状態にある、すべての例外ハンドラの一覧を表示します。他の例外ハンドラを参照したい場合は、関連するフレームに（`up`コマンド、`down`コマンド、`frame`コマンドを使用して）移動してから、`info catch`を実行します。See Section 5.1.3 [Setting catchpoints], page 34。

## 6.5 MIPS/Alpha マシンの関数スタック

Alpha ベースのコンピュータと MIPS ベースのコンピュータは、変わったスタック・フレームを使用しています。そのため、関数の先頭を見つけるために、GDB はときどきオブジェクト・コードを逆方向に検索する必要があります。

応答時間を改善するために（特に、このような検索を実行するのに、速度の遅いシリアル回線を使用するほかない、組み込みアプリケーションの場合）、以下に列挙するコマンドを使用して検索量を制限するとよいでしょう。

`set heuristic-fence-post limit`

関数の先頭を検索するために GDB が検査するバイト数を、最高で *limit* バイトに制限します。*limit* に 0（デフォルト）を指定すると、無制限に検索することを意味します。*limit* に 0 以外の値を指定すると、その値が大きければ大きいほど検索バイト数も多くなり、したがって検索の実行により長い時間がかかります。

`show heuristic-fence-post`

現在の検索制限値を表示します。

これらのコマンドは、GDB が、Alpha プロセッサ上、または、MIPS プロセッサ上においてプログラムをデバッグするよう構成されている場合にのみ使用することができます。



## 7 ソース・ファイルの検査

GDB は、ユーザ・プログラムのソース・コードの一部を表示することができます。これは、プログラムの中に記録されているデバッグ情報によって、そのプログラムをビルドするのに使用されたソース・ファイルを GDB が知ることができるからです。ユーザ・プログラムが停止すると、GDB は自動的にプログラムが停止した行を表示します。同様に、ユーザがあるスタック・フレーム (see Section 6.3 [Selecting a frame], page 49) を選択すると、そのフレームにおいて実行が停止している行を GDB は表示します。明示的にコマンドを使用することで、ソース・ファイルの他の部分を表示することも可能です。

GNU Emacs インターフェイス経由で GDB を使用しているユーザは、Emacs の提供する機能を使ってソース・ファイルを参照する方を好むかもしれません。これについては、See Chapter 16 [Using GDB under GNU Emacs], page 141。

### 7.1 ソース行の表示

ソース・ファイル内の行を表示するには、`list` コマンド (省略形は `l`) を使用します。デフォルトでは、10 行が表示されます。ソース・ファイルのどの部分を表示するかを指定する方法がいくつかあります。

最もよく使われる `list` コマンドの形式を以下に示します。

`list linenum`

現在のソース・ファイルの行番号 *linenum* を中心に、その前後の行を表示します。

`list function`

関数 *function* の先頭を中心に、その前後の行を表示します。

`list`        ソース・ファイル行の続きを表示します。既に表示された最後の行が `list` コマンドによって表示されたのであれば、その最後の行の次の行以降が表示されます。しかし、既に表示された最後の行が、スタック・フレーム (see Chapter 6 [Examining the Stack], page 47) の表示の一部として 1 行だけ表示されたのであれば、その行の前後の行が表示されます。

`list -`       前回表示された行の前に位置する行を表示します。

`list` コマンドを上記の形式のいずれかによって実行すると、GDB はデフォルトでは 10 行のソース行を表示します。これは `set listsize` コマンドによって変更することができます。

`set listsize count`

`list` コマンドで表示される行数を *count* に設定します (`list` コマンドの引数で他の値が明示的に指定された場合は、この設定は効力を持ちません)。

`show listsize`

`list` コマンドが表示する行数を表示します。

`list` コマンドを実行後、`(RET)` キーによって `list` コマンドを実行した場合、引数は破棄されます。したがって、これは単に `list` と入力して実行したのと同じことになります。同じ行が繰り返し表示されるよりも、この方が役に立つでしょう。ただし、引数 `-` は例外となります。この引数は繰り返し実行の際にも維持されるので、繰り返し実行することで、ソース・ファイルの内容がさかのぼって表示されていきます。

一般的には、`list` コマンドは、ユーザによって 0 個、1 個、または 2 個の行指定 (*linespec*) が与えられることを期待しています。ここで行指定とは、ソース行を指定するものです。いくつかの記述方法がありますが、いずれも結果的には何らかのソース行を指定するものです。`list` コマンドの引数として使用できる引数の完全な説明を以下に示します。

`list linespec`

*linespec* によって指定される行を中心に、その前後の行を表示します。

`list first,last`

*first* 行から *last* 行までを表示します。両引数はいずれも行指定です。

`list ,last` *last* 行までを表示します。

`list first,`

*first* 行以降を表示します。

`list +` 最後に表示された行の次の行以降を表示します。

`list -` 最後に表示された行の前の行以前を表示します。

`list` 前述のとおり。

以下に、ソースの特定の 1 行を指定する方法を示します。これは、いずれも行指定です。

*number* 現在のソース・ファイルの行番号 *number* の行を指定します。`list` コマンドの引数に 2 つの行指定がある場合、2 つめの行指定は、最初の行指定と同一のソース・ファイルを指定します。

`+offset` 最後に表示された行から *offset* で指定される行数だけ下にある行を指定します。2 つの行指定を引数として持つ `list` コマンドにおいて、これが 2 つめの行指定として使用される場合、最初の行指定から *offset* で指定される行数だけ下の行を指定します。

`-offset` 最後に表示された行から *offset* で指定される行数だけ上にある行を指定します。

`filename: number`

ソース・ファイル *filename* の行番号 *number* の行を指定します。

*function* 関数 *function* の本体の先頭行を指定します。例えば C 言語では、左括弧 (‘{’) のある行を指します。

`filename: function`

ファイル *filename* 内の関数 *function* の本体を開始する左括弧 (‘{’) のある行を指定します。異なるソース・ファイルの中に同一の名前の関数が複数ある場合にのみ、あいまいさを回避するために、関数名とともにファイル名を指定する必要があります。

`*address` プログラム・アドレス *address* を含む行を指定します。*address* には任意の式を指定することができます。

## 7.2 ソース・ファイル内の検索

カレントなソース・ファイル内において正規表現による検索を行うためのコマンドが 2 つあります。

`forward-search regexp`

`search regexp`

‘`forward-search regexp`’ コマンドは、最後に `list` コマンドによって表示された行の 1 つ下の行から、1 行ずつ正規表現 *regexp* による検索を行います。正規表現にマッチす



るものが見つかったら、その行を表示します。‘*search regexp*’という同義のコマンドを使うこともできます。コマンド名は、省略して *fo* とすることができます。

*reverse-search regexp*

‘*reverse-search regexp*’コマンドは、最後に *list* コマンドによって表示された行の 1 つ上の行から、1 行ずつ逆方向に向かって正規表現 *regexp* による検索を行います。正規表現にマッチするものが見つかったら、その行を表示します。コマンド名は、省略して *rev* とすることができます。

### 7.3 ソース・ディレクトリの指定

実行形式プログラムは、それがコンパイルされたソース・ファイルの名前だけを記録して、ソース・ファイルの存在するディレクトリ名を記録しないことがあります。また、ディレクトリ名が記録された場合でも、コンパイル時とデバッグ時との間に、そのディレクトリが移動してしまっている可能性があります。GDB は、ソース・ファイルを検索すべきディレクトリの一覧を持っています。これは、ソース・パスと呼ばれます。GDB は、ソース・ファイルが必要なときにはいつでも、それが見つかるまで、このリストの中のすべてのディレクトリを、リストの中に記述されている順に探します。実行ファイルのサーチ・パスは、この目的では使用されないことに気をつけてください。またカレントな作業ディレクトリも、それがたまたまソース・パスの中にある場合を除けば、この目的で使用されることはありません。

GDB がソース・パスの中でソース・ファイルを見つけることができない場合、プログラムがディレクトリ名を記録してあれば、そのディレクトリも検索されます。ソース・パスにディレクトリの指定がなく、コンパイルされたディレクトリの名前も記録されていない場合、GDB は最後の手段としてカレント・ディレクトリを探します。

ソース・パスを空にした場合、または、再調整した場合、ソース・ファイルを見つけた場所や個々の行のファイル内の位置のような、GDB が内部でキャッシュしている情報は消去されます。

GDB を起動した時点では、ソース・パスにはディレクトリの指定がありません。ディレクトリをソース・パスに追加するには、*directory* コマンドを使用してください。

*directory dirname ...*

*dir dirname ...*

ディレクトリ *dirname* をソース・パスの先頭に追加します。個々のディレクトリをコロン ‘:’ または空白で区切ることによって、複数のディレクトリをこのコマンドに渡すことができます。ソース・パスの中に既に存在するディレクトリを指定することもできます。この場合、そのディレクトリの、ソース・パスの中での位置が前に移動するので、GDB はそのディレクトリの中を以前よりも早く検索することになります。

(コンパイル時のディレクトリが記録されていれば) それを指すのに文字列 ‘\$cdir’ を使うことができます。また、カレントな作業ディレクトリを指すには、文字列 ‘\$cwd’ を使うことができます。‘\$cwd’ と ‘.’ (ピリオド) とは同じではありません。前者は、GDB セッション内においてカレントな作業ディレクトリが変更された場合、変更されたディレクトリを指します。これに対して後者は、ソース・パスへの追加を行ったときに、その時点におけるカレント・ディレクトリに展開されてしまいます。

*directory*

ソース・パスの内容を再び空にします。ソース・パスを空にする前に、確認を求めてきます。

`show directories`

ソース・パスを表示します。ソース・パスに含まれるディレクトリ名を見ることができます。

ソース・パスの中に、不要となってしまったディレクトリが混在していると、GDB が誤ったバージョンのソースを見つけてしまい、混乱をもたらすことがあります。以下の手順によって、正常な状態にすることができます。

1. ソース・パスを空にするために、`directory` コマンドを引数なしで実行します。
2. ソース・パス中に含めたいディレクトリが組み込まれるよう、`directory` コマンドに適切な引数を指定して実行します。すべてのディレクトリを、1 回のコマンド実行で追加することができます。

## 7.4 ソースとマシン・コード

`info line` コマンドを使用してソース行をプログラム・アドレスに（あるいは、プログラム・アドレスをソース行に）対応付けすることができます。また、`disassemble` コマンドを使用して、あるアドレス範囲をマシン命令として表示することもできます。GNU Emacs モードで実行されている場合、現在の `info line` コマンドは、指定された行を示す矢印を表示します。また、`info line` コマンドは、アドレスを 16 進形式だけではなくシンボリック形式でも表示します。

`info line linespec`

ソース行 *linespec* に対応するコンパイル済みコードの開始アドレス、終了アドレスを表示します。`list` コマンド（see Section 7.1 [Printing source lines], page 53）が理解できる任意の形式によってソース行を指定することができます。

例えば、`info line` コマンドによって、関数 `m4_changequote` の最初の行に対応するオブジェクト・コードの位置を知ることができます。

```
(gdb) info line m4_changequote
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

また、（*linespec* の形式として *\*addr* を使用することで）ある特定のアドレスがどのソース行に含まれているのかを問い合わせることができます。

```
(gdb) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.
```

`info line` の実行後、`x` コマンドのデフォルト・アドレスは、その行の先頭アドレスに変更されます。これにより、マシン・コードの調査を開始するには '`x/i`' を実行するだけで十分となります（see Section 8.5 [Examining memory], page 63）。また、このアドレスはコンビニエンス変数 `$_` の値として保存されます（see Section 8.9 [Convenience variables], page 71）。

`disassemble`

この特殊コマンドは、あるメモリ範囲をマシン命令としてダンプ出力します。デフォルトのメモリ範囲は、選択されたフレームにおいてプログラム・カウンタが指している箇所を含む関数です。このコマンドに引数を 1 つ渡すと、それはプログラム・カウンタ値を指定することになります。GDB は、その値が指す箇所を含んでいる関数をダンプ出力します。2 つの引数を渡すと、ダンプ出力するアドレス範囲（1 つめのアドレスは含まれますが、2 つめのアドレスは含まれません）を指定することになります。

以下の例は、あるアドレス範囲の HP PA-RISC 2.0 コードを逆アセンブルした結果を示しています。

```
(gdb) disas 0x32c4 0x32e4
Dump of assembler code from 0x32c4 to 0x32e4:
0x32c4 <main+204>:      addil 0,dp
0x32c8 <main+208>:      ldw 0x22c(sr0,r1),r26
0x32cc <main+212>:      ldil 0x3000,r31
0x32d0 <main+216>:      ble 0x3f8(sr4,r31)
0x32d4 <main+220>:      ldo 0(r31),rp
0x32d8 <main+224>:      addil -0x800,dp
0x32dc <main+228>:      ldo 0x588(r1),r26
0x32e0 <main+232>:      ldil 0x3000,r31
End of assembler dump.
```

アーキテクチャによっては、一般に使用される命令ニーモニックを複数持つものや、異なる構文を持つものがあります。

`set assembly-language instruction-set`

`disassemble` コマンドまたは `x/i` コマンドによってプログラムの逆アセンブルを行う際に使用する命令セットを選択します。

現在のところ、このコマンドは、Intel x86 ファミリに対してのみ定義されています。`instruction-set` は、`i386` と `i8086` のいずれかにセットすることができます。デフォルトは `i386` です。



## 8 データの検査

ユーザ・プログラムの中のデータを調べる通常の方法は、`print` コマンド（省略形は `p`）またはそれと同義のコマンドである `inspect` コマンドを使用することです。これは、ユーザ・プログラムが記述された言語（see Chapter 9 [Using GDB with Different Languages], page 75）による式を評価し、その値を出力するものです。

```
print exp
print /f exp
```

`exp` は（ソース言語による）式です。デフォルトでは、`exp` の値は、`exp` のデータ型にとって適切な形式で表示されます。‘`/f`’を指定することで、他の形式を選択することも可能です。‘`/f`’の `f` は形式を指定する文字です。see Section 8.4 [Output formats], page 62。

```
print
print /f
```

`exp` を省略すると、GDB は値ヒストリ（see Section 8.8 [Value history], page 70）の最後の値を再表示します。これは、同じ値を異なる形式で調べるのに便利です。

データを調べるためのより低レベルの方法は、`x` コマンドを使うことです。これは、指定されたアドレスのメモリ上のデータを、指定された形式で表示するものです。See Section 8.5 [Examining memory], page 63。

型に関する情報に関心があるとき、また、構造体やクラスのフィールドがどのように宣言されているかという点に関心があるときは、`print` コマンドではなく `ptype exp` コマンドを使用してください。See Chapter 10 [Examining the Symbol Table], page 91。

### 8.1 式

`print` コマンド、および、ほかの多くの GDB コマンドは、式を受け取って、その値を評価します。ユーザの使用しているプログラミング言語によって定義されている定数、変数、演算子は、いずれも GDB における式の中で有効です。これには、条件式、関数呼び出し、キャスト、文字列定数が含まれます。しかし、プリプロセッサの `#define` コマンドによって定義されるシンボルは、残念ながら含まれません。

現在の GDB は、ユーザの入力する式において配列定数をサポートします。その構文は、`{element, element...}` です。例えば、コマンド `print {1, 2, 3}` を使用して、ターゲット・プログラム内で `malloc()` によって獲得されたメモリ内に配列を作成することができます。

C 言語は大変広汎に使用されているので、このマニュアルの中で示される例の中の式は C 言語で記述されています。他の言語での式の使い方に関する情報については、See Chapter 9 [Using GDB with Different Languages], page 75。

この節では、プログラミング言語によらず GDB の式で利用できる演算子を説明します。

キャストは、C 言語のみならず、すべての言語でサポートされています。これは、メモリ内のあるアドレスにある構造体を調べるのに、数値をポインタにキャストするのが大変便利であるからです。

プログラミング言語によらず共通に使用可能な演算子に加えて、GDB は以下の演算子をサポートしています。

- @        ‘@’は、メモリの一部を配列として処理するための 2 項演算子です。詳細については、See Section 8.3 [Artificial arrays], page 61。
- ::       ‘::’によって、それを定義している関数またはファイルを特定して、変数を指定することができます。See Section 8.2 [Program variables], page 60。

`{type} addr`

`addr` で示されるメモリ上のアドレスに格納されている、`type` で示される型のオブジェクトを参照します。`addr` には、評価結果が整数値またはポインタになるような任意の式を指定することができます（ただし、2 項演算子の前後には、キャストを使う場合と同様の括弧が必要です）。これは、`addr` の位置に通常存在するデータの型がいかなるものであろうとも、使用することができます。

## 8.2 プログラム変数

最も一般的に使用される式は、ユーザ・プログラム内部の変数名です。

式の中の変数は、選択されたスタック・フレーム（see Section 6.3 [Selecting a frame], page 49）内において解釈されます。これは、以下の 2 つのいずれかとなります。

- グローバル変数（または、ファイル・スコープの静的変数）

あるいは

- プログラム言語のスコープ規則によって、そのフレームの実行中の箇所から可視の変数

つまり、以下の例において、ユーザ・プログラムが関数 `foo` を実行中は、変数 `a` を調べたり使用したりすることができますが、変数 `b` を使用したり調べたりすることができるのは、`b` が宣言されているブロックの内部をユーザ・プログラムが実行中である場合に限られます。

```
foo (a)
{
    int a;
    {
        bar (a);
        {
            int b = test ();
            bar (b);
        }
    }
}
```

ただし、これには 1 つ例外があります。特定の 1 ソース・ファイルをスコープとする変数や関数は、たとえ現在の実行箇所がそのファイルの中ではなくても、参照することができます。しかし、このような変数または関数が（異なるソース・ファイル中に）同じ名前でも複数個存在するということがありえます。このような場合、その名前を参照すると予期できない結果をもたらします。2 つのコーロンを並べる記法によって、特定の関数またはファイルの中の静的変数を指定することができます。

```
file::variable
function::variable
```

ここで `file` または `function` は、静的変数 `variable` のコンテキスト名です。ファイル名の場合は、引用符を使用することによって、GDB がファイル名を確実に 1 つの単語として解釈するようにさせることができます。例えば、ファイル `'f2.c'` の中で定義されたグローバル変数 `x` の値を表示するには、

```
(gdb) p 'f2.c'::x
```

のようにします。

このような `::` の用途が、これと非常によく似ている C++ における `::` の用途と衝突することは非常に稀です。GDB は、式の内部において C++ のスコープ解釈演算子の使用もサポートしています。

注意: ときどき、新しいスコープに入った直後やスコープから出る直前に、関数内部の特定の箇所から見ると、ローカル変数の値が正しくないように見えることがあります。

マシン命令単位でステップ実行を行っているときに、このような問題を経験することがあるかもしれません。これは、ほとんどのマシンでは、(ローカル変数定義を含む)スタック・フレームのセットアップに複数の命令が必要となるからです。マシン命令単位でステップ実行を行う場合、スタック・フレームが完全に構築されるまでの間は、変数の値が正しくないように見えることがあります。スコープから出るときには、スタック・フレームを破棄するのに、通常複数のマシン命令が必要とされます。それらの命令群の中をステップ実行し始めた後には、ローカル変数の定義は既に存在しなくなっているかもしれません。

このようなことは、コンパイラが重要な最適化を実施する場合にも、発生する可能性があります。常に正確な値が見えることを確実にするためには、コンパイルの際に、すべての最適化を行わないようにします。

### 8.3 人工配列

メモリ内に連続的に配置されている同一型のオブジェクトを表示することが役に立つことがよくあります。配列の一部や動的にサイズの決定される配列にアクセスするのに、そこへのポインタしかプログラム内部に存在しないような場合です。

これは、2項演算子 '@' を使用して、連続したメモリ範囲を人工配列として参照することで可能です。 '@' の左側のオペランドは、参照したい配列の最初の要素で、かつ、1個のオブジェクトでなければなりません。また、右側のオペランドは、その配列の中の参照したい部分の長さでなければなりません。結果は、その要素がすべて左側の引数と同型である配列の値です。第1の要素は左側の引数そのものです。第2の要素は、第1の要素を保持するメモリ域の直後のメモリ上から取られます。これ以降の要素も同様です。以下に例を示します。プログラムが以下のようになっているとしましょう。

```
int *array = (int *) malloc (len * sizeof (int));
```

以下を実行することで、arrayの内容を表示することができます。

```
p *array@len
```

'@' の左側のオペランドは、メモリ上に実在するものでなければなりません。このような方法で '@' によって作成された配列の値は、配列の添字付けの見地からは他の配列と同様に振る舞い、式の中で使用された場合は強制的にポインタとして扱われます。人工配列は、一度表示された後、値ヒストリ (see Section 8.8 [Value history], page 70) を通して式の中に現れることがよくあります。

人工配列を作成するもう1つの方法は、キャストを使用することです。これによって、ある値を配列として解釈し直します。この値は、メモリ上に実在するものでなくてもかまいません。

```
(gdb) p/x (short[2])0x12345678
$1 = {0x1234, 0x5678}
```

ユーザの便宜を考慮して、(例えば、'(type[])value'のように)配列の長さが省略された場合その値を満たすサイズを ('sizeof(value)/sizeof(type)'のように) GDB が計算します。

```
(gdb) p/x (short[])0x12345678
$2 = {0x1234, 0x5678}
```

ときには、人工配列の機構では十分でないことがあります。かなり複雑なデータ構造では、関心のある要素が連続的に並んでいないことがあります。例えば、配列の中のポインタの値に関心がある場合です。このような状況において役に立つ回避策の1つに、関心のある値のうち最初のもを表示する式の中のカウンタとしてコンビニエンス変数 (see Section 8.9 [Convenience variables], page 71) を使用し、(RET) キーによってその式を繰り返し実行することです。例えば、構造体へのポインタの配列 dtab があり、個々の構造体のフィールド fv の値に関心があるとしましょう。以下に、この場合の例を示します。

```

set $i = 0
p dtab[$i++]>->fv
(RET)
(RET)
...

```

## 8.4 出力フォーマット

デフォルトでは、GDB はデータの型にしたがって値を表示します。ときには、これが望ましくない場合もあります。例えば、数値を 16 進で表示したい場合やポインタを 10 進で表示したい場合があるでしょう。あるいは、メモリ内のある特定のアドレスのデータを文字列や命令として表示させたい場合もあるでしょう。このようなことをするためには、値を表示するとき出力フォーマットを指定します。

出力フォーマットの最も単純な使用法は、既に評価済みの値の表示方法を指定することです。これは、`print` コマンドの最初の引数をスラッシュとフォーマット文字で開始することで行います。サポートされているフォーマット文字は、以下のとおりです。

- x            値を整数値とみなし、16 進で表示します。
  - d            値を符号付き 10 進の整数値として表示します。
  - u            値を符号なし 10 進の整数値として表示します。
  - o            値を 8 進の整数値として表示します。
  - t            値を 2 進の整数値として表示します。‘t’は two を省略したものです。<sup>1</sup>
  - a            値を、16 進の絶対アドレス、および、そのアドレスより前にあるシンボルのうち最も近い位置にあるものからのオフセット・アドレスとして表示します。このフォーマットを使用することで、未知のアドレスがどこに（どの関数の中に）あるのかを知ることができます。
- ```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```
- c            値を整数値とみなし、文字定数として表示します。
  - f            値を浮動小数点数値とみなし、典型的な浮動小数点の構文で出力します。

例えば、プログラム・カウンタの値を 16 進数で表示する（see Section 8.10 [Registers], page 72）には、以下を実行してください。

```
p/x $pc
```

スラッシュの前にはスペースが必要ではないことに注意してください。これは、GDB のコマンド名にはスラッシュを含めることができないからです。

値ヒストリの最後の値を異なる形式で再表示するには、`print` コマンドに対して式を指定せずにフォーマットだけを指定して実行します。例えば、‘`p/x`’を実行すると最後の値を 16 進で再表示します。

<sup>1</sup> 原注：フォーマット文字 ‘b’ は使用できません。フォーマット文字は `x` コマンドでも共通して使用されますが、`x` コマンドでは、‘b’ は byte の省略形として使用されているためです。See Section 8.5 [Examining memory], page 63。



## 8.5 メモリの調査

コマンド `x` ( `examine` の `x` ) を使用することで、ユーザ・プログラム内のデータ型にかかわらず、メモリ上の値をいくつかの形式で調べることができます。

`x/nfu addr`

`x addr`

`x`                   メモリ上の値を調べるには `x` コマンドを使用してください。

`n`、`f`、`u` はいずれも、どれだけのメモリをどのようにフォーマットして表示するかを指定するための、必須ではないパラメータです。`addr` は、メモリの表示を開始するアドレスを指定する式です。`nfu` の部分にデフォルトを使用するのであれば、スラッシュ '/' は必要ありません。いくつかのコマンドによって、`addr` に対して便利なデフォルト値を指定することができます。

`n` ( 繰り返し回数 )

繰り返し回数は 10 進の整数値です。デフォルトは 1 です。これによって、( 単位 `u` の ) メモリをどれだけ表示するかを指定します。

`f` ( 表示フォーマット )

表示フォーマットには、`print` コマンドによって使用されるフォーマット、`'s'` ( NULL 文字で終了する文字列 )、`'i'` ( マシン命令 ) のいずれかを指定します。初期状態では、デフォルトは `'x'` ( 16 進 ) です。デフォルトは、`x` コマンドまたは `print` コマンドを実行するたびに変更されます。

`u` ( メモリ・サイズの単位 )

単位の大きさは以下のいずれかになります。

`b`                   バイト

`h`                   ハーフ・ワード ( 2 バイト )

`w`                   ワード ( 4 バイト ) —これが初期状態のデフォルトです。

`g`                   ジャイアント・ワード ( 8 バイト )

`x` コマンド実行時に単位の大きさを指定するたびに、その大きさが、次に `x` コマンドを実行する際のデフォルトになります ( フォーマット `'s'` および `'i'` については、単位の大きさは無視されます。これらについては、通常、単位の大きさを指定しません )。

`addr` ( 表示を開始するアドレス )

`addr` は、GDB にメモリの表示を開始させたいアドレスです。この式は、必ずしもポインタ値を持つ必要はありません ( ポインタ値を持つことも可能です )。これは常に、メモリ内のある 1 バイトを指す整数値のアドレスとして解釈されます。式に関する詳細については、See Section 8.1 [Expressions], page 59。 `addr` のデフォルトは通常、最後に調べられたアドレスの次のアドレスになります。しかし、ほかのコマンドによってもデフォルトのアドレスが設定されます。該当するコマンドは、`info breakpoints` ( デフォルトは、最後に表示されたブレイクポイントのアドレスに設定されます )、`info line` ( デフォルトは、行の先頭アドレスに設定されます ) および `print` コマンド ( メモリ内の値を表示するのに使用した場合 ) です。

例えば、`'x/3uh 0x54320'` は、先頭アドレス `0x54320` から始めて、メモリ上の 3 個のハーフ・ワード ( `h` ) の値を、符号なし 10 進整数値 ( `'u'` ) としてフォーマットして表示するよう求める要求です。また、`'x/4xw $sp'` は、スタック・ポインタ ( `'$sp'` については、see Section 8.10 [Registers], page 72 ) の上位 4 ワード ( `'w'` ) のメモリの内容を 16 進 ( `'x'` ) で表示します。

単位の大きさを示す文字と出力フォーマットを指定する文字とは異なるので、単位の大きさとフォーマットのどちらが前にくるべきかを記憶しておく必要はありません。どちらを先に記述しても動作します。‘4xw’という出力指定と‘4wx’という出力指定とは、全く同一の意味を持ちます（ただし、繰り返し回数 *n* は最初に指定しなければなりません。‘wx4’ではうまく動きません）。

単位の大きさ *u* は、フォーマット‘s’および‘i’については無視されますが、繰り返し回数 *n* を使用したいことがあるかもしれません。例えば、‘3i’はオペランドも含めて 3 つのマシン命令を表示したいということを指定しています。disassemble コマンドは、マシン命令を調べる別の方法を提供してくれます。See Section 7.4 [Source and machine code], page 56。

x コマンドへの引数のデフォルトはすべて、x コマンドを使用してメモリ上を連続的に参照するために最少の情報だけを指定すればよいように設計されています。例えば、‘x/3i addr’によってマシン命令を調べた後、‘x/7’とするだけで、続く 7 個のマシン命令を調べることができます。(RET) キーによって x コマンドを繰り返し実行する場合は、前回の繰り返し回数 *n* が再度使用されます。その他の引数も、後続の x コマンド使用時のデフォルトになります。

x コマンドによって表示されるアドレスや内容は、値ヒストリに保存されません。これらの数がしばしば膨大になり、邪魔になるからです。その代わりに GDB は、これらの値をコンビニエンス変数 \$\_<sub>0</sub> および \$\_<sub>1</sub> の値として、後続の式の内部で使えるようにします。x コマンドを実行後、最後に調べられたアドレスは、コンビニエンス変数 \$\_<sub>0</sub> の値として式の中で使用することができます。また、GDB によって調べられたそのアドレスの内容は、コンビニエンス変数 \$\_<sub>1</sub> の値として使用可能です。

x コマンドに繰り返し回数が指定されている場合、保存されるのは、最後に表示されたメモリ単位のアドレスとその内容です。これは、最後の出力行にいくつかのメモリ単位が表示されている場合は、最後に表示されたアドレス値と一致しません。

## 8.6 自動表示

ある 1 つの式の値を（それがどのように変化するかを見るために）頻繁に表示したい場合は、その式を自動表示リストに加えて、ユーザ・プログラムが停止するたびに、GDB がその値を表示するようにするとよいでしょう。リストに加えられた個々の式には、それを識別するための番号が割り当てられます。ある式をリストから削除する際に、その番号を指定します。自動表示は、例えば以下のように表示されます。

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

ここでは、項目番号、式、および、その式の現在の値が表示されます。x コマンドや print コマンドによって手動で表示を要求する場合と同様、好みの出力フォーマットを指定することができます。実は、display コマンドは、ユーザのフォーマットの指定の詳細度によって、print コマンドと x コマンドのいずれを使用するかを決定しています。単位の大きさが指定された場合や、x コマンドでしかサポートされていない 2 つのフォーマット（‘i’と‘s’）のいずれかが指定された場合には、x コマンドが使用されます。それ以外の場合は、print コマンドが使用されます。

display exp

ユーザ・プログラムが停止するたびに表示される式のリストに、式 *exp* を追加します。See Section 8.1 [Expressions], page 59。

コマンドの実行後に (RET) キーを押しても、display コマンドは繰り返し実行されません。

display/fmt exp

*fmt* の部分に、大きさや繰り返し回数は指定せず、出力フォーマットだけを指定した場合は、式 *exp* を自動表示リストに追加して、出力時のフォーマットが常に、指定されたフォーマット *fmt* になるよう調整します。See Section 8.4 [Output formats], page 62。

`display/fmt addr`

`fmt` の部分に 'i'、's' を指定した場合、あるいは、単位の大きさ、単位の数を指定した場合は、ユーザ・プログラムが停止するたびに調べるメモリ・アドレスとして式 `addr` を追加します。ここで「調べる」というのは、実際には '`x/fmt addr`' を実行することを意味します。See Section 8.5 [Examining memory], page 63。

例えば、'`display/i $pc`' は、ユーザ・プログラムが停止するたびに、次に実行されるマシン命令を見るのに便利です ('`$pc`' は、プログラム・カウンタを指すのに一般に使用される名前です。see Section 8.10 [Registers], page 72 )

`undisplay dnums...`

`delete display dnums...`

表示すべき式のリストから、項目番号 `dnums` に対応する要素を削除します。

`undisplay` コマンドを実行後に `(RET)` キーを押しても、コマンドは再実行されません ( 仮に再実行されてしまうとすると、'No display number ...' というエラーになるだけです )

`disable display dnums...`

項目番号 `dnums` の表示を不可にします。表示不可にされた表示項目は自動的に表示されませんが、削除されたわけではありません。後に、表示可能にすることができます。

`enable display dnums...`

項目番号 `dnums` の表示を可能にします。これにより、表示不可が指定されるまで、式の自動表示が再度有効になります。

`display` リスト上の式のカレントな値を表示します。これは、ユーザ・プログラムが停止したときに実行されるのと同じ処理です。

`info display`

自動的に表示されるよう設定された式のリストを表示します。個々の式の項目番号は表示されますが、値は表示されません。このリストには、表示不可になっている式も含まれ、そのことが分かるようにマーク付けされています。また、表示されるリストには、その時点ではアクセスできない自動変数を参照しているために、その時点では値を表示することのできない式も含まれます。

表示される式がローカル変数への参照を含む場合、そのローカル変数がセットアップされているコンテキストの範囲外では、その式は無意味です。このような式は、その中の変数の 1 つでも定義されないコンテキストが実行開始されると表示不可になります。例えば、引数 `last_char` を取る関数の内部で `display last_char` コマンドを実行すると、その関数の内部でユーザ・プログラムが実行を停止し続ける間は、GDB はこの引数を表示します。ほかの箇所 ( `last_char` という変数が存在しない箇所 ) で停止したときには、自動的に表示不可となります。次にユーザ・プログラムが `last_char` が意味を持つ箇所では、再びその式の表示を可能にすることができます。

## 8.7 表示設定

GDB は、配列、構造体、シンボルをどのように表示するかを制御するための方法を提供しています。

これらの設定は、どのプログラミング言語で記述されたプログラムのデバッグにも便利です。

```
set print address
```

```
set print address on
```

これにより GDB は、メモリ・アドレスの内容を表示する場合でも、スタック・トレース、構造体の値、ポインタの値、ブレイクポイントなどの位置を示すアドレスを表示します。デフォルトは on です。例として、set print address on のときのスタック・フレームの表示結果を示します。

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530          if (lquote != def_lquote)
```

```
set print address off
```

アドレスの内容を表示するときには、そのアドレスを表示しません。例えば、set print address off のときに前の例と同一のスタック・フレームを表示すると、以下のようになります。

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530          if (lquote != def_lquote)
```

‘set print address off’を使用することで、GDB のインターフェイスからマシンに依存する表示を取り除くことができます。例えば、print address off を指定してあれば、ポインタ引数の有無にかかわらず、すべてのマシン上において同一のバックトレース情報を得るはずで。

```
show print address
```

アドレスが表示されるか否かを示します。

GDB がシンボリックなアドレスを表示する際には通常、そのアドレスの前にある最も近い位置のシンボルと、そのシンボルからのオフセットを表示します。そのシンボルによってアドレスが一意に決まらない場合（例えば、単一のファイル内でのみ有効な名前である場合）には、確認の必要があるかもしれません。1 つの方法は、例えば ‘info line \*0x4537’ のように、info line コマンドを実行することです。または、シンボリックなアドレスを表示するときに、一緒にソース・ファイルや行番号を表示するよう GDB を設定する方法もあります。

```
set print symbol-filename on
```

シンボリックな形式のアドレスの表示において、そのシンボルのソース・ファイル名と行番号を表示するよう GDB に通知します。

```
set print symbol-filename off
```

シンボルのソース・ファイル名と行番号を表示しません。これがデフォルトです。

```
show print symbol-filename
```

シンボリックな形式でのアドレス表示において、GDB がそのシンボルのソース・ファイル名と行番号を表示するか否かを示します。

シンボルのソース・ファイル名と行番号を表示するのが役に立つもう 1 つの状況として、コードを逆アセンブルする場合があります。GDB が、個々の命令に対応する行番号とソース・ファイルを表示してくれます。

また、アドレスをシンボリック形式で表示させるのは、そのアドレスと、そのアドレスより前にあるシンボルのうち、そのアドレスに最も近い位置にあるものとの間が適度に接近している場合に限定させたいこともあるかもしれません。

```
set print max-symbolic-offset max-offset
```

アドレスと、そのアドレスより前にある最も近いシンボルの間のオフセットが *max-offset* 未満のときのみ、そのアドレスをシンボリックな形式で表示するよう GDB に通知します。デフォルトは 0 で、これは GDB に対して、アドレスより前にシンボルがある場合には、常にそのアドレスをシンボリックな形式で表示するよう通知します。

```
show print max-symbolic-offset
```

GDB がシンボリックなアドレスを表示する上限となる、最大のオフセット値を問い合わせます。

あるポインタがどこを指しているか定かではない場合には、`'set print symbol-filename on'` を試みてください。こうすれば、`'p/a pointer'` を使用して、そのポインタが指している変数の名前とソース・ファイル上の位置が分かります。これは、アドレスをシンボリック形式で解釈します。例えば以下の例では、ある変数 `ptt` がファイル `'hi2.c'` 内で定義された別の変数 `t` を指していることを、`valueGDBN` が教えてくれています。

```
(gdb) set print symbol-filename on
(gdb) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

注意: ローカル変数を指すポインタについては、たとえ適切な `set print` オプションが有効になっていても、`'p/a'` はそのポインタによって参照される変数のシンボル名やファイル名を表示しません。

異なる種類のオブジェクトについては、他の設定によって表示方法が制御されます。

```
set print array
```

```
set print array on
```

配列をきれいに表示します。このフォーマットは読むのには便利ですが、より多くのスペースを取ります。デフォルトは `'off'` です。

```
set print array off
```

配列を詰め込み形式で表示します。

```
show print array
```

配列の表示方法として、詰め込み形式ときれいな形式のどちらが選択されているかを示します。

```
set print elements number-of-elements
```

GDB によって表示される配列の要素の数に上限を設定します。GDB が大きな配列を表示している際に、表示された要素の数が `set print elements` コマンドで設定された数に達すると、そこで表示が停止されます。この上限は、文字列の表示にも適用されます。*number-of-elements* に 0 をセットすると、要素は無制限に表示されます。

```
show print elements
```

大きな配列を表示する際に GDB が表示する要素数を示します。0 の場合、表示される要素数に制限はありません。

```
set print null-stop
```

最初に NULL が検出された時点で、GDB に文字配列の表示を停止させます。これは、大きな配列が実際には短い文字列しか含んでいないときに役に立ちます。

```
set print pretty on
```

構造体を表示する際に、インデントされた形式で 1 行に 1 メンバずつ GDB に表示させます。以下に例を示します。

```

$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}

```

`set print pretty off`

構造体を詰め込み形式で GDB に表示させます。以下に例を示します。

```

$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, \
  meat = 0x54 "Pork"}

```

これがデフォルトの形式です。

`show print pretty`

GDB が、構造体を表示するのにどちらの形式を使用しているかを示します。

`set print sevenbit-strings on`

7 ビット文字だけを使用して表示します。このオプションがセットされていると、GDB は ( 文字列内または単一文字内の ) 8 ビット文字を `\nnn` という表記法で表示します。この設定は、英語 ( ASCII ) 環境において、文字の最上位ビットをマークや「メタ」ビットとして使用する場合に最適です。

`set print sevenbit-strings off`

8 ビット文字を表示します。これにより文字セットの使用が国際的になります。これがデフォルトです。

`show print sevenbit-strings`

GDB が 7 ビット文字だけを表示するか否かを示します。

`set print union on`

GDB に対して、構造体の中に含まれている共用体を表示するよう通知します。これが、デフォルトの設定です。

`set print union off`

GDB に対して、構造体の中に含まれている共用体を表示しないよう通知します。

`show print union`

GDB に対して、構造体の中に含まれている共用体を表示するか否かを問い合わせます。

例えば、以下のように宣言されている場合、

```

typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
    Bug_forms;

struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
}

```

```
};

struct thing foo = {Tree, {Acorn}};
set print union on
```

が有効な場合、‘p foo’は以下のような表示を行います。

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

また、set print union offが有効な場合、‘p foo’は以下のような表示を行います。

```
$1 = {it = Tree, form = {...}}
```

以下の設定は、C++プログラムをデバッグしているときに関係があります。

```
set print demangle
set print demangle on
```

C++のシンボル名を、型セーフ ( type-safe ) なリンクのためにアセンブラ、リンカに渡されるエンコードされた ( mangled ) 形式ではなく、ソースに記述された形式で表示します。デフォルトは ‘on’ です。

```
show print demangle
```

C++のシンボル名が、エンコードされた ( mangled ) 形式、ソース ( demangled ) 形式のいずれの形式で表示されるかを示します。

```
set print asm-demangle
set print asm-demangle on
```

C++のシンボル名を、命令の逆アセンブル時のようにアセンブラ・コードで表示しているときにも、エンコードされた ( mangled ) 形式ではなく、ソース形式で表示します。デフォルトは ‘off’ です。

```
show print asm-demangle
```

アセンブラ・コードの表示において、C++シンボル名をエンコードされた ( mangled ) 形式、ソース ( demangled ) 形式のいずれの形式で表示するかを示します。

```
set demangle-style style
```

C++シンボル名を表現するために様々なコンパイラによって使用されるいくつかのエンコーディング方式の中から1つを選択します。現在 *style* として選択可能であるのは、以下のとおりです。

|       |                                                                                                                                                          |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| auto  | GDB がユーザ・プログラムを解析してデコーディング方式を決定することを許します。                                                                                                                |
| gnu   | GNU C++ ( g++ ) エンコーディング・アルゴリズムに基づいてデコードします。これが、デフォルトです。                                                                                                 |
| hp    | HP ANSI C++ ( aCC ) エンコーディング・アルゴリズムに基づいてデコードします。                                                                                                         |
| lucid | Lucid C++ ( lcc ) エンコーディング・アルゴリズムに基づいてデコードします。                                                                                                           |
| arm   | <i>C++ Annotated Reference Manual</i> に記述されているアルゴリズムを使用してデコードします。注意: この設定だけでは、cfrontによって生成された実行モジュールをデバッグするのに十分ではありません。これを可能にするためには、GDB をさらに拡張する必要があります。 |

*style* を指定しないと、指定可能なフォーマットの一覧が表示されます。

`show demangle-style`

C++シンボルをデコードするのに現在使用されているエンコーディング方式を示します。

`set print object`

`set print object on`

オブジェクトへのポインタを表示する際に、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（派生された）型を表示します。

`set print object off`

仮想関数テーブルは参照せず、オブジェクトの宣言された型だけを表示します。これがデフォルトの設定です。

`show print object`

オブジェクトの実際の型と宣言された型のどちらが表示されるかを示します。

`set print static-members`

`set print static-members on`

C++のオブジェクトを表示する際、静的メンバを表示します。デフォルトは‘on’です。

`set print static-members off`

C++のオブジェクトを表示する際、静的メンバを表示しません。

`show print static-members`

C++の静的メンバが表示されるか否かを示します。

`set print vtbl`

`set print vtbl on`

C++の仮想関数テーブルをきれいな形式で表示します。デフォルトは‘off’です。

`set print vtbl off`

C++の仮想関数テーブルをきれいな形式で表示しません。

`show print vtbl`

C++の仮想関数テーブルをきれいな形式で表示するか否かを示します。

## 8.8 値ヒストリ

`print`コマンドにより表示された値は、GDBの値ヒストリに保存されます。これによりユーザは、これらの値をほかの式の中で参照することができます。値は、シンボル・テーブルが（例えば、`file`コマンドや `symbol-file`コマンドにより）再読み込みされるか破棄されるまで、維持されます。シンボル・テーブルが変更されると、値ヒストリが破棄されるのは、その中の値が、シンボル・テーブル内で定義されている型を参照しているかもしれないからです。

表示される値はヒストリ番号を与えられ、この番号によって参照することができます。この番号は1から始まる連続した整数です。`print`コマンドは、値に割り当てられたヒストリ番号を、値の前に‘`$num =`’という形で表示します。ここで、`num`がそのヒストリ番号です。

値ヒストリの中の任意の値を参照するには、‘\$’に続けてヒストリ番号を指定します。`print`コマンドが出力に付加するラベルは、ユーザにこのことを知らせるためのものです。`$`単体では、ヒストリ内の最も新しい値を参照し、`$$`はその1つ前の値を参照します。`$$n`は、最新のものから数えて`n`番目の値を参照します。`$$2`は`$$`の1つ前の値を参照し、`$$1`は`$$`と同一、`$$0`は`$`と同一です。

例えば、ユーザがたった今、構造体へのポインタを表示し、今度はその構造体の内容を見たいと考えているとしましょう。この場合は、



```
p *$
```

を実行すれば十分です。

また、連結された構造体があり、そのメンバの `next` が次の構造体を指すポインタであるとする、次の構造体の内容を表示するには、

```
p *$.next
```

とします。

このように連結された構造体を次々に表示するには、このコマンドを繰り返し実行すればよく、それは `(RET)` キーによって可能です。

このヒストリは、式ではなく、値を記録するという点に注意してください。x の値が 4 のときに、以下のコマンドを実行すると、`print` コマンドによって値ヒストリに記録される値は、x の値が変化したにもかかわらず 4 のままです。

```
print x
set x=5
```

```
show values
```

値ヒストリ内の最新の 10 個の値を、項目番号付きで表示します。これは、`'p $9'` を 10 回実行するようなものですが、両者の違いは、`show values` がヒストリを変更しないという点にあります。

```
show values n
```

値ヒストリ内の項目番号 `n` を中心に、その前後の 10 個の値を表示します。

```
show values +
```

値ヒストリ内の値のうち最後に表示されたものの直後にある 10 個の値を表示します。値が存在しない場合には、何も表示されません。

`show values n` を繰り返し実行するのに `(RET)` キーを押すことは、`'show values +'` を実行するのと全く同じ結果をもたらします。

## 8.9 コンビニエンス変数

GDB のコンビニエンス変数は、GDB 中にある値を保持しておいて、それを後に参照するという目的で使うことができます。これらの変数は、GDB 内部においてのみ存在するものです。それらはユーザ・プログラムの中に存在するものではなく、コンビニエンス変数を設定してもユーザ・プログラムの実行には直接影響を与えません。したがって、ユーザはこれを自由に使うことができます。

コンビニエンス変数名は、先頭が `'$'` で始まります。`'$'` で始まる名前は、あらかじめ定義されたマシン固有のレジスタ名 (see Section 8.10 [Registers], page 72) と一致しない限り、コンビニエンス変数の名前として使うことができます (これに対して、値ヒストリの参照名では `'$'` に続けて番号を記述します。See Section 8.8 [Value history], page 70)。

ユーザ・プログラムの中で変数に値を設定すると同じように、代入式を使用してコンビニエンス変数に値を保存することができます。例えば、`object_ptr` が指すオブジェクトが保持する値を `$foo` に保存するには、以下のようにします。

```
set $foo = *object_ptr
```

コンビニエンス変数は、最初に使用されたときに生成されますが、新しい値を割り当てるまで、その値は空 (void) です。値は、いつでも代入することによって変更可能です。

コンビニエンス変数には決まった型はありません。コンビニエンス変数には、既に異なる型のデータが割り当てられている場合でも、構造体や配列を含めた任意の型のデータを割り当てることができます。コンビニエンス変数は、式として使用される場合には、その時点における値の型を持ちます。

`show convenience`

それまでに使用されたコンビニエンス変数とその値の一覧を表示します。省略形は、`show con`です。

コンビニエンス変数の1つの使い方に、インクリメントされるカウンタや先へ進んでいくポインタとしての使い方があります。例えば、構造体配列の中の連続する要素のあるフィールドの値を表示したい場合、以下のコマンドを(RET)キーで繰り返し実行します。

```
set $i = 0
print bar[$i++]>contents
```

GDBによって、いくつかのコンビニエンス変数が自動的に作成され、役に立ちそうな値が設定されます。

`$_`      `$_`変数には、`x`コマンドによって最後に調べられたアドレスが自動的に設定されます( see Section 8.5 [Examining memory], page 63 )。 `x`コマンドによって調べられるデフォルトのアドレスを提供する他のコマンドも、`$_`にそのアドレスを設定します。このようなコマンドには、`info line`や `info breakpoint`があります。`$_`の型は、`x`コマンドによって設定された場合は`$_`の型へのポインタであり、それ以外の場合は `void *`です。

`$__`      `$__`変数には、`x`コマンドによって最後に調べられたアドレス位置にある値が自動的に設定されます。型は、データが表示されたフォーマットに適合するように選択されます。

`$_exitcode`

`$_exitcode`変数には、デバッグされているプログラムが終了した際の終了コードが自動的に設定されます。

## 8.10 レジスタ

マシン・レジスタの内容は、先頭が '\$' で始まる名前を持つ変数として、式の中で参照することができます。レジスタの名前は、マシンによって異なります。`info registers` コマンドを使用することで、そのマシンで使用されているレジスタの名前を知ることができます。

`info registers`

( 選択されたスタック・フレームにおける ) 浮動小数点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。

`info all-registers`

浮動小数点レジスタも含めてすべてのレジスタの名前と値を表示します。

`info registers regname ...`

指定されたレジスタ `regname` の相対化された値 ( *relativized value* ) を表示します。以下に詳しく述べるように、レジスタの値は、通常は、選択されたスタック・フレームと関係を持つ相対的な値です。`regname` には、ユーザの使用しているマシン上において有効な任意のレジスタの値が設定可能です。先頭の '\$' は、あってもなくてもかまいません。

GDB は、そのマシン・アーキテクチャが持つレジスタの正規のニーモニックと衝突しない限り、ほとんどのマシン上（の式の中）において利用可能な、4 つの「標準的」なレジスタ名を持っています。レジスタ名 `$pc` と `$sp` は、プログラム・カウンタ・レジスタとスタック・ポインタを指すために使われます。`$fp` は、カレントなスタック・フレームへのポインタを保持するレジスタを指すために使われます。`$ps` は、プロセッサの状態を保持するレジスタを指すために使われます。例えば、プログラム・カウンタの値を 16 進数で表示するには、以下のように実行します。

```
p/x $pc
```

また、次に実行される命令を表示するには、以下のように実行します。

```
x/i $pc
```

さらに、スタック・ポインタ<sup>2</sup> に 4 を加えるには、以下のように実行します。

```
set $sp += 4
```

可能な場合にはいつでも、これら 4 つの標準的なレジスタ名が使用可能です。ユーザのマシンが異なる正規のニーモニックを使用している場合でも、名前の衝突さえ起こらなければ、使用可能です。info registers コマンドにより、正規名を見ることができます。例えば、SPARC 上で info registers コマンドを実行すると、プロセッサ・ステータス・レジスタは `$psr` と表示されますが、このレジスタを `$ps` として参照することもできます。

レジスタがこの方法で調べられるとき、GDB は普通のレジスタの内容を常に整数値とみなします。マシンによっては、浮動小数点値以外を保持できないレジスタを持つものがあります。このようなレジスタは、浮動小数点値を持つものとみなされます。普通のレジスタの内容を浮動小数点値として参照する方法はありません（`'print/f $regname'` により、浮動小数点値として値を表示することはできます）。

レジスタには、raw と virtual の 2 つの異なるデータ形式を取るものがあります。これは、オペレーティング・システムによってレジスタの内容が保存されるときにデータ形式が、ユーザ・プログラムが通常認識しているものと同じではないことを意味しています。例えば、68881 浮動小数点コプロセッサのレジスタの値は常に extended（raw）形式で保存されていますが、C 言語によるプログラムは通常 double（virtual）形式を想定しています。このような場合、GDB は通常（ユーザ・プログラムにとって意味のある形式である）virtual 形式だけを扱いますが、info registers コマンドはデータを両方の形式で表示してくれます。

通常、レジスタの値は、選択されたスタック・フレーム（see Section 6.3 [Selecting a frame], page 49）と関係を持つ相対的な値です。これは、ユーザにレジスタの値として見えるものは、選択されたフレームから呼び出されているすべてのスタック・フレームが終了し、退避されたレジスタの値が復元されたときに、そのレジスタが持つであろう値です。ハードウェア・レジスタの本当の値を知りたい場合は、最下位のフレームを（`'frame 0'` で）選択しなければなりません。

しかし、GDB は、コンパイラが生成したコードから、どこにレジスタが保存されているかを推論する必要があります。退避されていないレジスタがある場合や、GDB が退避されたレジスタを見つけないことができない場合は、どのスタック・フレームを選択していても結果は同じです。

---

<sup>2</sup> 原注：これは、スタックがメモリの下位方向に伸長するマシン（最近のほとんどのマシンがそうです）上において、スタックから 1 ワードを取り除く方法です。これは、最下位のスタック・フレームが選択されていることを想定しています。これ以外のスタック・フレームが選択されているときには、`$sp` に値を設定することは許されません。マシン・アーキテクチャに依存することなくスタックからフレーム全体を取り除くには、return を使用します。See Section 11.4 [Returning from a function], page 97。

```
set rstack_high_address address
```

AMD 29000 ファミリ・プロセッサでは、レジスタは「レジスタ・スタック」と呼ばれるところに退避されます。GDB には、このスタックの大きさを知ることはできません。通常 GDB は、スタックは十分に大きいと想定します。このために、実際には存在しないメモリ位置を、GDB が参照してしまうことがあります。必要であれば、`set rstack_high_address` コマンドによってレジスタ・スタックの最終アドレスを指定することによって、この問題を回避することができます。引数はアドレスでなければなりません。`'0x'` を先頭に記述することで、アドレスを 16 進数で指定することができます。

```
show rstack_high_address
```

AMD 29000 ファミリ・プロセッサにおけるレジスタ・スタックのカレントな上限を表示します。

### 8.11 浮動小数ハードウェア

構成によっては、GDB は浮動小数ハードウェアの状態について、より詳しい情報を提供することができます。

```
info float
```

浮動小数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。浮動小数チップの種類によって、表示内容やレイアウトは変わります。現在、`'info float'` は ARM マシンと x86 マシンにおいてサポートされています。

## 9 異なる言語の使用

異なるプログラミング言語であっても共通点があるのが普通ですが、その表記法が全く同様であるということはめったにありません。例えば、ポインタ `p` の指す値を取り出す方法は、ANSI C では `*p` ですが、Modula-2 では `p^` です。値の表現方法（および表示方法）もまた異なります。16 進数は、C では `'0x1ae'` のようになりますが、Modula-2 では `'1AEH'` のようになります。

いくつかの言語については、言語固有の情報が GDB に組み込まれており、これにより、プログラムを記述した言語を使って上記のような操作を記述したり、プログラムを記述した言語の構文にしたがって GDB に値を出力させることができます。式を記述するのに使用される言語を、作業言語と呼びます。

### 9.1 ソース言語の切り替え

作業言語を制御する方法は 2 つあります。GDB に自動的に設定させる方法と、ユーザが手作業で選択する方法です。どちらの目的でも、`set language` コマンドを使用することができます。起動時のデフォルトでは、GDB が言語を自動的に設定するようになっています。作業言語は、ユーザの入力する式がどのように解釈されるか、あるいは、値がどのように表示されるかを決定します。

この作業言語とは別に、GDB の認識しているすべてのソース・ファイルには、それ自体の作業言語があります。オブジェクト・ファイルのフォーマットによっては、ソース・ファイルの記述言語を示す情報を、コンパイラが書き込んでいることがあるかもしれません。しかし、ほとんどの場合、GDB はファイル名から言語を推定します。ソース・ファイルの言語の種類が、C++ シンボル名がデコード（demangle）されるか否かを制御します。これにより `backtrace` は、個々のフレームを、その対応する言語にしたがって適切に表示することができます。GDB の中から、ソース・ファイルの言語を設定することはできません。

他の言語で記述されたソースから C のソースを生成する、`cfront` や `f2c` のようなプログラムをユーザが使用する場合には、このことが問題となるでしょう。このような場合には、生成される C の出力に `#line` 指示子を使用するよう、そのプログラムを設定してください。こうすることによって、GDB は、元になったプログラムのソース・コードが記述された言語を正しく知ることができ、生成された C のコードではなく、元になったソース・コードを表示します。

#### 9.1.1 ファイル拡張子と言語のリスト

ソース・ファイル名が以下のいずれかの拡張子を持つ場合、GDB はその言語を以下に示すものと推定します。

|                       |                  |
|-----------------------|------------------|
| <code>' .c '</code>   | C ソース・ファイル       |
| <code>' .C '</code>   |                  |
| <code>' .cc '</code>  |                  |
| <code>' .cp '</code>  |                  |
| <code>' .cpp '</code> |                  |
| <code>' .cxx '</code> |                  |
| <code>' .c++ '</code> | C++ ソース・ファイル     |
| <code>' .f '</code>   |                  |
| <code>' .F '</code>   | Fortran ソース・ファイル |

```

‘.ch’
‘.c186’
‘.c286’    CHILL ソース・ファイル

‘.mod’    Modula-2 ソース・ファイル

‘.s’
‘.S’      アセンブラ言語のソース・ファイル。この場合、実際の動作はほとんど C 言語と同様ですが、ステップ実行時に、関数呼び出しのための事前処理部を GDB はスキップしません。

```

さらに、言語に対してファイル名の拡張子を関連付けすることも可能です。See Section 9.2 [Displaying the language], page 77。

### 9.1.2 作業言語の設定

GDB に言語を自動的に設定させる場合、ユーザのデバッグ・セッションとユーザのプログラムにおいて、式は同様に解釈されます。

もしそうしなければ、言語を手作業で設定することもできます。そのためには、コマンド ‘set language lang’ を実行します。ここで、*lang* は、c や modula-2 のような言語名です。サポートされている言語のリストは、‘set language’ で表示させることができます。

言語を手作業で設定すると、GDB は、作業言語を自動的に更新することができなくなります。このことは、作業言語がソースの言語と同一ではなく、かつ、ある式がどちらの言語でも有効でありながら、その意味が異なるような状況でプログラムをデバッグしようとしたときに、混乱をもたらす可能性があります。例えば、カレントなソース・ファイルが C 言語で記述されていて、GDB がそれを Modula-2 として解析している場合に、

```
print a = b + c
```

のようなコマンドを実行すると、その結果は意図したものとは異なるものになるでしょう。これは C 言語では、b と c とを加算して、その結果を a に入れるということを意味し、表示される結果は、a の値となります。Modula-2 では、これは a と b+c の結果を比較して BOOLEAN 型の値を出力することを意味します。

### 9.1.3 GDB によるソース言語の推定

GDB に作業言語を自動的に設定させるには、‘set language local’ または ‘set language auto’ を使用します。この場合、GDB は作業言語を推定します。つまり、ユーザ・プログラムが（通常はブレークポイントに達することによって）あるフレーム内部で停止したとき、GDB は、そのフレーム内の関数に対して記録されている言語を作業言語として設定します。フレームの言語が不明の場合（つまり、そのフレームに対応する関数またはブロックが、既知ではない拡張子を持つソース・ファイルにおいて定義されている場合）カレントな作業言語は変更されず、GDB は警告メッセージを出力します。

このようなことは、全体がただ 1 つの言語で記述されているほとんどのプログラムにおいては不要であると思われるでしょう。しかし、あるソース言語で記述されたプログラム・モジュールやライブラリは、他のソース言語で記述されたメイン・プログラムから使用することができます。このような場合に ‘set language auto’ を使用することで、作業言語を手作業で設定する必要がなくなります。

## 9.2 言語の表示

以下のコマンドは、作業言語、および、ソース・ファイルの記述言語を知りたいときに役に立ちます。

`show language`

カレントな作業言語を表示します。printコマンドなどによってユーザ・プログラム内部の変数を含む式を作成したり評価したりするには、このコマンドによって示される言語を使用します。

`info frame`

選択されているフレームのソース言語を表示します。このフレームの中の識別子を使用すると、この言語が作業言語になります。このコマンドにより表示される他の情報について知りたい場合は、See Section 6.4 [Information about a frame], page 50。

`info source`

選択されているソース・ファイルのソース言語を表示します。このコマンドにより表示される他の情報のことを知りたい場合は、See Chapter 10 [Examining the Symbol Table], page 91。

普通ではない状況においては、標準のリストに含まれない拡張子を持つソース・ファイルがあるかもしれません。この場合には、その拡張子を特定の言語に明示的に関連付けすることができます。

`set extension-language .ext language`

拡張子`.ext`を持つソース・ファイルは、ソース言語 *language* によって記述されているものと想定するよう設定します。

`info extensions`

すべてのファイル拡張子と、その拡張子に関連付けされた言語を一覧表示します。

## 9.3 型と範囲のチェック

注意: 現在のリリースでは、型チェックと範囲チェックを行う GDB コマンドは組み込まれていますが、それらは実際には何も実行しません。このセクションでは、これらのコマンドが本来持つべく意図されている機能について記述してあります。

いくつかの言語は、一連のコンパイル時チェック、実行時チェックによって、一般によく見られるエラーの発生を防ぐように設計されています。これらのチェックには、関数や演算子への引数の型のチェックや、数学的操作の結果のオーバーフローを実行時に確実に検出することなどが含まれています。このようなチェックは、型の不一致を排除したり、ユーザ・プログラムの実行時に範囲エラーをチェックしたりすることによって、コンパイル後のプログラムの正しさを確かなものにするのに役に立ちます。

GDB は、ユーザが望むのであれば、上記のような条件のチェックを行います。GDB はユーザ・プログラムの文をチェックすることはしませんが、例えば、printコマンドによる評価を目的として GDB に直接入力された式をチェックすることはできます。作業言語の場合と同様に、GDB が自動的にチェックを行うか否かを、ユーザ・プログラムのソース言語によって決定することもできます。サポートされている言語のデフォルトの設定については、See Section 9.4 [Supported languages], page 80。

### 9.3.1 型チェックの概要

いくつかの言語、例えば Modula-2 などは、強く型付けされています。これは、演算子や関数への引数は正しい型でなくてはならず、そうでない場合にはエラーが発生するということを意味しています。このようなチェックは、型の不一致のエラーが実行時に問題を発生させるのを防いでくれます。例えば、 $1+2$  は

$$1 + 2 \Rightarrow 3$$

ですが、 $1+2.3$  は

error  $1 + 2.3$

のようにエラーになります。

第 2 の例がエラーになるのは、CARDINAL 型の 1 は REAL 型の 2.3 と型の互換性がないからです。

GDB コマンドの中で使われる式については、ユーザが GDB の型チェック機能に対して、以下のよう指示を出すことができます。

- チェックを行わない
- あらゆる不一致をエラーとして扱い、式を破棄する
- 型の不一致が発生したときには警告メッセージを出力するだけで、式の評価を実行する

最後の指示が選択された場合、GDB は上記の第 2 の ( エラー ) 例のような式でも評価しますが、その際には警告メッセージを出力します。

型チェックをしないよう指示した場合でも、型に関係のある原因によって GDB が式の評価ができなくなる場合があります。例えば、GDB は `int` の値と `struct foo` の値を加算する方法を知りません。こうした特定の型エラーは、使用されている言語に起因するものではなく、この例のように、そもそも評価することが意味をなさないような式に起因するものです。

個々の言語は、それが型に関してどの程度厳密であるかを定義しています。例えば、Modula-2 と C はいずれも、算術演算子への引数としては数値を要求します。C では、列挙型とポインタは数値として表わすことができますので、これらは算術演算子への正当な引数となります。特定の言語に関する詳細については、See Section 9.4 [Supported languages], page 80。

GDB は、型チェック機能を制御するためのコマンドをさらにいくつか提供しています。

`set check type auto`

カレントな作業言語に応じて、型チェックを実行する、または、実行しないよう設定します。個々の言語のデフォルトの設定については、See Section 9.4 [Supported languages], page 80。

`set check type on`

`set check type off`

カレントな作業言語のデフォルトの設定を無視して、型チェックを実行する、または、実行しないよう設定します。その設定が言語のデフォルトと一致しない場合は、警告メッセージが出力されます。型チェックを実行するよう設定されているときの式の評価において型の不一致が発生した場合には、GDB はメッセージを出力して式の評価を終了させます。

`set check type warn`

型チェック機能に警告メッセージを出力させますが、式の評価自体は常に実行するよう試みさせます。式の評価は、他の原因のために不可能になる場合もあります。例えば、GDB には数値と構造体の加算はできません。

`show type` 型チェック機能のカレントな設定と、GDB がそれを自動的に設定しているか否かを表示します。



### 9.3.2 範囲チェックの概要

いくつかの言語（例えば、Modula-2）では、型の上限を超えるとエラーになります。このチェックは、実行時に行われます。このような範囲チェックは、計算結果がオーバーフローしたり、配列の要素へのアクセス時に使うインデックスが配列の上限を超えたりすることがないことを確実にすることによって、プログラムの正しさを確かなものにすることを意図したものです。

GDB コマンドの中で使う式については、範囲エラーの扱いを以下のいずれかにするよう GDB に指示することができます。

- 範囲エラーを無視する
- 範囲エラーを常にエラーとして扱い、式を破棄する
- 警告メッセージを出力するだけで、式を評価する

範囲エラーは、数値がオーバーフローした場合、配列インデックスの上限を超えた場合、どの型のメンバでもない定数が入力された場合に発生します。しかし、言語の中には、数値のオーバーフローをエラーとして扱わないものもあります。C 言語の多くの実装では、数学的演算によるオーバーフローは、結果の値を「一巡」させて小さな値にします。例えば、 $m$  が整数値の最大値、 $s$  が整数値の最小値とすると、

$$m + 1 \Rightarrow s$$

になります。

これも個々の言語に固有な性質であり、場合によっては、個々のコンパイラやマシンに固有な性質であることもあります。特定の言語に関する詳細については、See Section 9.4 [Supported languages], page 80。

GDB は、範囲チェック機能を制御するためのコマンドをさらにいくつか提供しています。

`set check range auto`

カレントな作業言語に応じて、範囲チェックを実行する、または、実行しないよう設定します。個々の言語のデフォルトの設定については、See Section 9.4 [Supported languages], page 80。

`set check range on`

`set check range off`

カレントな作業言語のデフォルトの設定を無視して、範囲チェックを実行する、または、実行しないよう設定します。設定が言語のデフォルトとは異なる場合は、警告メッセージが出力されます。範囲エラーが発生した場合は、メッセージが表示され、式の評価は終了させられます。

`set check range warn`

GDB の範囲チェック機能が範囲エラーを検出した場合、メッセージを出力し、式の評価を試みます。例えば、プロセスが、自分の所有していないメモリをアクセスした場合（多くの Unix システムで典型的に見られる例です）など、他の理由によって式の評価が不可能な場合があります。

`show range`

範囲チェック機能のカレントな設定と、それが GDB によって自動的に設定されているのか否かを表示します。

## 9.4 サポートされる言語

GDB は、C、C++、Fortran、Chill、アセンブリ言語、Modula-2 をサポートしています。いくつかの GDB の機能は、使用されている言語にかかわらず、式の中で使用できます。GDB の @ 演算子、:: 演算子、および '{type}addr' ( see Section 8.1 [Expressions], page 59 ) は、サポートされている任意の言語において使用することができます。

次節以降で、個々のソース言語が GDB によってどの程度までサポートされているのかを詳しく説明します。これらの節は、言語についてのチュートリアルやリファレンスとなることを意図したものではありません。むしろ、GDB の式解析機能が受け付ける式や、異なる言語における正しい入出力フォーマットのリファレンス・ガイドとしてのみ役に立つものです。個々の言語については良い書籍が数多く出ています。言語についてのリファレンスやチュートリアルが必要な場合は、これらの書籍を参照してください。

### 9.4.1 C/C++

C と C++ は密接に関連しているので、GDB の機能の多くは両方の言語に適用できます。このようなものについては、2 つの言語を一緒に議論します。

C++ のデバッグ機能は、C++ コンパイラと GDB によって協同で実装されています。したがって、C++ のコードを効率よくデバッグするには、GNU g++、HP ANSI C++ コンパイラ ( aCC ) などの、サポートされている C++ コンパイラで、C++ のプログラムをコンパイルする必要があります。

GNU C++ を使用する場合、最高の結果を引き出すには、stabs デバッグ・フォーマットを使用してください。g++ のコマンドライン・オプション '-gstabs'、または、'-gstabs+' によって、このフォーマットを明示的に選択することができます。詳細については、section "Options for Debugging Your Program or GNU CC" in *Using GNU CC* の部分を参照してください。

#### 9.4.1.1 C/C++ 演算子

演算子は、特定の型の値に対して定義されなければなりません。例えば、+ は数値に対しては定義されていますが、構造体に対しては定義されていません。演算子は、型のグループに対して定義されることがよくあります。

C/C++ に対しては、以下の定義が有効です。

- 整数型には、任意の記憶クラス指定子を持つ int が含まれます。char、enum も整数型です。
- 浮動小数点型には、float と double が含まれます。
- ポインタ型には、型 *type* に対して (*type* \*) により定義されるすべての型が含まれます。
- スカラ型には、上記のすべてが含まれます。

以下の演算子がサポートされています。これらは優先順位の低いものから順に並べられています。

|     |                                                                                                                                    |
|-----|------------------------------------------------------------------------------------------------------------------------------------|
| ,   | カンマ、あるいは、順序付けの演算子です。カンマによって区切られたリストの中の式は、左から右の順で評価されます。最後に評価された式の結果が、式全体の評価結果になります。                                                |
| =   | 代入。代入された値が、代入式の値になります。スカラ型に対して定義されています。                                                                                            |
| op= | $a \text{ op} = b$ という形式の式において使用され、 $a = a \text{ op } b$ に変換されます。op= と = は、同一の優先順位を持ちます。op には、 、^、&、<<、>>、+、-、*、/、% の各演算子が使用できます。 |

|           |                                                                                                                                                                                                                         |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?:        | 3 項演算子です。 $a ? b : c$ は、 $a$ が真であれば $b$ 、偽であれば $c$ とみなすことができます。 $a$ は整数型でなければなりません。                                                                                                                                     |
|           | 論理 OR です。整数型に対して定義されています。                                                                                                                                                                                               |
| &&        | 論理 AND です。整数型に対して定義されています。                                                                                                                                                                                              |
|           | ビットごとの OR です。整数型に対して定義されています。                                                                                                                                                                                           |
| ^         | ビットごとの排他的 OR です。整数型に対して定義されています。                                                                                                                                                                                        |
| &         | ビットごとの AND です。整数型に対して定義されています。                                                                                                                                                                                          |
| ==、!=     | 等価、および、不等価です。スカラ型に対して定義されています。これらの式の値は、偽のときはゼロであり、真のときはゼロ以外の値となります。                                                                                                                                                     |
| <、>、<=、>= | 未満、超過、以下、以上です。スカラ型に対して定義されています。これらの式の値は、偽のときはゼロであり、真のときはゼロ以外の値となります。                                                                                                                                                    |
| <<、>>     | 左シフト、右シフトです。整数型に対して定義されています。                                                                                                                                                                                            |
| @         | GDB の「人工配列」演算子です ( see Section 8.1 [Expressions], page 59 )。                                                                                                                                                            |
| +、-       | 加算および減算です。整数型、浮動小数点型、ポインタ型に対して定義されています。                                                                                                                                                                                 |
| *, /、%    | 乗算、除算、剰余です。乗算と除算は、整数型と浮動小数点型に対して定義されています。剰余は、整数型に対して定義されています。                                                                                                                                                           |
| ++, --    | インクリメント、デクリメントです。変数の前にある場合は、式の中でその変数が使用される前に実行されます。変数の後ろにある場合は、変数の値が使用された後に実行されます。                                                                                                                                      |
| *         | ポインタの間接参照です。ポインタ型に対して定義されています。++ と同一の優先度を持ちます。                                                                                                                                                                          |
| &         | アドレス参照演算子です。変数に対して定義されています。++ と同一の優先順位を持ちます。<br>C++ のデバッグでは、C++ 言語そのものにおいては許されていないような '&' の使用法を、GDB は実装しています。C++ の ( '&ref' ) により宣言される ) 参照変数が格納されているアドレスを調べるのに、 '&(&ref)' (あるいは、もしそうしたいのであれば単に '&&ref' ) を使用することができます。 |
| -         | マイナス ( 負 ) です。整数型と浮動小数点型に対して定義されています。++ と同一の優先順位を持ちます。                                                                                                                                                                  |
| !         | 論理 NOT です。整数型に対して定義されています。++ と同一の優先順位を持ちます。                                                                                                                                                                             |
| ~         | ビットごとの NOT ( 補数 ) 演算子です。整数型に対して定義されています。++ と同一の優先順位を持ちます。                                                                                                                                                               |
| ., ->     | 構造体のメンバ、ポインタの指す構造体のメンバをそれぞれ指定する演算子です。便宜上、GDB は両者を同一のものとして扱い、格納されている型情報をもとに、ポインタによる間接参照の必要性を判断します。構造体 ( struct ) および共用体 ( union ) に対して定義されています。                                                                          |
| []        | 配列のインデックスです。 $a[i]$ は、 $*(a+i)$ として定義されています。-> と同一の優先順位を持ちます。                                                                                                                                                           |

- ( )           関数のパラメータ・リストです。->と同一の優先順位を持ちます。
- ::           C++のスコープ解決演算子です。構造体( struct )、共用体( union )、クラス( class )  
に対して定義されています。
- ::           2重コロンはまた、GDBのスコープ演算子も表わします( see Section 8.1 [Expressions], page 59 )。上記の::と同一の優先順位を持ちます。

#### 9.4.1.2 C/C++定数

GDBでは、以下のような方法によって、C/C++の定数を表わすことができます。

- 整数型定数は、数字の連続したものです。8進数定数は、先頭の‘0’（ゼロ）により指定されます。16進数定数は、先頭の‘0x’または‘0X’により指定されます。定数は、文字‘l’（エル）により終わることもあります。この場合、定数が long型の値として扱われるべきことを意味します。
- 浮動小数点型定数は、連続した数字、その後ろに小数点、さらにその後ろに数字という形式です。場合によっては、最後に指数部が付くこともあります。指数部は、‘e[+|-]nnn’という形式を取ります。ここで、nnnは連続した数字です。‘+’は、正の指数を示す記号で、必ずしも必要ではありません。
- 列挙型定数は、列挙識別子、またはそれに対応する整数値より構成されます。
- 文字型定数は、単一引用符( ’ )によって囲まれた単一の文字、あるいは、その文字に対応する序数（通常は、ASCII 値）です。引用符の中の単一文字は、文字またはエスケープ・シーケンスによって表わすことができます。エスケープ・シーケンスには2つの表記方法があります。第1の形式は‘\nnn’で、nnnはその文字の序数を表わす8進数です。第2の形式は‘\x’で、‘x’はあらかじめ定義された特別な文字です。例えば、‘\n’は改行を表わします。
- 文字列型定数は、連続した文字定数が2重引用符( " )で囲まれたものです。
- ポインタ型定数は、整数値です。定数へのポインタを、Cの‘&’演算子を使用して記述することができます。
- 配列定数は、括弧‘{’と‘}’で囲まれ、カンマで区切られたリストです。例えば、‘{1,2,3}’は3つの整数値を要素として持つ配列です。‘{{1,2},{3,4},{5,6}}’は、3 × 2の配列です。また、‘{&"hi", &"there", &"fred"}’は3つのポインタを要素として持つ配列です。

#### 9.4.1.3 C++式

GDBが持っている、式を処理する機能は、C++のほとんどの式を解釈することができます。

注意: GDBは、適切なコンパイラが使用されている場合のみ、C++のコードをデバッグすることができます。典型的な例を挙げると、C++のデバッグでは、シンボル・テーブルの中の追加的なデバッグ情報に依存するため、特別なサポートが必要になるということがあります。使用されるコンパイラが、a.out、MIPS ECOFF、RS/6000 XCOFF、ELFを、シンボル・テーブルに対する stabs 拡張付きで生成することができるのであれば、以下に列挙する機能を使用することができます（GNU CCの場合は、‘-gstabs’オプションを使用して明示的に stabs デバッグ拡張を要求することができます）。一方、オブジェクト・コードのフォーマットが、標準 COFF や ELF の DWARF である場合には、GDBの提供するほとんどの C++サポートは機能しません。

1. メンバ関数の呼び出しが許されます。以下のような式を使用することができます。

```
count = aml->GetOriginal(x, y)
```

2. メンバ関数が ( 選択されたスタック・フレームの中で ) アクティブな場合、入力された式は、そのメンバ関数と同一の名前空間を利用することができます。すなわち、GDB は、C++ と同様の規則にしたがって、クラス・インスタンスへのポインタ `this` への暗黙の参照を許します。
3. オーバーロードされた関数を呼び出すことができます。GDB は、正しい定義の関数呼び出しを決定します。ただし、制限が一点あります。実際に呼び出したい関数が要求する型の引数を使用しなければなりません。GDB は、コンストラクタやユーザ定義の型演算子を必要とするような変換を実行しません。
4. GDB は、C++ の参照変数として宣言された変数を理解します。C++ のソース・コードで参照変数を使用するのと同じ方法で、参照変数を式の中で使用することができます。参照変数は自動的に間接参照されます。  
GDB がフレームを表示する際に表示されるパラメータ一覧の中では、参照変数の値は ( 他の変数とは異なり ) 表示されません。これにより、表示が雑然となることを回避できます。というのは、参照変数は大きい構造体に対して使用されることが多いからです。参照変数のアドレスは、`'set print address off'` を指定しない限り、常に表示されます。
5. GDB は C++ の名前解決演算子 `::` をサポートしています。プログラム中と同様に、式の中でこれを使用することができます。あるスコープが別のスコープの中で定義されることがありえるため、必要であれば `::` を繰り返し使用することができます。例えば、`'scope1::scope2::name'` という具合です。GDB はまた、C および C++ のデバッグにおいて、ソース・ファイルを指定することで名前のスコープを解決することを許します ( see Section 8.2 [Program variables], page 60 )。

#### 9.4.1.4 C/C++ のデフォルト

GDB が自動的に型チェックや範囲チェックの設定を行うことを許すと、作業言語が C や C++ に変更されるときにはいつも、それらの設定はデフォルトで `off` になります。これは、作業言語を選択したのがユーザであっても GDB であっても同様です。

GDB が自動的に言語の設定を行うことを許すと、GDB は、名前が `'.c'`、`'.C'`、`'.cc'` などと終わるソース・ファイルを認識していて、これらのファイルからコンパイルされたコードの実行を開始するときに、作業言語を C または C++ に設定します。詳細については、See Section 9.1.3 [Having GDB infer the source language], page 76。

#### 9.4.1.5 C/C++ の型チェックと範囲チェック

デフォルトでは、GDB が C や C++ の式を解析するときには、型チェックは行われません。しかし、ユーザが型チェックを有効にすると、GDB は以下の条件が成立するときに、2 つの変数の型が一致しているとみなします。

- 2 つの変数が構造を持ち、同一の構造体タグ、共用体タグ、または列挙型タグを持つ。
- 2 つの変数が同一の型名を持つ、あるいは、`typedef` によって同一の型に宣言されている型を持つ。

範囲チェックは、`on` に設定されている場合、数学的演算において実行されます。配列のインデックスは、それ自体は配列ではないポインタのインデックスとして使用されることが多いため、チェックされません。

#### 9.4.1.6 GDB と C

`set print union` コマンドと `show print union` コマンドは共用体型 ( `union` ) に適用されます。`'on'` に設定されると、構造体 ( `struct` ) やクラス ( `class` ) の内部にある共用体 ( `union` ) はすべて表示されます。`'on'` でない場合、それは `'{...}'` と表示されます。

@オペレータは、ポインタとメモリ割り当て関数によって作られた動的配列のデバッグに役に立ちます。See Section 8.1 [Expressions], page 59。

#### 9.4.1.7 C++用の GDB 機能

GDB のコマンドの中には、C++を使用しているときに特に役に立つものがあり、また、C++専用 に特に設計されたものがあります。以下に、その要約を示します。

`breakpoint menus`

名前がオーバーロードされている関数の内部にブレイクポイントを設定したい場合、関心のある関数定義を指定するのに、GDB のブレイクポイント・メニューが役に立ちます。See Section 5.1.8 [Breakpoint menus], page 40。

`rbreak regex`

あるオーバーロードされたメンバ関数が、特別なクラスだけが持つメンバ関数というわけではない場合、そのメンバ関数にブレイクポイントを設定するのに、正規表現によるブレイクポイントの設定が役に立ちます。See Section 5.1.1 [Setting breakpoints], page 30。

`catch throw`

`catch catch`

C++の例外処理をデバッグするのに使用します。See Section 5.1.3 [Setting catch-points], page 34。

`ptype typename`

型 *typename* に関して、継承関係などの情報を表示します。See Chapter 10 [Examining the Symbol Table], page 91。

`set print demangle`

`show print demangle`

`set print asm-demangle`

`show print asm-demangle`

コードを C++のソースとして表示する場合と、逆アセンブル処理の結果を表示する場合に、C++のシンボルをソース形式で表示するか否かを制御します。See Section 8.7 [Print settings], page 65。

`set print object`

`show print object`

オブジェクトの型を表示する際に、派生した ( 実際の ) 型と宣言された型のどちらを表示するかを選択します。See Section 8.7 [Print settings], page 65。

`set print vtbl`

`show print vtbl`

仮想関数テーブルの表示形式を制御します。See Section 8.7 [Print settings], page 65。

オーバーロードされたシンボル名

オーバーロードされたシンボルを宣言するのに C++において使用されるのと同じの表記法を使用して、オーバーロードされたシンボル定義のうち、特定のものを指定することができます。単に *symbol* と入力するのではなく、*symbol(types)* と入力してください。GDB コマンドラインの単語補完機能を使用して、利用可能な選択肢を一覧表示させたり、型のリストを完結させたりすることができます。この機能の使用の詳細については、See Section 3.2 [Command completion], page 13。

## 9.4.2 Modula-2

Modula-2 をサポートするために開発された GDB の拡張機能は、( 現在開発中の ) GNU Modula-2 コンパイラによって生成されたコードだけをサポートします。他の Modula-2 コンパイラは現在サポートされていません。他の Modula-2 コンパイラが生成した実行形式モジュールをデバッグしようとすると、おそらく、GDB が実行モジュールのシンボル・テーブルを読み込もうとしたところでエラーになるでしょう。

### 9.4.2.1 Modula-2 演算子

演算子は、特定の型の値に対して定義されなければなりません。例えば、+ は数値に対して定義され、構造体に対しては定義されません。演算子は、型のグループに対して定義されることがよくあります。Modula-2 においては、以下の定義が有効です。

- 整数型は、INTEGER、CARDINAL、およびそのサブ範囲 ( subrange ) から成ります。
- 文字型は、CHAR とそのサブ範囲から成ります。
- 浮動小数点型は、REAL から成ります。
- ポインタ型は、POINTER TO *type* のように宣言された任意の型から成ります。
- スカラ型は、上記のすべての型から成ります。
- 集合型は、SET、BITSET から成ります。
- ブール型は、BOOLEAN から成ります。

以下の演算子がサポートされています。ここでは、優先順位の低いものから順に並べています。

|          |                                                                                                           |
|----------|-----------------------------------------------------------------------------------------------------------|
| ,        | 関数の引数の区切り記号、または、配列のインデックスの区切り記号です。                                                                        |
| :=       | 代入です。var := value の値は value です。                                                                           |
| <, >     | 未満、超過です。整数型、浮動小数点型、列挙型に対して定義されています。                                                                       |
| <=, >=   | 整数型、浮動小数点型、列挙型に対しては、以下、以上を表わします。集合型に対しては、集合の包含関係を表わします。< と同一の優先順位を持ちます。                                   |
| =, <>, # | スカラ型に対して定義されている等価および 2 種類の不等価です。< と同一の優先順位を持ちます。GDB スクリプトの中では、# がスクリプトのコメント記号でもあるため、不等価としては <> だけが使用可能です。 |
| IN       | 集合のメンバを表わします。集合型、およびそのメンバの型に対して定義されています。< と同一の優先順位を持ちます。                                                  |
| OR       | ブール型の OR ( disjunction ) です。ブール型に対して定義されています。                                                             |
| AND, &   | ブール型の AND ( conjunction ) です。ブール型に対して定義されています。                                                            |
| @        | GDB の「人工配列」演算子です ( see Section 8.1 [Expressions], page 59 )。                                              |
| +, -     | 整数型、浮動小数点型に対しては、加算、減算を表わします。集合型に対しては、和集合 ( union )、差集合 ( difference ) を表わします。                             |
| *        | 整数型、浮動小数点型に対しては、乗算を表わします。集合型に対しては、積集合 ( intersection ) を表わします。                                            |
| /        | 浮動小数点型に対しては、除算を表わします。集合型に対しては、対称的差集合 ( symmetric difference ) を表わします。* と同一の優先順位を持ちます。                     |

|         |                                                                |
|---------|----------------------------------------------------------------|
| DIV、MOD | 整数型の除算における商と剰余を表わします。整数型に対して定義されています。*と同一の優先順位を持ちます。           |
| -       | マイナス（負）です。INTEGER、REAL型のデータに対して定義されています。                       |
| ^       | ポインタの間接参照です。ポインタ型に対して定義されています。                                 |
| NOT     | ブール型の NOT です。ブール型に対して定義されています。^と同一の優先順位を持ちます。                  |
| .       | RECORDフィールドの区切り記号です。RECORDデータに対して定義されます。^と同一の優先順位を持ちます。        |
| []      | 配列のインデックスを指定します。ARRAY型のデータに対して定義されています。^と同一の優先順位を持ちます。         |
| ()      | プロシージャの引数リストを指定します。PROCEDUREオブジェクトに対して定義されています。^と同一の優先順位を持ちます。 |
| ::、.    | GDB および Modula-2 のスコープ指定演算子です。                                 |

注意: 集合、および集合に対する操作は、まだサポートされていません。このため、GDB は IN 演算子、あるいは、集合に対して +、-、\*、/、=、<>、#、<=、>= のいずれかの演算子が使用された場合、これをエラーとして扱います。

#### 9.4.2.2 組み込み関数と組み込みプロシージャ

Modula-2 では、いくつかの組み込みプロシージャ、組み込み関数を使用できます。これらの説明にあたり、以下のメタ変数を使用します。

|          |                                                                                                                                      |
|----------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>a</i> | ARRAY型の変数を表わします。                                                                                                                     |
| <i>c</i> | CHAR型の定数、または変数を表わします。                                                                                                                |
| <i>i</i> | 整数型の変数、または定数を表わします。                                                                                                                  |
| <i>m</i> | 集合に属する識別子を表わします。通常、同一関数の中でメタ変数 <i>s</i> とともに使用されます。 <i>s</i> の型は、SET OF <i>mtype</i> でなければなりません (ここでの <i>mtype</i> は <i>m</i> の型です)。 |
| <i>n</i> | 整数型または浮動小数点型の、変数または定数を表わします。                                                                                                         |
| <i>r</i> | 浮動小数点型の変数または定数を表わします。                                                                                                                |
| <i>t</i> | 型を表わします。                                                                                                                             |
| <i>v</i> | 変数を表わします。                                                                                                                            |
| <i>x</i> | 多くの型の中の 1 つの型の、変数または定数を表わします。詳細については、関数の説明の部分を参照してください。                                                                              |

また、すべての Modula-2 の組み込みプロシージャは、以下に説明する値を返します。

|                 |                                                                          |
|-----------------|--------------------------------------------------------------------------|
| ABS( <i>n</i> ) | 値 <i>n</i> の絶対値を返します。                                                    |
| CAP( <i>c</i> ) | <i>c</i> が小文字であれば、それを大文字にして返します。 <i>c</i> が小文字でなければ、 <i>c</i> をそのまま返します。 |
| CHR( <i>i</i> ) | 序数が <i>i</i> である文字を返します。                                                 |
| DEC( <i>v</i> ) | 変数 <i>v</i> の値から 1 を引きます。新しい値を返します。                                      |



- DEC(*v*,*i*) 変数 *v* の値から *i* で示される値を引きます。新しい値を返します。
- EXCL(*m*,*s*) 集合 *s* から要素 *m* を取り除きます。新しい集合を返します。
- FLOAT(*i*) 整数値 *i* に等しい浮動小数点値を返します。
- HIGH(*a*) 配列 *a* の最後の要素のインデックスを返します。
- INC(*v*) 変数 *v* の値に 1 を加えます。新しい値を返します。
- INC(*v*,*i*) 変数 *v* の値に *i* で示される値を加えます。新しい値を返します。
- INCL(*m*,*s*) 集合 *s* に要素 *m* が存在しない場合、要素 *m* を追加します。新しい集合を返します。
- MAX(*t*) 型 *t* の最大値を返します。
- MIN(*t*) 型 *t* の最小値を返します。
- ODD(*i*) *i* が奇数であればブール型の TRUE を返します。
- ORD(*x*) 引数の序数値を返します。例えば、文字の序数値は、( ASCII 文字セットをサポートするマシン上では ) その ASCII 値です。ここで *x* は、整数型、文字型、列挙型のような順序を持つ型でなければなりません。
- SIZE(*x*) 引数のサイズを返します。*x* は変数または型のいずれかです。
- TRUNC(*r*) *r* の整数部を返します。
- VAL(*t*,*i*) 型 *t* のメンバのうち、その序数値が *i* であるものを返します。

注意: 集合、および集合に対する操作はまだサポートされていません。したがって、INCL プロシージャ、EXCL プロシージャを使用すると、GDB はエラーとして扱います。

### 9.4.2.3 定数

GDB では、Modula-2 の定数を以下のような方法で表現することができます。

- 整数型の定数は、単に数字が連続したものです。式の中で使用された場合、定数は、式の他の部分と互換性のある型を持つものとみなされます。16 進数の整数は末尾に 'H' を付加することで、また、8 進数の整数は末尾に 'B' を付加することで指定されます。
- 浮動小数点型の定数は、連続した数字、その後ろに小数点、さらにその後ろに連続した数字が続くものです。場合によっては、この後ろに指数部を指定することができます。指数部の形式は 'E[+|-]nnn' で、'[+|-]nnn' の部分で希望する指数を指定します。浮動小数点型定数のすべての数字は、有効な 10 進数値でなければなりません。
- 文字型定数は、単一引用符 ( ' ) または 2 重引用符 ( " ) で囲まれた単一文字より成ります。文字型定数は、その文字の序数値 ( 通常は ASCII 値 ) の後ろに 'C' を付加することで表現することもできます。
- 文字列型定数は、単一引用符 ( ' ) または 2 重引用符 ( " ) で囲まれた連続する文字から成ります。C 言語のスタイルでのエスケープ・シーケンスも使用できます。エスケープ・シーケンスに関する簡単な説明については、See Section 9.4.1.2 [C and C++ constants], page 82。
- 列挙型定数は、列挙識別子から成ります。
- ブール型定数は、識別子 TRUE および FALSE から成ります。
- ポインタ型定数は、整数値だけから成ります。
- 集合型定数は、まだサポートされていません。

#### 9.4.2.4 Modula-2 デフォルト

型チェックと範囲チェックが GDB により自動的に設定される場合、作業言語が Modula-2 に変わるたびに、それらはデフォルトで on に設定されます。これは、作業言語を選択したのがユーザであろうと GDB であろうと同様です。

GDB に自動的に言語を設定させると、ファイル名の末尾が `.mod` であるファイルからコンパイルされたコードに入るたびに、作業言語は Modula-2 に設定されます。詳細については、See Section 9.1.3 [Having GDB set the language automatically], page 76。

#### 9.4.2.5 標準 Modula-2 との差異

Modula-2 プログラムのデバッグを容易にするために 2、3 の修正が施されています。これは主に、型に対する厳密性を緩めることによって実現されています。

- 標準 Modula-2 とは異なり、ポインタ型定数は整数値から作成することができます。これにより、デバッグ中にポインタ変数の値を変更できるようになります (標準 Modula-2 では、ポインタ変数に格納されている実際のアドレスを知ることができません。ポインタ変数内のアドレスは、他のポインタ変数、または、ポインタを返す式を直接的に代入することによってのみ修正することができます)。
- 表示不可の文字を表わすのに、C 言語のエスケープ・シーケンスを文字列や文字において使用することができます。GDB はこれらのエスケープ・シーケンスを埋め込んだまま文字列を表示します。表示不可の単一文字は、`'CHR(nnn)'` という形式で表示されます。
- 代入演算子 (`:=`) は、右側の引数の値を返します。
- すべての組み込みプロシージャは、引数を修正し、さらにそれを返します。

#### 9.4.2.6 Modula-2 の型チェックと範囲チェック

注意: GDB は現在のところ、型チェック、範囲チェックをまだ実装していません。

GDB は、以下のいずれかの条件が成立するとき、2 つの Modula-2 変数の型が等しいとみなします。

- 2 つの型が、`TYPE t1 = t2` 文によって等しいと宣言されている型である。
- 2 つの型が同一行において宣言されている (注: これは GNU Modula-2 コンパイラにおいては正しいのですが、他のコンパイラにおいては正しくない可能性があります)。

型チェックが有効である限り、等しくない型の変数を組み合わせようとする試みはすべてエラーとなります。

範囲チェックは、数学的操作、代入、配列のインデックス境界、およびすべての組み込み関数、組み込みプロシージャにおいて実行されます。

#### 9.4.2.7 スコープ演算子 `::` と `.`

Modula-2 のスコープ演算子 (`.`) と GDB のスコープ演算子 (`::`) との間には 2、3 の微妙な相違点があります。この 2 つは似た構文を持っています。

```
module . id
scope :: id
```

ここで、*scope* はモジュール名またはプロシージャ名、*module* はモジュール名、*id* はユーザ・プログラムの中で宣言された任意の（異なるモジュール以外の）識別子です。

:: 演算子を使用すると、GDB は *scope* によって指定されたスコープにおいて識別子 *id* を探します。指定されたスコープにおいてそれを見つけないと、GDB は *scope* によって指定されたスコープを包含するすべてのスコープを探します。

. 演算子を使用すると、GDB はカレントなスコープにおいて、*module* によって指定された定義モジュールから取り込まれた、*id* によって指定される識別子を探します。この演算子では、識別子 *id* が定義モジュール *module* から取り込まれていない場合や *module* において *id* が識別子でない場合は、エラーになります。

#### 9.4.2.8 GDB と Modula-2

GDB コマンドの中には、Modula-2 プログラムのデバッグにはほとんど役に立たないものがあります。set print、show print の 5 つのサブ・コマンド ‘vtbl’、‘demangle’、‘asm-demangle’、‘object’、‘union’ は C/C++ にのみ適用されます。最初の 4 つは C++ に適用され、最後の 1 つは C の共用体 (union) に適用されます。これらは、Modula-2 において直接類似するものが存在しません。

@ 演算子 (see Section 8.1 [Expressions], page 59) は、どの言語においても使用することができますが、Modula-2 においてはあまり役に立ちません。この演算子は、動的配列のデバッグを支援することを目的とするものですが、C/C++ では作成できる動的配列は、Modula-2 では作成できません。しかし、整数値定数によってアドレスを指定することができるので、‘{type} adrexp’ は役に立ちます (see Section 8.1 [Expressions], page 59)。

GDB スクリプトの中では、Modula-2 の不等価演算子 # はコメントの開始記号として解釈されます。代わりに <> を使用してください。



## 10 シンボル・テーブルの検査

ここで説明するコマンドによって、ユーザ・プログラムの中で定義されているシンボル情報（変数名、関数名、型名）に関する問い合わせを行うことができます。この情報はユーザ・プログラムのテキストに固有のもので、プログラムの実行時に変わるものではありません。GDBはこの情報を、ユーザ・プログラムのシンボル・テーブルの中、または、GDB 起動時に指定されたファイル（see Section 2.1.1 [Choosing files], page 10）の中で見つけるか、ファイル管理コマンド（see Section 12.1 [Commands to specify files], page 99）の実行によって見つけます。

ときには、参照する必要のあるシンボルの中に、GDB が通常は単語の区切り文字として扱う文字が含まれていることがあるかもしれません。特に多いのが、他のソース・ファイルの中の静的変数を参照する場合です（see Section 8.2 [Program variables], page 60）。ファイル名は、オブジェクト・ファイルの中にデバッグ・シンボルとして記録されていますが、GDB は通常、典型的なファイル名、例えば `'foo.c'` を解析して、3 つの単語 `'foo'`、`'.'`（ピリオド）、`'c'` であるとみなします。GDB が `'foo.c'` を単一のシンボルであると認識できるようにするには、それを単一引用符で囲みます。例えば、

```
p 'foo.c'::x
```

は、`x` の値をファイル `'foo.c'` のスコープの中で検索します。

**info address *symbol***

*symbol* で指定されるシンボルのデータがどこに格納されているかを示します。レジスタ変数の場合は、それがどのレジスタに入っているかを示します。レジスタ変数ではないローカル変数の場合は、その変数が常に格納されている位置の、スタック・フレーム内におけるオフセット値を表示します。

`'print &symbol'` との相違に注意してください。`'print &symbol'` はレジスタ変数に対しては機能しませんし、スタック内のローカル変数に対して実行すると、その変数のカレントなインスタンスの存在するアドレスそのものが表示されます。

**whatis *exp***

式 *exp* のデータ型を表示します。*exp* は実際には評価されず、*exp* 内の副作用を持つ操作（例えば、代入や関数呼び出し）は実行されません。See Section 8.1 [Expressions], page 59。

**whatis**      値ヒストリの最後の値である \$ のデータ型を表示します。

**ptype *typename***

データ型 *typename* の説明を表示します。*typename* は型の名前です。C で記述されたコードの場合は、`'class class-name'`、`'struct struct-tag'`、`'union union-tag'`、`'enum enum-tag'` という形式を取ることができます。

**ptype *exp***

**ptype**      式 *exp* の型に関する説明を表示します。単に型の名前を表示するだけではなく、詳細な説明も表示するという点で、**ptype** は **whatis** と異なります。

例えば、変数宣言

```
struct complex {double real; double imag;} v;
```

に対して、**whatis**、**ptype** はそれぞれ以下のような出力をもたらします。

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

`whatis`と同様、引数なしで `ptype` を使用すると、値ヒストリの最後の値である \$ の型を参照することになります。

`info types regexp`

`info types`

その名前が `regexp` で指定される正規表現にマッチするすべての型 (あるいは、引数を指定しなければ、ユーザ・プログラム中のすべての型) の簡単な説明を表示します。個々の型の完全な名前は、それ自体が 1 つの完全な行を構成するものとみなして、マッチされます。したがって、`'i type value'` は、ユーザ・プログラムの中で、その名前が文字列 `value` を含むすべての型に関する情報を表示し、`'i type ^value$'` は、名前が `value` そのものである型に関する情報だけを表示します。

このコマンドは `ptype` とは 2 つの点で異なります。まず第 1 に `whatis` と同様、詳細な情報を表示しません。第 2 に、型が定義されているすべてのソース・ファイルを一覧表示します。

`info source`

カレントなソース・ファイル、すなわち、カレントな実行箇所を含む関数のソース・ファイルの、ファイル名とそれが記述された言語の名前を表示します。

`info sources`

ユーザ・プログラムのソース・ファイルのうち、デバッグ情報の存在するものすべての名前を、2 つの一覧にして表示します。2 つの一覧とは、シンボルが既に読み込まれたファイルの一覧と、後に必要なときにシンボルが読み込まれるファイルの一覧です。

`info functions`

すべての定義済み関数の名前とデータ型を表示します。

`info functions regexp`

その名前が `regexp` で指定される正規表現にマッチする部分を持つすべての定義済み関数の名前とデータ型を表示します。したがって、`'info fun step'` は、その名前が文字列 `step` を含むすべての関数を見つけ、`'info fun ^step'` は、名前が文字列 `step` で始まるすべての関数を見つけてくれます。

`info variables`

関数の外部で宣言されているすべての変数 (つまり、ローカル変数を除く変数) の名前とデータ型を表示します。

`info variables regexp`

その名前が正規表現 `regexp` にマッチする部分を持つすべての (ローカル変数を除く) 変数の名前とデータ型を表示します。

いくつかのシステムにおいては、ユーザ・プログラムの停止・再起動を伴うことなく、そのユーザ・プログラムを構成する個々のオブジェクト・ファイルを更新することができます。例えば、VxWorks では、欠陥のあるオブジェクト・ファイルを再コンパイルして、実行を継続することができます。このようなマシン上でプログラムを実行している

のであれば、自動的に再リンクされたモジュールのシンボルを GDB に再ロードさせることができます。

```
set symbol-reloading on
```

ある特定の名前を持つオブジェクト・ファイルが再検出されたときに、対応するソース・ファイルのシンボル定義を入れ替えます。

```
set symbol-reloading off
```

同じ名前を持つオブジェクト・ファイルを再検出したときに、シンボル定義を入れ替えません。これがデフォルトの状態です。モジュールの自動再リンクを許しているシステム上でプログラムを実行しているのでない場合は、symbol-reloadingの設定はoffのままにするべきです。さもないと、(異なるディレクトリやライブラリの中にある)同じ名前を持ついくつかのモジュールを含むような大きなプログラムをリンクする際に、GDB はシンボルを破棄してしまうかもしれません。

```
show symbol-reloading
```

symbol-reloadingのカレントな設定 ( onまたは off ) を表示します。

```
maint print symbols filename
```

```
maint print psymbols filename
```

```
maint print msymbols filename
```

デバッグ・シンボル・データのダンプをファイル *filename* の中に書き込みます。これらのコマンドは、GDB のシンボル読み込みコードをデバッグするのに使われています。デバッグ・データを持つシンボルだけがダンプに含まれます。‘maint print symbols’を使用すると、GDB は、完全な詳細情報を入手済みのすべてのシンボルの情報をダンプに含めます。つまり、ファイル *filename* には、GDB がそのシンボルを読み込み済みのファイルに対応するシンボルが反映されます。info sourcesコマンドを使用することで、これらのファイルがどれであるかを知ることができます。代わりに‘maint print psymbols’を使用すると、GDB が部分的にしか知らないシンボルに関する情報もダンプの中に含まれます。これは、GDB がざっと読みはしたものの、まだ完全には読み込んでいないファイルに定義されているシンボルに関する情報です。最後に‘maint print msymbols’では、GDB が何らかのシンボル情報を読み込んだオブジェクト・ファイルから、最小限必要とされるシンボル情報がダンプされます。GDB がどのようにしてシンボルを読み込むかについては、Section 12.1 [Commands to specify files], page 99 ( の symbol-file の説明の部分 ) を参照してください。





## 11 実行の変更

ユーザ・プログラムの中に誤りのある箇所を見つけると、その明らかな誤りを訂正することで、その後の実行が正しく行われるかどうかを知りたくなるでしょう。GDB にはプログラムの実行に変化を与える機能があり、これを使って実験することで、その答を知ることができます。

例えば、変数やメモリ上のある箇所に新しい値を格納すること、ユーザ・プログラムにシグナルを送ること、ユーザ・プログラムを異なるアドレスで再起動すること、関数が完全に終了する前に呼び出し元に戻るなどが可能です。

### 11.1 変数への代入

ある変数の値を変更するには、代入式を評価します。See Section 8.1 [Expressions], page 59. 例えば、

```
print x=4
```

は、変数 `x` に値 4 を格納してから、その代入式の値（すなわち 4）を表示します。サポートされている言語の演算子の詳細情報については、See Chapter 9 [Using GDB with Different Languages], page 75.

代入の結果を表示させることに関心がなければ、`print` コマンドの代わりに `set` コマンドを使用してください。実際のところ `set` コマンドは、式の値が表示もされず、値ヒストリ（see Section 8.8 [Value history], page 70）にも入らないということを除けば、`print` コマンドと同等です。式は、その結果の入手だけを目的として評価されます。

`set` コマンドの引数となる文字列の先頭の部分が、`set` コマンドのサブ・コマンドの名前と一致してしまうような場合には、ただの `set` コマンドではなく `set variable` コマンドを使用してください。このコマンドは、サブ・コマンドを持たないという点を除けば、`set` コマンドと同等です。例えば、ユーザ・プログラムに `width` という変数がある場合、`'set width=13'` によってこの変数に値を設定しようとするとエラーになります。これは、GDB が `set width` というコマンドを持っているためです。

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

ここで不正な表現となっているのは、もちろん `=47` の部分です。プログラム内の変数 `width` に値を設定するには、以下のようにしてください。

```
(gdb) set var width=47
```

GDB は、代入時の暗黙の型変換を C 言語よりも多くサポートしています。整数値を自由にポインタ型変数に格納できますし、その逆もできます。また、任意の構造体を、同じサイズの別の構造体、または、より小さいサイズの別の構造体に変換することができます。

メモリ上の任意の箇所に値を格納するには、指定されたアドレスにおいて指定された型の値を生成するために、`{...}` を使用します（see Section 8.1 [Expressions], page 59）。例えば `{int}0x83040` は、メモリ・アドレス `0x83040` を整数値として参照します（メモリ上における、ある特定のサイズと表現を示唆しています）。また、

```
set {int}0x83040 = 4
```

は、そのメモリ・アドレスに値 4 を格納します。

## 11.2 異なるアドレスにおける処理継続

通常、ユーザ・プログラムを継続実行するには、`continue`コマンドを使用して、停止した箇所から継続実行させます。以下のコマンドを使用することで、ユーザが選択したアドレスにおいて実行を継続させることができます。

`jump linespec`

*linespec* で指定される行において、実行を再開します。その行にブレイクポイントが設定されている場合には、実行は再びすぐに停止します。*linespec* の形式については、See Section 7.1 [Printing source lines], page 53。一般的な慣例として、`jump`コマンドは、`tbreak`コマンドと組み合わせて使用されます。See Section 5.1.1 [Setting breakpoints], page 30。

`jump`コマンドは、カレントなスタック・フレーム、スタック・ポインタ、メモリ内の任意の箇所の内容、プログラム・カウンタを除くレジスタの内容を変更しません。*linespec* で指定される行が、現在実行されている関数とは異なる関数の中にある場合、それら2つの関数が異なるパターンの引数やローカル変数を期待していると、奇妙な結果が発生するかもしれません。このため、指定された行が、現在実行されている関数の中になければ、`jump`コマンドは実行の確認を求めてきます。しかし、ユーザがプログラムのマシン言語によるコードを熟知していたとしても、奇妙な結果の発生することが予想されます。

`jump *address`

*address* で指定されるアドレスにある命令から、実行を再開します。

レジスタ`$pc`に新しい値を設定することで、`jump`コマンドとほとんど同等の効果を実現することができます。両者の違いは、レジスタ`$pc`に値を設定しただけでは、ユーザ・プログラムの実行は再開されないという点にあります。ユーザが実行を継続するときに、プログラムが実行を再開するアドレスが変更されるだけです。例えば、

```
set $pc = 0x485
```

を実行すると、次に `continue`コマンドやステップ実行を行うコマンドが実行されるとき、ユーザ・プログラムが停止したアドレスにある命令ではなく、アドレス `0x485`にある命令から実行されることになります。See Section 5.2 [Continuing and stepping], page 41。

`jump`コマンドが最も一般的に使用されるのは、既に実行されたプログラム部分を、さらに多くのブレイクポイントを設定した状態で再実行する場合でしょう。これにより、実行される処理の内容をさらに詳しく調べることができます。

## 11.3 ユーザ・プログラムへのシグナルの通知

`signal signal`

実行を停止した箇所からユーザ・プログラムを再開させますが、すぐに *signal* で指定されるシグナルを通知します。*signal* には、シグナルの名前または番号を指定できます。例えば、多くのシステムにおいて、`signal 2`と `signal SIGINT`はどちらも、割り込みシグナルを通知する方法です。

一方、*signal* が `0` であれば、シグナルを通知することなく実行を継続します。ユーザ・プログラムがシグナルのために停止し、通常であれば、`continue`コマンドによって実行を再開するとそのシグナルを検知してしまうような場合に便利です。‘`signal 0`’を実行すると、プログラムはシグナルを受信することなく実行を再開します。

`signal`を実行した後、`(RET)`キーを押しても、繰り返し実行は行われません。

signalコマンドを実行することは、シェルから killユーティリティを実行するのと同じではありません。killによってシグナルを送ると、GDBはシグナル処理テーブルによって何をすべきかを決定します (see Section 5.3 [Signals], page 43)。一方、signalコマンドは、ユーザ・プログラムに直接シグナルを渡します。

## 11.4 関数からの復帰

```
return  
return expression
```

returnコマンドによって、呼び出されている関数の実行をキャンセルすることができます。式 *expression* を引数に指定すると、その値が関数の戻り値として使用されます。

returnを実行すると、GDBは選択されているスタック・フレーム (および、その下位にあるすべてのフレーム) を破棄します。破棄されたフレームは、実行を完結する前に復帰したのだと考えればよいでしょう。戻り値を指定したいのであれば、その値を returnへの引数として渡してください。

このコマンドは、選択されているスタック・フレーム (see Section 6.3 [Selecting a frame], page 49) および、その下位にあるすべてのフレームをポップして、もともと選択されていたフレームを呼び出したフレームを、最下位のフレームにします。つまり、そのフレームが選択されることになります。指定された値は、関数から戻り値を返すのに使用されるレジスタに格納されます。

returnコマンドは実行を再開しません。関数から復帰した直後の状態で、プログラムを停止したままにします。これに対して、finishコマンド (see Section 5.2 [Continuing and stepping], page 41) は、選択されているスタック・フレームが自然に復帰するまで、実行を再開、継続します。

## 11.5 プログラム関数の呼び出し

call *expr* void型の戻り値を表示することなく、式 *expr* を評価します。

ユーザ・プログラムの中からある関数を呼び出したいが、void 型の戻り値を出力させたくない場合、この printコマンドの変種を使用することができます。void型でない戻り値は表示され、値ヒストリに保存されます。

A29K では、ユーザに制御される変数 call\_scratch\_addressによって、GDBがデバッグ対象の関数を呼び出すときに使用するスクラッチ領域が指定されます。通常はスクラッチ領域をスタック上に置きますが、この方法は命令空間とデータ空間を別々に持つシステム上では機能しないため、これが必要になります。

## 11.6 プログラムへのパッチ適用

デフォルトでは、GDBはユーザ・プログラムの実行コードを持つファイル (あるいは、コア・ファイル) を書き込み不可の状態オープンします。これにより、マシン・コードを誤って変更してしまうことを防ぐことができます。しかし、ユーザ・プログラムのバイナリに意図的にパッチを適用することもできなくなってしまう。

バイナリにパッチを適用したいのであれば、set writeコマンドによって明示的にそのことを指定することができます。例えば、内部的なデバッグ・フラグを立てたり、緊急の修正を行いたいということがあられるでしょう。

```
set write on
```

```
set write off
```

‘set write on’を指定すると、GDB は実行ファイルやコア・ファイルを、読み込み、書き込みともに可能な状態でオープンします。‘set write off’（デフォルト）を指定すると、GDB はこれらのファイルを読み込みしかできない状態でオープンします。

既にファイルをロード済みの場合、set writeの設定を変更後、その変更を反映させるためには、( exec-fileコマンド、core-fileコマンドを使用して )、そのファイルを再ロードしなければなりません。

```
show write
```

実行ファイル、コア・ファイルが、読み込みだけではなく書き込みもできる状態でオープンされる設定になっているか否かを表示します。

## 12 GDB ファイル

GDB はデバッグ対象となるプログラムのファイル名を知っている必要があります。これは、プログラムのシンボル・テーブルを読み込むためでもあり、また、プログラムを起動するためでもあります。過去に生成されたコア・ダンプをデバッグするには、GDB にコア・ダンプ・ファイルの名前を教えてやらなければなりません。

### 12.1 ファイルを指定するコマンド

実行ファイルやコア・ダンプ・ファイルの名前を指定したい場合があります。これは通常、GDB の起動コマンドへの引数を利用して、起動時に行います ( see Chapter 2 [Getting In and Out of GDB], page 9 )

ときには、GDB のセッション中に、異なるファイルに切り替える必要がでてくることがあります。あるいは、GDB を起動するときに、使いたいファイルの名前を指定するのを忘れたということもあるかもしれません。このような場合に、新しいファイルを指定する GDB コマンドが便利です。

`file filename`

`filename` で指定されるプログラムをデバッグ対象にします。そのプログラムは、シンボル情報とメモリ内容を獲得するために読み込まれます。また、ユーザが `run` コマンドを使用したときに実行されます。ユーザがディレクトリを指定せず、そのファイルが GDB の作業ディレクトリに見つからない場合、シェルが実行すべきファイルを探すときと同様、GDB は、ファイルを探すべきディレクトリのリストとして環境変数 `PATH` の値を使用します。 `path` コマンドによって、GDB、ユーザ・プログラムの両方について、この変数の値を変更することができます。

ファイルをメモリにマップすることのできるシステムでは、補助的なファイル '`filename.syms`' に、ファイル `filename` のシンボル・テーブル情報が格納されることがあります。このような場合、GDB は、 '`filename.syms`' というファイルからシンボル・テーブルをメモリ上にマップすることで、起動に要する時間を短くします。詳細については、( 以下に説明する `file` コマンド、`symbol-file` コマンド、`add-symbol-file` コマンドを実行する際にコマンドライン上で使用可能な ) ファイル・オプションの '`-mapped`'、 '`-readnow`' の説明を参照してください。

`file`      `file` コマンドを引数なしで実行すると、GDB は実行ファイル、シンボル・テーブルに関して保持している情報をすべて破棄します。

`exec-file [ filename ]`

実行するプログラムが `filename` で指定されるファイル内に存在する ( ただし、シンボル・テーブルはそこには存在しない ) ということを指定します。GDB は、必要であれば、ユーザ・プログラムの存在場所を見つけるために、環境変数 `PATH` を使用します。 `filename` を指定しないと、実行ファイルに関して保持している情報を破棄するよう指示したことになります。

`symbol-file [ filename ]`

`filename` で指定されるファイルからシンボル・テーブル情報を読み込みます。必要な場合には `PATH` が検索されます。同一のファイルから、シンボル・テーブルと実行プログラムの両方を獲得する場合には、`file` コマンドを使用してください。

`symbol-file` を引数なしで実行すると、GDB がユーザ・プログラムのシンボル・テーブルに関して持っている情報は消去されます。

symbol-file コマンドが実行されると、それまで GDB が保持していたコンビニエンス変数、値ヒストリ、すべてのブレイクポイント、自動表示式は破棄されます。その理由は、これらの情報の中に、GDB が破棄した古いシンボル・テーブルのデータの一部である、シンボルやデータ型を記録する内部データへのポインタが含まれているかもしれないからです。

symbol-file を一度実行した後に **(RET)** キーを押しても、symbol-file の実行は繰り返されません。

GDB は、特定の環境用に構成されると、その環境において生成される標準フォーマットのデバッグ情報を理解するようになります。GNU コンパイラを使うこともできますし、ローカルな環境の規約に従う他のコンパイラを使用することもできます。通常は、GNU コンパイラを使用しているときに最高の結果を引き出すことができます。例えば gcc を使用すると、最適化されたコードに対してデバッグ情報を生成することができます。

COFF を使用する古い SVR3 システムを除外すれば、ほとんどの種類のオブジェクト・ファイルでは、symbol-file コマンドを実行しても、通常は、ただちにシンボル・テーブルの全体が読み込まれるわけではありません。実際に存在するソース・ファイルとシンボルを知るために、シンボル・テーブルを素早く調べるだけです。詳細な情報は、後にそれが必要になったときに、一度に 1 ソース・ファイルずつ読み込まれます。

2 段階に分けて読み込むという手法は、GDB の起動時間の短縮を目的としています。ほとんどの場合、このような手法が採用されているということに気付くことはありません。せいぜい、特定のソース・ファイルに関するシンボル・テーブルの詳細が読み込まれている間、たまに停止するくらいです（もしそうしたいのであれば、set verbose コマンドを使うことによって、このようにして停止しているときにはメッセージを表示させることもできます。See Section 14.6 [Optional warnings and messages], page 134）。

COFF については、まだこの 2 段階方式を実装していません。シンボル・テーブルが COFF フォーマットで格納されている場合、symbol-file コマンドはシンボル・テーブル・データの全体をただちに読み込みます。COFF の stabs 拡張フォーマット (stabs-in-COFF) では、デバッグ情報が実際には stabs フォーマットの内部に存在するため、2 段階方式が実装されていることに注意してください。

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

```
file filename [ -readnow ] [ -mapped ]
```

GDB が確実にシンボル・テーブル全体を保持しているようにしたいのであれば、シンボル・テーブル情報を読み込む任意のコマンド実行時に ‘-readnow’ オプションを使用することで、2 段階によるシンボル・テーブル読み込み方式を使わないようにさせることができます。

mmap システム・コールによるファイルのメモリへのマッピングがシステム上において有効な場合、もう 1 つのオプション ‘-mapped’ を使って、GDB に対して、再利用可能なファイルの中にユーザ・プログラムのシンボルを書き込ませることができます。後の GDB デバッグ・セッションは、（プログラムに変更がない場合）実行プログラムからシンボル・テーブルを読み込むのに時間を費やすことなく、この補助シンボル・ファイルからシンボル情報をマップします。‘-mapped’ オプションを使用することは、コマンドライン・オプション ‘-mapped’ を指定して GDB を起動するのと同じ効果を持ちます。

補助シンボル・ファイルがユーザ・プログラムのシンボル情報をすべて確実に持つように、両方のオプションを同時に指定することもできます。

myprog という名前のプログラムの補助シンボル・ファイルは、‘myprog.syms’ という名前になります。このファイルが存在すると、（それが、対応する実行ファイルよりも新

しい限り)ユーザが *myprog* をデバッグしようとする、GDB は常にそのファイルを使おうとします。特別なオプションやコマンドは必要ありません。

‘.syms’ファイルは、GDB を実行したホスト・マシンに固有のもので、それは、GDB 内部におけるシンボル・テーブルの正確なイメージを保持しています。複数のホスト・プラットフォーム間で共用することはできません。

**core-file** [ *filename* ]

「メモリ上のイメージ」として使用されるコア・ダンプ・ファイルの存在場所を指定します。伝統的に、コア・ファイルは、それを生成したプロセスのアドレス空間の一部だけを保持しています。GDB は、実行ファイルそのものにアクセスすることによって、保持されていない部分を獲得することができます。

**core-file** を引数なしで実行すると、コア・ファイルを一切使用しないことを指定したことになります。

ユーザ・プログラムが実際に GDB の管理下で実行中の場合は、コア・ファイルは無視されることに注意してください。したがって、ある時点までユーザ・プログラムを実行させた後に、コア・ファイルをデバッグしたくなったような場合、プログラムを実行しているサブ・プロセスを終了させなければなりません。サブ・プロセスの終了は、**kill** コマンドで行います ( see Section 4.8 [Killing the child process], page 23 )

**add-symbol-file** *filename* *address*

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

**add-symbol-file** コマンドは、*filename* で指定されるファイルから追加的なシンボル・テーブル情報を読み込みます。*filename* で指定されるファイルが ( 何か別の方法によって ) 実行中のプログラムの中に動的にロードされた場合に、このコマンドを使用します。*address* は、ファイルがロードされたメモリ・アドレスでなければなりません。GDB は独力でこのアドレスを知ることはできません。*address* は式として指定することもできます。

*filename* で指定されるファイルのシンボル・テーブルは、もともと **symbol-file** コマンドによって読み込まれたシンボル・テーブルに追加されます。**add-symbol-file** コマンドは何回でも使用することができます。新たに読み込まれたシンボル・テーブルのデータは、古いデータに追加されていきます。古いシンボル・データをすべて破棄するには、**symbol-file** コマンドを使用してください。

**add-symbol-file** コマンドを実行した後に (**RET**) キーを押しても、**add-symbol-file** コマンドは繰り返し実行されません。

**symbol-file** コマンドと同様、‘*-mapped*’オプションと ‘*-readnow*’オプションを使用して、*filename* で指定されるファイルのシンボル・テーブル情報を GDB がどのように管理するかを変更することができます。

**add-shared-symbol-file**

**add-shared-symbol-file** コマンドは、Motorola 88k 用の Harris’ CXUX オペレーティング・システム上でのみ使用することができます。GDB は自動的に共有ライブラリを探しますが、GDB がユーザの共有ライブラリを見つけてくれない場合には、**add-shared-symbol-file** コマンドを実行できます。このコマンドは引数を取りません。

**section** **section** コマンドは、実行ファイルの *section* セクションのベース・アドレスを *addr* に変更します。これは、( *a.out* フォーマットのように ) 実行ファイルがセクション・アドレスを保持していない場合や、ファイルの中で指定されているアドレスが誤っている場合に使うことができます。個々のセクションは、個別に変更されなければなりません。

‘info files’コマンドによって、すべてのセクションとそのアドレスを一覧表示することができます。

```
info files
info target
```

info files と info target は同義です。両方とも、カレント・ターゲット ( see Chapter 13 [Specifying a Debugging Target], page 105 ) に関する情報を表示します。表示される情報には、GDB が現在使用中の実行ファイルやコア・ダンプ・ファイルの名前、シンボルがそこからロードされたファイルの名前を含みます。help target コマンドは、カレントなターゲットではなく、すべての可能なターゲットを一覧表示します。

ファイルを指定するすべてのコマンドは、引数として、絶対パスによるファイル名と相対パスによるファイル名のどちらでも受け付けます。GDB は、常にファイル名を絶対パス名に変換して、絶対パスの形で記憶します。

GDB は、HP-UX、SunOS、SVr4、Irix 5、IBM RS/6000 の共有ライブラリをサポートします。ユーザが run コマンドを実行したり、コア・ファイルを調べようとする、GDB は自動的に共有ライブラリからシンボル定義をロードします ( ユーザが run コマンドを発行するまでは、共有ライブラリ内部の関数への参照があっても、GDB にはそれを理解することができません。コア・ファイルをデバッグしている場合は、この限りではありません )。

```
info share
info sharedlibrary
```

現在ロードされている共有ライブラリの名前を表示します。

```
sharedlibrary regex
share regex
```

UNIX の正規表現にマッチするファイルに対応する、共有オブジェクト・ライブラリのシンボルをロードします。自動的にロードされるファイルと同様、ユーザ・プログラムによってコア・ファイルのために必要とされる共有ライブラリ、または run コマンド実行時に必要とされる共有ライブラリだけがロードされます。regex が省略されると、ユーザ・プログラムによって必要とされるすべての共有ライブラリがロードされます。

## 12.2 シンボル・ファイル読み込み時のエラー

シンボル・ファイルの読み込み中に、GDB はときどき問題にぶつかることがあります。例えば、認識できないシンボル・タイプを見つけたり、コンパイラの出力に既知の問題を発見することがあります。デフォルトでは、このようなエラーがあったことを、GDB はユーザに知らせません。なぜなら、このようなエラーは比較的によく見られるものであり、コンパイラのデバッグをしているような人々だけが関心を持つようなものだからです。もし、正しく構築されていないシンボル・テーブルに関する情報を見ることに関心があれば、set complaints コマンドを使用することで、何回問題が発生しようと個々のタイプの問題について 1 回だけメッセージを出力するよう指示することができますし、また、何回問題発生したかを見るためにより多くのメッセージを表示するよう指示することもできます ( see Section 14.6 [Optional warnings and messages], page 134 )。

現在のバージョンで表示されるメッセージとその意味を以下に記します。

```
inner block not inside outer block in symbol
```

シンボル情報は、シンボルのスコープの先頭と末尾の位置を示します ( 例えば、ある関数の先頭、あるいは、ブロックの先頭など )。このエラーは、内側のスコープのブロックが、外側のスコープのブロックに完全に包含されていないことを意味しています。



GDB は、内側のブロックが外側のブロックと同一のスコープを持つものとして扱うことで、この問題を回避します。外側のブロックが関数でない場合には、エラー・メッセージの *symbol* の部分が `'(don't know)'` のように表示されることがあります。

`block at address out of order`

シンボルのスコープとなるブロックに関する情報は、アドレスの低い方から昇順に並んでいなければなりません。このエラーは、そうなっていないことを示しています。GDB はこの問題を回避することはず、読み込もうとしているソース・ファイルのシンボルを見つけるのに支障が出ます (`set verbose on` を指定することで、どのソース・ファイルが関係しているかを知ることができます。See Section 14.6 [Optional warnings and messages], page 134 )。

`bad block start address patched`

シンボルのスコープとなるブロックに関する情報の中の開始アドレスが、1 つ前のソース行のアドレスより小さい値です。これは、SunOS 4.1.1 ( および、それ以前のバージョン ) の C コンパイラで発生することが分かっています。

GDB は、シンボルのスコープとなるブロックが 1 つ前のソース行から始まるものとして扱うことによって、この問題を回避します。

`bad string table offset in symbol n`

シンボル番号 *n* のシンボルが持っている文字列テーブルへのポインタが、文字列テーブルのサイズを超える値です。

GDB は、このシンボルが `foo` という名前を持つものとみなすことによって、この問題を回避します。この結果、多くのシンボルが `foo` という名前を持つことになってしまうと、他の問題が発生する可能性があります。

`unknown symbol type 0xnn`

シンボル情報の中に、どのようにして読み取ればよいのか GDB には分からないような、新しいデータ型が含まれています。0xnn は理解できなかったシンボルの型を 16 進数で表わしたものです。

GDB は、このようなシンボル情報を無視することによって、このエラーを回避します。通常、プログラムのデバッグを行うことは可能になりますが、ある特定のシンボルにアクセスすることができなくなります。このような問題にぶつかり、それをデバッグしたいのであれば、gdb 自身を使って gdb をデバッグすることができます。この場合、シンボル `complain` にブレイクポイントを設定し、関数 `read_dbx_syntab` まで実行してから、`*bufp` によってシンボルを参照します。

`stub type has NULL name`

GDB は、ある構造体またはクラスに関する完全な定義を見つけることができませんでした。

`const/volatile indicator missing (ok if using g++ v1.x), got...`

ある C++ のメンバ関数に関するシンボル情報に、より新しいコンパイラを使用した場合には生成されるいくつかの情報が欠けています。

`info mismatch between compiler and debugger`

GDB は、コンパイラが生成した型の指定を解析できませんでした。



## 13 デバッグ・ターゲットの指定

ターゲットとは、ユーザ・プログラムが持つ実行環境を指します。多くの場合、GDB はユーザ・プログラムと同一のホスト環境上で実行されます。この場合には、`file` コマンドや `core` コマンドを実行すると、その副作用としてデバッグ・ターゲットが指定されます。例えば、物理的に離れた位置にあるホスト・マシン上で GDB を実行したい場合や、シリアル・ポート経由でスタンドアロン・システムを制御したい場合、または、TCP/IP 接続を利用してリアルタイム・システムを制御したい場合などのように、より多くの柔軟性が必要とされる場合、`target` コマンドを使うことによって、GDB に設定されたターゲットの種類の中から 1 つを指定することができます ( see Section 13.2 [Commands for managing targets], page 105 )。

### 13.1 アクティブ・ターゲット

ターゲットには 3 つのクラスがあります。プロセス、コア・ファイル、そして、実行ファイルです。GDB は同時に、1 クラスにつき 1 つ、全体で最高で 3 つまでアクティブなターゲットを持つことができます。これにより、( 例えば ) コア・ファイルに対して行ったデバッグ作業を破棄することなく、プロセスを起動してその動作を調べることができます。

例えば、`'gdb a.out'` を実行すると、実行ファイル `a.out` が唯一のアクティブなターゲットになります。コア・ファイル ( おそらくは、前回実行したときにクラッシュしてコア・ダンプしたもの ) を併せて指定すると、GDB は 2 つのターゲットを持ち、メモリ・アドレスを知る必要がある場合には、それを知るために 2 つのターゲットを並行して使用します。この場合、まずコア・ファイルを参照し、次に実行ファイルを参照します。( 典型的には、これら 2 つのクラスのターゲットは相互に補完的です。というのも、コア・ファイルには、プログラムが持っている変数などの読み書き可能なメモリ域の内容とマシン・ステータスだけがあり、実行ファイルには、プログラムのテキストと初期化されたデータだけがあるからです )。

`run` コマンドを実行すると、ユーザの実行ファイルはアクティブなプロセス・ターゲットにもなります。プロセス・ターゲットがアクティブな間は、メモリ・アドレスを要求するすべての GDB コマンドは、プロセス・ターゲットを参照します。アクティブなコア・ファイル・ターゲットや実行ファイル・ターゲットの中のアドレスは、プロセス・ターゲットがアクティブな間は、隠された状態になります。

新しいコア・ファイル・ターゲットや実行ファイル・ターゲットを選択するには、`core-file` コマンドや `exec-file` コマンドを使用します ( see Section 12.1 [Commands to specify files], page 99 )。既に実行中のプロセスをターゲットとして指定するには、`attach` コマンドを使用します ( see Section 4.7 [Debugging an already-running process], page 23 )。

### 13.2 ターゲットを管理するコマンド

#### `target type parameters`

GDB のホスト環境をターゲット・マシンまたはターゲット・プロセスに接続します。ターゲットとは、典型的には、デバッグ機能と通信するためのプロトコルを指します。引数 `type` によって、ターゲット・マシンの種類またはプロトコルを指定します。

`parameters` はターゲット・プロトコルによって解釈されるものですが、典型的には、接続すべきデバイス名やホスト名、プロセス番号、ボーレートなどが含まれます。

`target` コマンドを実行した後に `(RET)` キーを押しても、`target` コマンドは再実行されません。

**help target**

利用可能なすべてのターゲットの名前を表示します。現在選択されているターゲットを表示させるには、`info target` コマンドまたは `info files` コマンドを使用します ( see Section 12.1 [Commands to specify files], page 99 )

**help target *name***

ある特定のターゲットに関する説明を表示します。選択時に必要となるパラメータも表示されます。

**set gnutarget *args***

GDB は、自分で持っているライブラリ BFD を使用してユーザ・ファイルを読み込みます。GDB は、実行ファイル、コア・ファイル、`.o` ファイルのどれを自分が読み込んでいるのかを知っています。しかし、`set gnutarget` コマンドを使用して、ファイルのフォーマットを指定することもできます。ほとんどの `target` コマンドとは異なり、`gnutarget` における `target` は、マシンではなくプログラムです。

注意: `set gnutarget` でファイル・フォーマットを指定するには、実際の BFD 名を知っている必要があります。

See Section 12.1 [Commands to specify files], page 99.

**show gnutarget**

`gnutarget` がどのようなファイル・フォーマットを読むよう設定されているかを表示させるには、`show gnutarget` コマンドを使用します。`gnutarget` を設定していない場合、個々のファイルのフォーマットを GDB が自動的に決定します。この場合、`show gnutarget` を実行すると 'The current BDF target is "auto"' と表示されます。

以下に、一般的なターゲットをいくつか示します ( GDB の構成によって、利用可能であったり利用不可であったりします )

**target exec *program***

実行ファイルです。'target exec *program*' は 'exec-file *program*' と同じです。

**target core *filename***

コア・ダンプ・ファイルです。'target core *filename*' は 'core-file *filename*' と同じです。

**target remote *dev***

GDB 固有のプロトコルによる、リモートのシリアル・ターゲットです。引数 *dev* によって、接続を確立するために使用するシリアル装置 (例えば、'/dev/ttya') を指定します。See Section 13.4 [Remote debugging], page 110. `target remote` は、`load` コマンドもサポートするようになりました。これは、スタブをターゲット・システム上に持っていく方法が別にあり、かつ、ダウンロードが実行されたときに破壊されないようなメモリ域にそれを置くことができる場合にのみ役に立ちます。

**target sim**

CPU シミュレータです。See Section 13.4.10 [Simulated CPU Target], page 128.

以下のターゲットはすべて、特定の CPU に固有のものであり、特定の構成においてのみ利用可能です。

**target abug *dev***

M68K 用の ABug ROM モニタです。

`target adapt dev`  
A29K 用の Adapt モニタです。

`target amd-eb dev speed PROG`  
シリアル回線により接続されている、リモートの PC に組み込まれた AMD EB29K ボードです。 `target remote` の場合と同様、 `dev` はシリアル装置です。 `speed` によって回線速度を指定することができます。 `PROG` は、デバッグ対象となるプログラムを PC 上の DOS から見た場合の名前です。 See Section 13.4.4 [The EBMON protocol for AMD29K], page 119.

`target array dev`  
Array Tech LSI33K RAID コントローラ・ボードです。

`target bug dev`  
MVME187 ( m88k ) ボード上で動作する BUG モニタです。

`target cpu32bug dev`  
CPU32 ( M68K ) ボード上で動作する CPU32BUG モニタです。

`target dbug dev`  
Motorola ColdFire 用の dBUG ROM モニタです。

`target ddb dev`  
Mips Vr4300 用の NEC DDB モニタです。

`target dink32 dev`  
PowerPC 用の DINK32 ROM モニタです。

`target e7000 dev`  
日立 H8、SH 用の E7000 エミュレータです。

`target es1800 dev`  
M68K 用の ES-1800 エミュレータです。

`target est dev`  
CPU32 ( M68K ) ボード上で動作する EST-300 ICE モニタです。

`target hms dev`  
ユーザのホストにシリアル回線で接続された日立の SH、H8/300、H8/500 ボードです。特別なコマンドである `device` と `speed` によって、使用されるシリアル回線と通信速度を制御します。 See Section 13.4.8 [GDB and Hitachi Microprocessors], page 125.

`target lsi dev`  
Mips 用の LSI ROM モニタです。

`target m32r dev`  
三菱 M32R/D ROM モニタです。

`target mips dev`  
Mips 用の IDT/SIM ROM モニタです。

`target mon960 dev`  
Intel i960 用の MON960 モニタです。

`target nindy devicename`  
Nindy Monitor により制御される Intel 960 ボードです。 `devicename` は、接続に使用するシリアル装置の名前です。例えば `‘/dev/ttya’` です。 See Section 13.4.2 [GDB with a remote i960 (Nindy)], page 117.

`target nrom dev`  
NetROM ROM エミュレータです。このターゲットは、ダウンロードのみサポートしています。

`target op50n dev`  
OKI HPPA ボード上で動作する OP50N モニタです。

`target pmon dev`  
Mips 用の PMON ROM モニタです。

`target ppcbug dev`  
`target ppcbug1 dev`  
PowerPC 用の PPCBUG ROM モニタです。

`target r3900 dev`  
東芝 R3900 Mips 用の Densan DVE-R3900 ROM モニタです。

`target rdi dev`  
RDI ライブラリ・インターフェイスを経由した ARM Angel モニタです。

`target rdp dev`  
ARM Demon モニタです。

`target rom68k dev`  
M68K IDP ボード上で動作する ROM 68K モニタです。

`target rombug dev`  
OS/9000 用の ROMBUG ROM モニタです。

`target sds dev`  
( Motorola の ADS などの ) PowerPC ボード上で動作する SDS モニタです。

`target sparclite dev`  
ロードするためだけの目的で使用される、富士通の sparclite ボードです。プログラムをデバッグするためには、さらに別のコマンドを使用しなければなりません。一例を挙げると、GDB の標準的なリモート・プロトコルを使用する `target remote dev` です。

`target sh3 dev`  
`target sh3e dev`  
日立 SH-3、SH-3E ターゲット・システムです。

`target st2000 dev speed`  
Tandem STDEBUG プロトコルを実行している Tandem ST2000 電話交換機です。`dev` は、ST2000 のシリアル回線に接続されている装置の名前です。`speed` は通信回線の速度です。GDB が ST2000 に TCP または Telnet で接続するよう構成されている場合、引数は使用されません。See Section 13.4.5 [GDB with a Tandem ST2000], page 121。

`target udi keyword`  
AMD UDI プロトコルを使用する Remote AMD29K ターゲットです。引数 `keyword` が、使用する 29K ボードまたはシミュレータを指定します。See Section 13.4.3 [The UDI protocol for AMD29K], page 119。

`target vxworks machinename`  
TCP/IP で接続された VxWorks システムです。引数 `machinename` は、ターゲット・システムのマシン名または IP アドレスです。See Section 13.4.6 [GDB and VxWorks], page 122。

```
target w89k dev
```

Winbond HPPA ボード上で動作する W89K モニタです。

GDB の構成によって、利用可能なターゲットも異なるものになります。構成次第で、ターゲットの数は多くなったり少なくなったりします。

多くのリモート・ターゲットでは、接続に成功すると、実行プログラムのコードをダウンロードすることが必要となります。

```
load filename
```

構成によって GDB に組み込まれたリモート・デバッグ機能によっては、load コマンドが使用可能になります。これが利用可能な場合、実行ファイル *filename* が (例えば、ダウンロードやダイナミック・リンクによって) リモート・システム上でデバッグできるようになることを意味します。また、load コマンドは add-symbol-file コマンドと同様、ファイル *filename* のシンボル・テーブルを GDB 内に記録します。

GDB が load コマンドを提供していない場合、それを実行しようとすると「You can't do that when your target is ...」というエラー・メッセージが表示されます。

実行ファイルの中で指定されたアドレスに、ファイルはロードされます。オブジェクト・ファイルのフォーマットによっては、プログラムをリンクするときに、ファイルをロードするアドレスを指定できるものもあります。これ以外のフォーマット (例えば、a.out) では、オブジェクト・ファイルのフォーマットによって自動的にアドレスが指定されます。

VxWorks で load コマンドを実行すると、*filename* で指定される実行ファイルがカレントなターゲット・システム上で動的にリンクされ、シンボルが GDB に追加されます。

Intel 960 ボードの Nindy インターフェイスでは、load コマンドは *filename* で指定されるファイルを 960 側にダウンロードし、そのシンボルを GDB に追加します。

日立の SH、H8/300、H8/500 ボード (see Section 13.4.8 [GDB and Hitachi Microprocessors], page 125) に対するリモート・デバッグを選択すると、load コマンドはユーザ・プログラムを日立ボードにダウンロードし、(file コマンドと同様) ユーザのホスト・マシン上の GDB のカレントなターゲット実行ファイルとしてオープンします。

load コマンドを実行した後に **(RET)** キーを押しても、load コマンドは繰り返し実行されません。

### 13.3 ターゲットのバイト・オーダの選択

MIPS、PowerPC、Hitachi SH などのプロセッサは、ビッグ・エンディアン、リトル・エンディアンのどちらのバイト・オーダでも実行することができます。通常は、実行ファイルまたはシンボルの中に、エンディアン種別を指定するビットがあるので、どちらを使用するかを気にする必要はありません。しかし、GDB の認識しているプロセッサのエンディアン種別を手作業で調整することができれば、便利なこともあるでしょう。

```
set endian big
```

GDB に対して、ターゲットはビッグ・エンディアンであると想定するよう指示します。

```
set endian little
```

GDB に対して、ターゲットはリトル・エンディアンであると想定するよう指示します。

```
set endian auto
```

GDB に対して、実行ファイルに関連付けされているバイト・オーダを使用するよう指示します。

`show endian`

GDB が認識している、ターゲットの現在のバイト・オーダ種別を表示します。

これらのコマンドは、ホスト上でのシンボリック・データの解釈を調整するだけであり、ターゲット・システムに対しては全く何の影響も持たないということに注意してください。

## 13.4 リモート・デバッグ

通常の方法で GDB を実行させることのできないマシン上で実行中のプログラムをデバッグするには、リモート・デバッグ機能を使うのが便利です。例えば、オペレーティング・システムのカーネルのデバッグや、フル機能を持つデバッガを実行するのに十分な機能を持つ汎用的なオペレーティング・システムを持たない小規模なシステムでのデバッグでは、ユーザはリモート・デバッグ機能を使うことになるかもしれません。

GDB は、その構成によっては、特別なシリアル・インターフェイスや TCP/IP インターフェイスを持ち、これを特定のデバッグ・ターゲット用に使用することができます。さらに、GDB には汎用的なシリアル・プロトコルが組み込まれており ( GDB 固有のもので、特定のターゲット・システムに固有なものではありません ) リモート・スタブを作成すれば、これを使用することができます。リモート・スタブとは、GDB と通信するためにリモート・システム上で動作するコードです。

GDB の構成によっては、他のリモート・ターゲットが利用可能な場合もあります。利用可能なリモート・ターゲットを一覧表示させるには、`help target` コマンドを使用します。

### 13.4.1 GDB リモート・シリアル・プロトコル

他のマシン上で実行中のプログラムをデバッグするには ( ターゲット・マシンをデバッグするには )、そのプログラムを単独で実行するために通常必要となる事前条件をすべて整える必要があります。例えば、C のプログラムの場合、

1. C の実行環境をセットアップするためのスタートアップ・ルーチンが必要です。これは通常 `'crt0'` のような名前を持っています。スタートアップ・ルーチンは、ハードウェアの供給元から提供されることもありますし、ユーザが自分で書かなければならないこともあります。
2. ユーザ・プログラムからのサブルーチン呼び出しをサポートするために、入出力の管理などを行う C のサブルーチン・ライブラリが必要になるかもしれません。
3. ユーザ・プログラムを他のマシンに持っていく手段、例えばダウンロード・プログラムが必要です。これはハードウェアの供給元から提供されることが多いのですが、ハードウェアのドキュメントをもとにユーザが自分で作成しなければならないこともあります。

次に、ユーザ・プログラムがシリアル・ポートを使って、GDB を実行中のマシン ( ホスト・マシン ) と通信できるように準備します。一般的には、以下のような形になります。

ホスト上では：

GDB は既にこのプロトコルの使い方を理解しています。他の設定がすべて終了した後、単に `'target remote'` コマンドを使用するだけです ( see Chapter 13 [Specifying a Debugging Target], page 105 )。

ターゲット上では：

ユーザ・プログラムに、GDB リモート・シリアル・プロトコルを実装した特別なサブルーチンをいくつかリンクする必要があります。これらのサブルーチンを含むファイルは、デバッグ・スタブと呼ばれます。



特定のリモート・ターゲットでは、ユーザ・プログラムにスタブをリンクする代わりに、gdbserverという補助プログラムを使うこともできます。詳細については、See Section 13.4.1.5 [Using the gdbserver program], page 115。

デバッグ・スタブはリモート・マシンのアーキテクチャに固有のものです。例えば、SPARC ボード上のプログラムをデバッグするには 'sparc-stub.c' を使います。

以下に実際に使えるスタブを列挙します。これらは、GDB とともに配布されています。

i386-stub.c

Intel 386 アーキテクチャ、およびその互換アーキテクチャ用です。

m68k-stub.c

Motorola 680x0 アーキテクチャ用です。

sh-stub.c

日立 SH アーキテクチャ用です。

sparc-stub.c

SPARC アーキテクチャ用です。

sparcl-stub.c

富士通 SPARCLITE アーキテクチャ用です。

GDB とともに配布される README ファイルには、新しく追加された他のスタブのことが記されているかもしれません。

#### 13.4.1.1 スタブの提供する機能

各アーキテクチャ用のデバッグ・スタブは、3 つのサブルーチンを提供します。

set\_debug\_traps

このルーチンは、ユーザ・プログラムが停止したときに handle\_exception が実行されるよう設定します。ユーザ・プログラムは、その先頭付近でこのサブルーチンを明示的に呼び出さなければなりません。

handle\_exception

これが中心的な仕事をする部分ですが、ユーザ・プログラムはこれを明示的には呼び出しません。セットアップ・コードによって、トラップが発生したときに handle\_exception が実行されるよう設定されます。

ユーザ・プログラムが実行中に（例えば、ブレイクポイントで）停止すると、handle\_exception が制御権を獲得し、ホスト・マシン上の GDB との通信を行います。これが、通信プロトコルが実装されている部分です。handle\_exception は、ターゲット・マシン上で GDB の代理として機能します。それはまず、ユーザ・プログラムの状態に関する情報を要約して送ることから始めます。次に、GDB が必要とする情報を入手して転送する処理を続けます。これは、ユーザ・プログラムの実行を再開させるような GDB コマンドが実行されるまで続きます。そのようなコマンドが実行されると、handle\_exception は、制御をターゲット・マシン上のユーザ・コードに戻します。

breakpoint

ユーザ・プログラムにブレイクポイントを持たせるには、この補助的なサブルーチンを使います。特定の状況においては、これが GDB が制御を獲得する唯一の方法です。例えば、ユーザのターゲット・マシンに割り込みを発生させるボタンのようなものがあれば、

このサブルーチン呼び出す必要はありません。割り込みボタンを押すことで、制御は `handle_exception` に、つまり事実上 GDB に渡されます。マシンによっては、シリアル・ポートから文字を受け取るだけでトラップが発生することもあります。このような場合には、ユーザ・プログラム自身から `breakpoint` を呼び出す必要はなく、ホストの GDB セッションから `'target remote'` を実行するだけで制御を得ることができます。これらのどのケースにも該当しない場合、あるいは、デバッグ・セッションの開始箇所としてあらかじめ決めてあるところでユーザ・プログラムが停止することを単に確実にしたいのであれば、`breakpoint` を呼び出してください。

### 13.4.1.2 スタブに対する必須作業

GDB とともに配布されるデバッグ用スタブは、特定のチップのアーキテクチャ用にセットアップされたものですが、デバッグのターゲット・マシンに関してそれ以外の情報は持っていません。

まず最初に、どのようにしてシリアル・ポートと通信するかをスタブに教えてやる必要があります。

```
int getDebugChar()
```

シリアル・ポートから単一文字を読み込むサブルーチンとしてこれを書きます。これは、ターゲット・システム上の `getchar` と同一かもしれません。これら 2 つを区別したい場合を考慮して、異なる名前が使われています。

```
void putDebugChar(int)
```

シリアル・ポートに単一文字を書き込むサブルーチンとしてこれを書きます。これは、ターゲット・システム上の `putchar` と同一かもしれません。これら 2 つを区別したい場合を考慮して、異なる名前が使われています。

実行中のユーザ・プログラムを GDB が停止できるようにしたいのであれば、割り込み駆動型のシリアル・ドライバを使用して、`^C` ( `control-C` 文字、すなわち `'\003'` ) を受信したときに停止するよう設定する必要があります。GDB はこの文字を使って、リモート・システムに対して停止するよう通知します。

デバッグ・ターゲットが適切なステータス情報を GDB に対して返せるようにするためには、おそらく標準のスタブを変更する必要があるでしょう。最も美しくなく、しかし最も手取り早くこれを実現する方法は、ブレイクポイント命令を実行することです ( この方法が「美しい」のは、GDB が `SIGINT` ではなく `SIGTRAP` を報告してくる点にあります )。

ユーザが提供する必要のあるルーチンには、ほかに以下のようなものがあります。

```
void exceptionHandler (int exception_number, void *exception_address)
```

例外処理テーブルに `exception_address` を組み込むよう、この関数を書きます。ユーザがこれを提供しなければならないのは、スタブにはターゲット・システム上の例外処理テーブルがどのようなものになるかを知る手段がないからです ( 例えば、プロセッサのテーブルは ROM 上にあり、その中のエントリが RAM 上のテーブルを指す、という形になっているかもしれません )。 `exception_number` は例外番号で、これは変更される必要があります。例外番号の意味は、アーキテクチャに依存します ( 例えば、0 による除算、境界を無視したメモリ・アクセス等は、異なる番号によって表わされるかもしれません )。この例外が発生したとき、制御は直接 `exception_address` に渡されなければならない。また、プロセッサの状態 ( スタック、レジスタなど ) はプロセッサ例外発生したときの状態と同じでなければなりません。したがって、`exception_address` に到達するのにジャンプ命令を使用したいのであれば、サブルーチン・ジャンプではなく、ただのジャンプ命令を使わなければなりません。

386 では、ハンドラが実行されているときに割り込みがマスクされるよう、*exception\_address* は割り込みゲートとして組み込まれる必要があります。そのゲートは特権レベル 0（最も高いレベル）でなければなりません。SPARC 用のスタブや 68k 用のスタブは、*exceptionHandler* の助けを借りなくても自分で割り込みをマスクすることができます。

```
void flush_i_cache()
```

（sparc、sparclite のみ）ターゲット・マシンに命令キャッシュがある場合、それをフラッシュするようこのサブルーチンを書きます。命令キャッシュがない場合には、このサブルーチンは何もしないものになるかもしれません。

命令キャッシュを持つターゲット・マシン上の GDB は、ユーザ・プログラムが安定した状態にあることがこの関数によって保証されることを必要とします。

また、次のライブラリ・ルーチンが使用可能であることを確かめなければなりません。

```
void *memset(void *, int, int)
```

あるメモリ領域に既知の値を設定する標準ライブラリ関数 *memset* です。フリーの *libc.a* を持っていれば、そこに *memset* があります。フリーの *libc.a* がなければ、*memset* をハードウェアの供給元から入手するか、自分で作成する必要があります。

GNU C コンパイラを使っていないのであれば、他の標準ライブラリ・サブルーチンも必要になるかもしれません。これは、スタブによっても異なりますが、一般的にスタブは、gcc がインライン・コードとして生成する共通ライブラリ・サブルーチンを使用する可能性があります。

### 13.4.1.3 ここまでのまとめ

要約すると、ユーザ・プログラムをデバッグする準備が整った後、以下の手順に従わなければなりません。

1. 下位レベルのサポート・ルーチンがあることを確認します（see Section 13.4.1.2 [What you must do for the stub], page 112）

```
getDebugChar, putDebugChar,
flush_i_cache, memset, exceptionHandler.
```

2. ユーザ・プログラムの先頭付近に以下の行を挿入します。

```
set_debug_traps();
breakpoint();
```

3. 680x0 のスタブに限り、*exceptionHook* という変数を提供する必要があります。通常は、以下のように使います。

```
void (*exceptionHook)() = 0;
```

しかし、*set\_debug\_traps* が呼び出される前に、ユーザ・プログラム内のある関数を指すようこの変数を設定すると、トラップ（例えば、バス・エラー）で停止した後に GDB が処理を継続実行するときに、その関数が呼び出されます。*exceptionHook* によって指される関数は、1 つの引数付きで呼び出されます。それは、*int* 型の例外番号です。

4. ユーザ・プログラム、ターゲット・アーキテクチャ用の GDB デバッグ・スタブ、サポート・サブルーチンをコンパイルしリンクします。
5. ターゲット・マシンと GDB ホストとの間がシリアル接続されていることを確認します。また、ホスト上のシリアル・ポートの名前を調べます。
6. ターゲット・マシンにユーザ・プログラムをダウンロードし（あるいは、製造元の提供する手段によってターゲット・マシンにユーザ・プログラムを持っていき）起動します。

7. リモート・デバッグを開始するには、ホスト・マシン上で GDB を実行し、リモート・マシン上で実行中のプログラムを実行ファイルとして指定します。これにより、ユーザ・プログラムのシンボルとテキスト域の内容を見つける方法が GDB に通知されます。

次に `target remote` コマンドを使って通信を確立します。引数には、シリアル回線に接続された装置名または ( 通常はターゲットと接続されたシリアル回線を持つ端末サーバの ) TCP ポートを指定することで、ターゲット・マシンとの通信方法を指定します。例えば、`‘/dev/ttyb’` という名前の装置に接続されているシリアル回線を使うには、

```
target remote /dev/ttyb
```

とします。

TCP 接続を使うには、`host:port` という形式の引数を使用します。例えば、`manyfarms` という名前の端末サーバのポート 2828 に接続するには、

```
target remote manyfarms:2828
```

とします。

ここまでくると、データの値の調査、変更、リモート・プログラムのステップ実行、継続実行に通常使用するすべてのコマンドを使用することができます。

リモート・プログラムの実行を再開し、デバッグするのをやめるには、`detach` コマンドを使います。

GDB がリモート・プログラムを待っているときにはいつでも、割り込み文字 ( 多くの場合 `C-C` ) を入力すると、GDB はそのプログラムを停止しようとしています。これは成功することも失敗することもあります。その成否は、リモート・システムのハードウェアやシリアル・ドライバにも依存します。割り込み文字を再度入力すると、GDB は以下のプロンプトを表示します。

```
Interrupted while waiting for the program.
Give up (and stop debugging it)? (y or n)
```

ここで `y` を入力すると、GDB はリモート・デバッグ・セッションを破棄します ( 後になって再実行したくなった場合には、接続するために `‘target remote’` を再度使用します )。 `n` を入力すると、GDB は再び待ち状態になります。

#### 13.4.1.4 通信プロトコル

GDB とともに提供されるスタブ・ファイルは、ターゲット側の通信プロトコルを実装します。そして GDB 側の通信プロトコルは、GDB のソース・ファイル `‘remote.c’` に実装されています。通常は、これらのサブルーチンに通信処理を任せて、詳細を無視することができます ( 独自のスタブ・ファイルを作成するときでも、詳細については無視して、既存のスタブ・ファイルをもとにして作成を始めることができます。 `‘sparc-stub.c’` が最もよく整理されており、したがって最も読みやすくなっています )。

しかし、場合によっては、プロトコルについて何かを知る必要が出てくることもあるでしょう。例えば、ターゲット・マシンにシリアル・ポートが 1 つしかなく、GDB に対して送られてきたパケットを検出したときに、ユーザ・プログラムが何か特別なことをするようにしたい場合です。

( 単一文字による確認メッセージを除く ) すべての GDB コマンドとそれに対する応答は、チェックサムを含むパケットとして送信されます。パケットは、文字 `‘$’` で始まり、文字 `‘#’` に 2 桁のチェックサム値が続いて終わります。

```
$packet info#checksum
```

ここで、`checksum` は `packet info` のすべての文字の値を合計したものを 256 で割った余りとして計算されます。

ホスト・マシンまたはターゲット・マシンがパケットを受信したとき、最初に期待される応答は確認メッセージです。これは単一文字で、( パッケージが正しく受信されたことを示す ) '+' または ( 再送要求を示す ) '-' です。

ホスト ( GDB ) がコマンドを送信し、ターゲット ( ユーザ・プログラムに組み込まれたデバッグ・スタブ ) が応答としてデータを送信します。ターゲットは、ユーザ・プログラムが停止したときにも、データを送信します。

コマンド・パケットは最初の文字で区別されます。最初の文字がコマンドの種類を表わします。

以下に、現在サポートされているコマンドをいくつか列挙します ( コマンドの完全なリストについては 'gdb/remote.c' を参照してください )。

**g** CPU レジスタの値を要求します。

**G** CPU レジスタの値を設定します。

**maddr,count**

*addr* で示される位置から *count* で示されるバイト数を読み込みます。

**Maddr,count:...**

*addr* で示される位置から *count* で示されるバイト数を書き込みます。

**c**

**caddr** カレントなアドレス ( *addr* が指定されているのであれば、それによって指定されるアドレスから ) 実行を再開します。

**s**

**saddr** プログラム・カウンタの指すカレントな箇所から ( *addr* が指定されているのであれば、それによって指定されるアドレスから ) ターゲット・プログラムを 1 命令だけステップ実行します。

**k** ターゲット・プログラムを終了させます。

**?** 最後に受信したシグナルを報告します。GDB のシグナル処理コマンドを利用できるように、デバッグ・スタブの中のある関数が、CPU トラップを対応する POSIX シグナル値として報告してきます。

**T** リモートのスタブに対して、GDB がシングル・ステップ処理や条件付きブレイクポイントに関する迅速な決定を下すのに必要となるレジスタの情報だけを送信するようにさせます。これによって、ステップ実行中の 1 命令ごとにすべてのレジスタの情報を入手する必要がなくなります。

現在の GDB は、レジスタへのライト・スルー・キャッシュを実装していて、ターゲットが実行された場合のみ、レジスタを再度読み込みます。

シリアル接続に問題がある場合には、`set remotedebug` コマンドを使うことができます。これにより GDB は、シリアル回線経由でリモート・マシンとの間で送受信したすべてのパケットを報告するようになります。パケット・デバッグ用の情報は GDB の標準出力ストリームに表示されます。`set remotedebug off` によってこの設定が解除され、`show remotedebug` によって現在の設定が表示されます。

### 13.4.1.5 gdbserverプログラムの使用

`gdbserver` は、UNIX 系システム用の制御プログラムで、これにより、通常のデバッグ用スタブをリンクすることなく、`target remote` コマンドによって、ユーザ・プログラムをリモートの GDB に接続することができます。

gdbserverは、デバッグ用スタブに完全に取って代わるものではありません。gdbserverは、GDBが必要とするのと同様のオペレーティング・システムの機能を基本的には必要とするからです。実際、リモートの GDB と接続するために gdbserver を実行できるシステムであれば、GDB をローカルに実行することも可能です。それでも、gdbserver は GDB と比較するとかなりサイズが小さいので、便利なことがあります。また、gdbserver の移植は GDB 全体の移植よりも簡単なので、gdbserver を使うことで、新しいシステムでの作業をより早く開始することができます。最後に、リアル・タイム・システムの開発をしている場合、リアル・タイムな操作に関わるトレードオフのために、例えばクロス・コンパイルなどによって、他のシステム上で可能な限り多くの開発作業を行ったほうが便利であるということがあるでしょう。デバッグ作業に関しても、gdbserver を使うことでこれと同じような選択を行うことができます。

GDB と gdbserver は、シリアル回線または TCP 接続を経由して、標準的な GDB リモート・シリアル・プロトコルによって通信します。

ターゲット・マシンでは：

デバッグしたいプログラムのコピーが 1 つ必要です。gdbserver はユーザ・プログラムのシンボル・テーブルを必要とはしませんので、スペースの節約が必要であれば、プログラムをストリップすることができます。ホスト・システム上の GDB が、シンボルに関するすべての処理を実行します。

gdbserver を使うには、GDB との通信方法、ユーザ・プログラムの名前、ユーザ・プログラムへの引数を教えてやる必要があります。構文は、以下のとおりです。

```
target> gdbserver comm program [ args ... ]
```

*comm* は ( シリアル回線を使うための ) 装置名、あるいは、TCP のホスト名とポート番号です。例えば、'foo.txt' という引数を指定して Emacs をデバッグし、シリアル・ポート '/dev/com1' 経由で GDB と通信するには、以下のように実行します。

```
target> gdbserver /dev/com1 emacs foo.txt
```

gdbserver は、ホスト側の GDB が通信してくるのを受動的に待ちます。

シリアル回線の代わりに TCP 接続を使うには、以下のようにします。

```
target> gdbserver host:2345 emacs foo.txt
```

前の例との唯一の違いは第 1 引数です。これは、ホストの GDB と TCP によって接続することを指定しています。'host:2345' は、マシン 'host' からローカルの TCP ポート 2345 への TCP 接続を gdbserver が期待していることを意味します ( 現在のバージョンでは、'host' の部分は無視されます )。ターゲット・システム上で既に使われている TCP ポートでなければ、任意の番号をポート番号として選択できます ( 例えば、23 は telnet に予約されています )<sup>1</sup>。ここで指定したのと同じポート番号を、ホスト上の GDB の target remote コマンドで使わなければなりません。

GDB のホスト・マシンでは：

GDB はシンボル情報、デバッグ情報を必要とするので、ストリップされていないユーザ・プログラムのコピーが必要です。通常どおり、第 1 引数にユーザ・プログラムのローカル・コピーの名前を指定して GDB を起動します ( シリアル回線の速度が 9600 bps 以外であれば、'--baud' オプションも必要になります )。その後、target remote コマンドによって gdbserver との通信を確立します。引数には、装置名 ( 通常は '/dev/ttyb' のようなシリアル装置 ) または、host:PORT という形式での TCP ポート記述子指定します。例えば、

<sup>1</sup> 原注：他のサービスによって使用されているポート番号を選択すると、gdbserver はエラー・メッセージを出力して終了します。

```
(gdb) target remote /dev/ttyb
```

では、シリアル回線 `/dev/ttyb` を介して `gdbserver` と通信します。また、

```
(gdb) target remote the-target:2345
```

では、ホスト `the-target` 上のポート 2345 に対する TCP 接続によって通信します。TCP 接続を使う場合には、`target remote` コマンドを実行する前に、`gdbserver` を起動しておかなければなりません。そうしないと、エラーになります。エラー・テキストの内容はホスト・システムによって異なりますが、通常は `Connection refused` のような内容です。

#### 13.4.1.6 gdbserve.nlmプログラムの使用

`gdbserve.nlm` は NetWare システムでの制御プログラムです。これによって、`target remote` コマンドでユーザ・プログラムをリモートの GDB に接続することができます。

GDB と `gdbserve.nlm` は、標準の GDB リモート・シリアル・プロトコルを使って、シリアル回線経由で通信します。

ターゲット・マシンでは：

デバッグしたいプログラムのコピーが 1 つ必要です。`gdbserve.nlm` はユーザ・プログラムのシンボル・テーブルを必要としないので、スペースの節約が必要であれば、プログラムをストリップすることができます。ホスト・システム上の GDB が、シンボルに関わるすべての処理を実行します。

`gdbserve.nlm` を使うには、GDB との通信方法、ユーザ・プログラムの名前、ユーザ・プログラムの引数を教えてやる必要があります。構文は、以下のとおりです。

```
load gdbserve [ BOARD=board ] [ PORT=port ]
               [ BAUD=baud ] program [ args ... ]
```

`board` と `port` がシリアル回線を指定します。`baud` は接続に使われるボーレートを指定します。`port` と `node` のデフォルト値は 0、`baud` のデフォルト値は 9600 bps です。

例えば、`foo.txt` という引数を指定して Emacs をデバッグし、シリアル・ポート番号 2、ボード 1 を経由して 19200 bps の接続で GDB と通信するには、以下のように実行します。

```
load gdbserve BOARD=1 PORT=2 BAUD=19200 emacs foo.txt
```

GDB のホスト・マシンでは：

GDB はシンボル情報、デバッグ情報を必要とするので、ストリップされていないユーザ・プログラムのコピーが必要です。通常どおり、第 1 引数にユーザ・プログラムのローカル・コピーの名前を指定して GDB を起動します（シリアル回線の速度が 9600 bps 以外であれば、`--baud` オプションも必要になります）。その後、`target remote` コマンドによって `gdbserve.nlm` との通信を確立します。引数には、装置名（通常は `/dev/ttyb` のようなシリアル装置）を指定します。例えば、

```
(gdb) target remote /dev/ttyb
```

は、シリアル回線 `/dev/ttyb` を経由して `gdbserve.nlm` と通信します。

#### 13.4.2 GDB とリモート i960 (Nindy)

`Nindy` は、Intel 960 ターゲット・システム用の ROM Monitor プログラムです。`Nindy` を使ってリモートの Intel 960 を制御するよう GDB が構成されている場合、いくつかの方法によって GDB に 960 との接続方法を教えることができます。

- シリアル・ポート、Nindy プロトコルのバージョン、通信スピードを指定するコマンドライン・オプションによる方法
- 起動時のプロンプトに答える方法
- GDB セッション中の任意の時点で `target` コマンドを使う方法 ( See Section 13.2 [Commands for managing targets], page 105 )

#### 13.4.2.1 Nindy 使用時の起動方法

コマンドライン・オプションを一切使わずに `gdb` を起動すると、通常の GDB プロンプトが表示される前に、使用するシリアル・ポートを指定するよう促されます。

Attach /dev/ttyNN -- specify NN, or "quit" to quit:

このプロンプトに対して、使いたいシリアル・ポートを示す ( `/dev/tty` の後ろの ) サフィックスを入力します。もしそうしたいのであれば、プロンプトに空行で答えることによって、Nindy との接続を確立せずに起動することもできます。この場合、後に Nindy と接続したいときには `target` コマンドを使います ( see Section 13.2 [Commands for managing targets], page 105 )

#### 13.4.2.2 Nindy 用のオプション

接続された Nindy-960 ボードとの GDB セッションを開始するための起動オプションを以下に示します。

- `-r port`      ターゲット・システムとの接続に使用されるシリアル・インターフェイスのシリアル・ポート名を指定します。このオプションは、GDB が Intel 960 ターゲット・アーキテクチャ用に構成されているときのみ利用可能です。`port` は、完全なパス名 ( 例: `-r /dev/ttya` )、`/dev` 配下のデバイス名 ( 例: `-r ttya` )、`tty` 固有の一意なサフィックス ( 例: `-r a` ) のいずれによっても指定することができます。
- `-0`            ( ゼロではなく、英大文字の O です )。GDB がターゲット・システムと接続する際に、古い Nindy モニタ・プロトコルを使用すべきであることを指定します。このオプションは、GDB が Intel 960 ターゲット・アーキテクチャ用に構成されているときのみ利用可能です。
- 注意: `-0` を指定したにもかかわらず、実際にはより新しいプロトコルを期待しているターゲット・システムに接続しようとした場合、接続は失敗します。この失敗は、あたかも通信速度の不一致が原因であるかのように見えてしまいます。GDB は、異なる回線速度によって再接続を繰り返し試みます。割り込みによって、この処理を中断させることができます。
- `-brk`          接続する前に Nindy ターゲットをリセットするために、ターゲット・システムに対して最初に BREAK 信号を送信するよう、GDB に対して指定します。
- 注意: 多くのターゲット・システムは、このオプションが必要とするハードウェアを備えていません。このオプションは、少数のボードでしか機能しません。

標準の `-b` オプションが、シリアル・ポート上で使用される回線速度を制御します。

#### 13.4.2.3 Nindy reset コマンド



**reset**      ターゲットが Nindy である場合、このコマンドは BREAK 信号をリモートのターゲット・システムに送信します。これは、BREAK 信号を受信したときにハード・リセット（または、その他の興味深いアクション）を実行する回路がターゲットに備わっている場合にのみ役に立ちます。

### 13.4.3 AMD29K 用の UDI プロトコル

GDB は、a29k プロセッサ・ファミリをデバッグするための AMD UDI ( Universal Debugger Interface ) プロトコルをサポートしています。MiniMON モニタを実行する AMD ターゲットという構成を使うには、AMD 社から無料で入手可能な MONTIP プログラムが必要になります。また、AMD 社から入手可能な UDI 準拠の a29k シミュレータ・プログラム ISSTIP とともに GDB を使うこともできます。

**target udi keyword**

リモートの a29k ボードまたはシミュレータへの UDI インターフェイスを選択します。**keyword** は、AMD 構成ファイル 'udi\_soc' 内のエントリです。このファイルには、a29k ターゲットに接続するときに使われるパラメータを指定するキーワード・エントリが含まれます。'udi\_soc' ファイルが作業ディレクトリにない場合には、環境変数 'UDICONF' にそのパス名を設定しなければなりません。

### 13.4.4 AMD29K の EBMON プロトコル

AMD 社は、PC 組み込み用の 29K 開発ボードを、DOS 上で動作する EBMON というモニタ・プログラムとともに配布しています。この開発システムは、省略して EB29K と呼ばれます。UNIX システム上の GDB を使って EB29K ボード上でプログラムを実行するには、まず ( EB29K を組み込んだ ) PC と UNIX システムのシリアル・ポートの間をシリアル回線で接続しなければなりません。以下の節では、PC の 'COM1' ポートと UNIX システムの '/dev/ttya' との間をケーブルで接続してあるものと仮定します。

#### 13.4.4.1 通信セットアップ

PC 上の DOS で以下のように実行することによって、PC のポートをセットアップします。

```
C:\> MODE com1:9600,n,8,1,none
```

MS DOS 4.0 上で実行されているこの例では、PC ポートを通信速度 9600 bps、パリティ・ビットなし、データ・ビット数 8、ストップ・ビット数 1、リトライなしに設定しています。UNIX 側を設定する際には、同一の通信パラメータを使わなければなりません。

シリアル回線の UNIX 側に PC の制御権を与えるには、DOS コンソール上で以下のように実行します。

```
C:\> CTTY com1
```

( 後に、DOS コンソールに制御を戻したいときには、CTTY con コマンドを使うことができます。ただし、制御権を持っている装置からこのコマンドを送信する必要があります。ここでの例では、'COM1' に接続されているシリアル回線を通して送信することになります )

UNIX のホストからは、PC と通信するのに tip や cu のような通信プログラムを使います。以下に例を示します。

```
cu -s 9600 -l /dev/ttya
```

ここで示されている cu オプションはそれぞれ、使用する回線速度とシリアル・ポートを指定しています。tip コマンドを使った場合は、コマンドラインは以下のようなものになるでしょう。

```
tip -9600 /dev/ttya
```

ここで tip への引数として指定した '/dev/ttya' の部分には、システムによって異なる名前を指定する必要があるかもしれません。使用するポートを含む通信パラメータは、“remote” 記述ファイルにおいて tip コマンドへの引数と関連付けられます。通常このファイルは、システム・テーブル '/etc/remote' です。

tip 接続または cu 接続を使用して DOS の作業ディレクトリを 29K プログラムが存在するディレクトリに変更し、PC プログラム EBMON (AMD 社からボードとともに提供される EB29K 制御プログラム) を起動します。以下に示す例によく似た、EBMON プロンプト '#' で終わる EBMON の初期画面が表示されるはずです。

```
C:\> G:
```

```
G:\> CD \usr\joe\work29k
```

```
G:\USR\JOE\WORK29K> EBMON
```

```
Am29000 PC Coprocessor Board Monitor, version 3.0-18
```

```
Copyright 1990 Advanced Micro Devices, Inc.
```

```
Written by Gibbons and Associates, Inc.
```

```
Enter '?' or 'H' for help
```

```
PC Coprocessor Type   = EB29K
I/O Base              = 0x208
Memory Base           = 0xd0000
```

```
Data Memory Size      = 2048KB
Available I-RAM Range = 0x8000 to 0x1fffff
Available D-RAM Range = 0x80002000 to 0x801fffff
```

```
PageSize              = 0x400
Register Stack Size   = 0x800
Memory Stack Size     = 0x1800
```

```
CPU PRL               = 0x3
Am29027 Available     = No
Byte Write Available  = Yes
```

```
# ~.
```

続いて、cu プログラムまたは tip プログラムを終了させます (上の例では、EBMON プロンプトにおいて ~. を入力することで終了させています)。EBMON は、GDB が制御権を獲得できる状態で、実行を継続します。

この例では、PC と UNIX システムの両方に同一の 29K プログラムが確実に存在するようにするのに、おそらく最も便利であろうと思われる方法を使うことを仮定しました。それは、PC/NFS による接続で、UNIX ホストのファイル・システムの 1 つを PC の G: ドライブとする方法です。PC/NFS、あるいは、2 つのシステム間を接続する類似の方法がない場合、フロッピー・ディスクによる転送など、UNIX システムから PC へ 29K プログラムを転送するための他の手段を準備する必要があります。GDB は、シリアル回線経由で 29K プログラムをダウンロードすることはありません。

#### 13.4.4.2 EB29K クロス・デバッグ

最後に、UNIX システム上の 29K プログラムが存在するディレクトリに `cd` コマンドによって移動して、GDB を起動します。引数には、29K プログラムの名前を指定します。

```
cd /usr/joe/work29k
gdb myfoo
```

これで `target` コマンドが使えるようになります。

```
target amd-eb /dev/ttya 9600 MYF00
```

この例では、ユーザ・プログラムは `'myfoo'` と呼ばれるファイルであると仮定しています。 `target amd-eb` に対して最後の引数として指定するファイル名は、DOS 上でのプログラム名でなければならない点に注意してください。この例では単に `MYF00` となっていますが、DOS のパス名を含むこともできますし、転送メカニズムによっては、UNIX 側での名前とは似ても似つかないものになることもあります。

ここまでくると、好きなようにブレークポイントを設定することができます。29K ボード上でのプログラムの実行を監視する準備が整えば、GDB の `run` コマンドを使います。

リモート・プログラムのデバッグを停止するには、GDB の `detach` コマンドを使います。

PC の制御を PC コンソールに戻すには、GDB セッションが終了した後に、EBMON にアタッチするために、もう一度 `tip` または `cu` を使います。その後、`q` コマンドによって EBMON をシャットダウンし、DOS のコマンドライン・インタプリタに制御を戻します。CTTY `con` と入力して、入力されたコマンドがメインの DOS コンソールによって受け取られるようにし、`~.` を入力して `tip` または `cu` を終了させます。

#### 13.4.4.3 リモート・ログ

`target amd-eb` コマンドは、接続に関わる問題のデバッグを支援するため、カレントな作業ディレクトリに `'eb.log'` というファイルを作成します。`'eb.log'` は、EBMON に送信されたコマンドのエコーを含む、EBMON からのすべての出力を記録します。別のウィンドウ内でこのファイルに対して `'tail -f'` を実行すると、EBMON に関わる問題や PC 側での予期せぬイベントを理解する助けになることがよくあります。

#### 13.4.5 GDB と Tandem ST2000

ST2000 をホスト・システムに接続する方法については、製造元のマニュアルを参照してください。ST2000 が物理的に接続されれば、それをデバッグ環境として確立するには、以下を実行します。

```
target st2000 dev speed
```

`dev` は通常、シリアル回線によって ST2000 と接続される `'/dev/ttya'` のようなシリアル装置の名前です。代わりに、`hostname:portnumber` という構文を使って（例えば、端末多重化装置経由で接続されたシリアル回線への）TCP 接続として `dev` を指定することもできます。

このターゲットに対して、`load` コマンドと `attach` コマンドは定義されていません。通常スタンダードアロンで操作している場合と同様、ST2000 にユーザ・プログラムをロードしなければなりません。GDB は（シンボルのような）デバッグ用の情報を、ホスト・コンピュータ上にある別のデバッグ・バージョンのプログラムから読みとります。

ST2000 での作業を支援するために、以下の補助的な GDB コマンドが利用可能です。

```
st2000 command
```

STDEBUG モニタに `command` を送信します。利用できるコマンドについては、製造元のマニュアルを参照してください。

**connect** STDEBUG コマンド・モニタに対して制御端末を接続します。STDEBUG の操作が終了した後、`(RET)~`。(Return キーに続いて、チルダとピリオドを入力) または、`(RET)~(C-d)` (Return キーに続いて、チルダと Control-D を入力) のいずれかを入力することによって GDB コマンド・プロンプトに戻ります。

### 13.4.6 GDB と VxWorks

開発者は、GDB を使用することによって、ネットワークに接続された VxWorks 端末上のタスクを、UNIX のホストから起動してデバッグすることができます。VxWorks シェルから起動され、既に実行中の状態のタスクをデバッグすることもできます。GDB は、UNIX ホスト上で実行されるコードと VxWorks ターゲット上で実行されるコードの両方を使います。gdb は、UNIX ホスト上にインストールされて実行されます ( ホスト上のプログラムをデバッグするのに使う GDB と区別するために、vxgdb という名前でインストールされることもあります )。

VxWorks-timeout args

すべての VxWorks ベースのターゲットが、vxworks-timeout オプションをサポートするようになりました。このオプションはユーザによってセットされるもので、args は、GDB が RPC の応答を待つ秒数を表わします。実際の VxWorks ターゲットが速度の遅いソフトウェア・シミュレータであったり、帯域の小さいネットワーク回線を介して遠距離にある場合などに使うとよいでしょう。

VxWorks との接続に関する以下の情報は、このマニュアルの作成時における最新の情報です。新しくリリースされた VxWorks では、手順が変更されているかもしれません。

VxWorks 上で GDB を使うためには、VxWorks カーネルを再構築して、VxWorks ライブラリ 'rdb.a' の中のリモート・デバッグ用のインターフェイス・ルーチンを組み込む必要があります。そのためには、VxWorks のコンフィギュレーション・ファイル 'configAll.h' の中で INCLUDE\_RDB を定義して、VxWorks カーネルを再構築します。この結果として生成されるカーネルには 'rdb.a' が組み込まれ、VxWorks の起動時にソース・デバッグ用のタスク tRdbTask が起動されます。VxWorks の構成や再構築に関する詳細については、製造元のマニュアルを参照してください。

VxWorks システム・イメージへの 'rdb.a' の組み込みが終わり、UNIX の実行ファイル・サーチ・パスに GDB の存在するパスを加えれば、GDB を実行するための準備は完了です。UNIX ホストから gdb ( インストールの方法によっては vxgdb ) を実行します。

GDB が起動されて、以下のプロンプトを表示します。

(vxgdb)

#### 13.4.6.1 VxWorks への接続

GDB の target コマンドによって、ネットワーク上の VxWorks ターゲットに接続します。tt というホスト名を持つターゲットに接続するには、以下のようになります。

(vxgdb) target vxworks tt

GDB は以下のようなメッセージを表示します。

Attaching remote machine across net...

Connected to tt.

続いて GDB は、最後に VxWorks ターゲットが起動されたときより後にロードされたオブジェクト・モジュールのシンボル・テーブルを読み込もうと試みます。GDB は、コマンドのサーチ・パスに含まれているディレクトリを探索することによって、これらのファイルを見つけます ( see Section 4.4 [Your program's environment], page 21 )。オブジェクト・ファイルを見つけない場合には、以下のようなメッセージを表示します。

```
prog.o: No such file or directory.
```

このような場合には、GDB の `path` コマンドによって適切なディレクトリを検索パスに加えてから、再度 `target` コマンドを実行します。

### 13.4.6.2 VxWorks ダウンロード

VxWorks ターゲットに接続済みの状態で、まだロードされていないオブジェクトをデバッグしたい場合には、GDB の `load` コマンドを使って UNIX から VxWorks へ追加的にファイルをダウンロードすることができます。`load` コマンドの引数として指定されたオブジェクト・ファイルは、実際には 2 回オープンされます。まず、コードをダウンロードするために VxWorks ターゲットによってオープンされ、次にシンボル・テーブルを読み込むために GDB によってオープンされます。2 つのシステム上のカレントな作業ディレクトリが異なると、問題が発生します。両方のシステムが同一のファイル・システムを NFS マウントしているのであれば、絶対パスを使うことで問題を回避することができます。そうでない場合は、両方のシステム上で、オブジェクト・ファイルが存在するディレクトリを作業ディレクトリにして、パスを一切使わずにファイル名だけでファイルを参照するのが、最も簡単でしょう。例えば、プログラム '`prog.o`' が、VxWorks では '`vxpath/vw/demo/rdb`' に存在し、ホストでは '`hostpath/vw/demo/rdb`' に存在するとしましょう。このプログラムをロードするには、VxWorks 上で以下のように実行します。

```
-> cd "vxpath/vw/demo/rdb"
```

GDB 上では、以下のように実行します。

```
(vxgdb) cd hostpath/vw/demo/rdb
```

```
(vxgdb) load prog.o
```

GDB は次のような応答を表示します。

```
Reading symbol data from wherever/vw/demo/rdb/prog.o... done.
```

ソース・ファイルを編集して再コンパイルした後に、`load` コマンドを使ってオブジェクト・モジュールを再ロードすることもできます。ただし、これを行うと、GDB はその時点で定義されているすべてのブレイクポイント、自動表示設定、コンビニエンス変数を削除し、値ヒストリを初期化してしまいますので、注意してください（これは、ターゲット・システムのシンボル・テーブルを参照するデバッガのデータ構造の完全性を保つために必要です）。

### 13.4.6.3 タスクの実行

以下のように `attach` コマンドを使うことで、既存のタスクにアタッチすることも可能です。

```
(vxgdb) attach task
```

`task` は、VxWorks の 16 進数のタスク ID です。アタッチするときに、タスクは実行中であってもサスペンドされていても構いません。実行中であったタスクは、アタッチされたときにサスペンドされます。

## 13.4.7 GDB と Sparclet

開発者は、GDB を使うことによって、Sparclet ターゲット上で実行中のタスクを Unix ホストからデバッグできるようになります。GDB は、Unix ホスト上で実行されるコードと Sparclet ターゲット上で実行されるコードの両方を使用します。`gdb` は、Unix ホスト上にインストールされて実行されます。

timeout args

GDB はオプション remotetimeout をサポートするようになりました。このオプションはユーザによって設定されるもので、args は GDB が応答を待つ秒数を表わします。

デバッグ用にコンパイルする際には、デバッグ情報を得るために "-g" オプションを、また、ターゲット上でロードしたい位置にプログラムを再配置するために "-Ttext" オプションを指定します。各セクションのサイズを小さくするために、"-n" または "-N" オプションを加えるのも良いでしょう。

```
sparclet-aout-gcc prog.c -Ttext 0x12010000 -g -o prog -N
```

アドレスが意図したものと一致しているかどうかを検証するのに、objdump を使うことができます。

```
sparclet-aout-objdump --headers --syms prog
```

GDB が見つかるように Unix の実行サーチ・パスを設定すれば、GDB を実行するための準備は完了です。Unix ホストから gdb ( インストールの方法によっては、sparclet-aout-gdb ) を実行します。

GDB が起動されて、以下のプロンプトを表示します。

```
(gdb) 
```

#### 13.4.7.1 デバッグするファイルの選択

GDB の file コマンドによって、デバッグするプログラムを選択することができます。

```
(gdb) file prog
```

このコマンドを実行すると、GDB は 'prog' のシンボル・テーブルを読み込もうとします。GDB は、コマンド・サーチ・パスに含まれるディレクトリを探索することによって、そのファイルを見つけます。そのファイルがデバッグ情報付き ( オプション "-g" ) でコンパイルされた場合は、ソース・ファイルも探します。GDB は、ディレクトリ・サーチ・パス ( see Section 4.4 [Your program's environment], page 21 ) に含まれるディレクトリを探索することによって、そのソース・ファイルを見つけます。ファイルが見つからない場合には、次のようなメッセージを表示します。

```
prog: No such file or directory.
```

このメッセージが表示された場合には、GDB の path コマンドと dir コマンドを使って適切なディレクトリをサーチ・パスに加えてから、target コマンドを再実行します。

#### 13.4.7.2 Sparclet への接続

GDB の target コマンドによって Sparclet ターゲットに接続することができます。シリアル・ポート "ttya" でターゲットに接続するには、以下のように入力します。

```
(gdb) target sparclet /dev/ttya
Remote target sparclet connected to /dev/ttya
main () at ../prog.c:3
```

GDB は以下のようなメッセージを表示します。

```
Connected to ttya.
```

#### 13.4.7.3 Sparclet ダウンロード

Sparclet ターゲットへの接続が完了すると、GDB の load コマンドを使って、ホストからターゲットへファイルをダウンロードすることができます。ファイル名とロード・オフセットを、load コマンドへの引数として渡さなければなりません。ファイル形式は aout ですので、プログラムはその

開始アドレスにロードされなければなりません。開始アドレスの値を知るには `objdump` を使うことができます。ロード・オフセットとは、ファイルの個々のセクションの VMA ( 仮想メモリ・アドレス ) に加算されるオフセットのことです。例えば、プログラム `'prog'` が、`text` セクションのアドレス `0x12010000`、`data` セクションのアドレス `0x12010160`、`bss` セクションのアドレス `0x12010170` にリンクされているとすると、GDB では以下のように入力します。

```
(gdb) load prog 0x12010000
Loading section .text, size 0xdb0 vma 0x12010000
```

プログラムがリンクされたアドレスとは異なるアドレスにコードがロードされた場合、どこにシンボル・テーブルをマップするかを GDB に通知するために、`section` コマンドと `add-symbol-file` コマンドを使う必要があるかもしれません。

#### 13.4.7.4 実行とデバッグ

以上により、GDB の実行制御コマンドである `b`、`step`、`run` などを使ってタスクのデバッグを開始することができます。コマンドの一覧については、GDB のマニュアルを参照してください。

```
(gdb) b main
Breakpoint 1 at 0x12010000: file prog.c, line 3.
(gdb) run
Starting program: prog
Breakpoint 1, main (argc=1, argv=0xeffff21c) at prog.c:3
3      char *symarg = 0;
(gdb) step
4      char *execarg = "hello!";
(gdb)
```

#### 13.4.8 GDB と日立のマイクロ・プロセッサ

日立の SH、H8/300、H8/500 と通信するためには、GDB は以下の情報を知っている必要があります。

1. ユーザは、日立マイクロ・プロセッサへのリモート・デバッグ用インターフェイスである `'target hms'` と、日立 SH や日立 300H のインサーキット・エミュレータである `target'e7000'` のどちらを使用したいかということ ( GDB が日立 SH、H8/300、H8/500 用に特に構成されている場合には、`'target hms'` がデフォルトです )
2. ホストと日立ボードを接続しているシリアル装置 ( デフォルトは、ホスト上で利用できる最初のシリアル装置です )
3. シリアル装置で使用する速度

##### 13.4.8.1 日立ボードへの接続

シリアル装置を明示的に指定する必要がある場合は、そのための専用コマンドである、gdb の `'device port'` コマンドを使用します。 `port` のデフォルトは、ホスト上で最初に利用可能なポートです。これは UNIX ホスト上でのみ必要であり、そこでは典型的には `'/dev/ttya'` という名前になります。

gdb には、通信速度を設定するための専用コマンド `'speed bps'` があります。このコマンドもまた UNIX ホストからのみ使用されるものです。DOS ホストでは通常どおり、GDB からではなく DOS の `mode` コマンドによって回線速度を設定します ( 例えば、9600 bps の接続を確立するには `'mode com2:9600,n,8,1,p'` のように実行します )

‘device’コマンドと‘speed’コマンドは、日立マイクロ・プロセッサ・プログラムのデバッグに UNIX ホストを使う場合のみ利用可能です。DOS ホストを使う場合、GDB は、PC のシリアル・ポート経由で開発ボードと通信するのに、`asynctsr`と呼ばれる補助的な常駐プログラムに依存します。DOS 側でシリアル・ポートの設定をする場合にも、DOS の `mode`コマンドを使わなければなりません。

#### 13.4.8.2 E7000 インサーキット・エミュレータの使用

E7000 インサーキット・エミュレータを使って、日立 SH または H8/300H 用のコードを開発することができます。‘target e7000’コマンドを以下のいずれかの形式で使って、GDB を E7000 に接続します。

`target e7000 port speed`

E7000 がシリアル・ポートに接続されている場合は、この形式を使ってください。引数 *port* が、使用するシリアル・ポートを指定します（例えば、‘com2’）。3 番目の引数は、秒あたりのビット数による回線速度です（例えば、‘9600’）。

`target e7000 hostname`

E7000 が TCP/IP ネットワーク上のホストとしてインストールされている場合、ホスト名だけを指定することもできます。GDB は接続に `telnet`を使います。

#### 13.4.8.3 日立マイクロ・プロセッサ用の特別な GDB コマンド

いくつかの GDB コマンドは、H8/300 または H8/500 用に構成された場合にのみ利用可能です。

`set machine h8300`

`set machine h8300h`

‘set machine’コマンドによって、2 種類の H8/300 アーキテクチャのどちらか一方にあわせて GDB を調整します。‘show machine’コマンドによって、現在有効なアーキテクチャを調べることができます。

`set memory mod`

`show memory`

‘set memory’コマンドによって、使用中の H8/500 メモリ・モデル (*mod*) を指定します。‘show memory’コマンドによって、現在有効なメモリ・モデルを調べます。*mod* に指定可能な値は、small、big、medium、compactのいずれかです。

#### 13.4.9 GDB とリモート MIPS ボード

GDB は、MIPS のリモート・デバッグ用のプロトコルを使って、シリアル回線に接続された MIPS ボードと通信することができます。これは、GDB を ‘--target=mips-idt-ecoff’によって構成することによって、利用することができます。

ターゲット・ボードとの接続を指定するには、以下の GDB コマンドを使用します。

`target mips port`

ボード上でプログラムを実行するには、引数にユーザ・プログラムの名前を指定して `gdb`を起動します。ボードに接続するには、‘target mips port’コマンドを使用します。*port* は、ボードに接続されているシリアル・ポートの名前です。プログラムがまだボードにダウンロードされていないのであれば、`load`コマンドを使ってダウンロードすることができます。その後、通常利用できるすべての GDB コマンドを使うことができます。



例えば以下の手順では、デバッガを使うことによって、シリアル・ポートを経由してターゲット・ボードに接続した後に、*prog* と呼ばれるプログラムをロードして実行しています。

```
host$ gdb prog
GDB is free software and ...
(gdb) target mips /dev/ttyb
(gdb) load prog
(gdb) run
```

`target mips hostname:portnumber`

GDB のホスト構成によっては、'*hostname:portnumber*' という構文を使うことで、シリアル・ポートの代わりに（例えば、端末多重化装置によって管理されているシリアル回線への）TCP 接続を指定することができます。

`target pmon port`

`target ddb port`

`target lsi port`

GDB は、MIPS ターゲットに対して、以下の特別なコマンドもサポートしています。

`set processor args`

`show processor`

プロセッサの種類に固有のレジスタにアクセスしたい場合には、`set processor` コマンドを使って MIPS プロセッサの種類を設定します。例えば、`set processor r3041` は、3041 チップで有効な CPO レジスタを使うよう、GDB に対して通知します。GDB が使っている MIPS プロセッサの種類を知るには、`show processor` コマンドを使います。GDB が使っているレジスタを知るには、`info reg` コマンドを使います。

`set mipsfpu double`

`set mipsfpu single`

`set mipsfpu none`

`show mipsfpu`

MIPS 浮動小数点コプロセッサをサポートしないターゲット・ボードを使う場合は、'`set mipsfpu none`' コマンドを使う必要があります（このようなことが必要な場合には、初期化ファイルの中にそのコマンドを入れてしまってもよいでしょう）。これによって、浮動小数値を返す関数の戻り値を見つける方法を GDB に知らせます。またこれにより、ボード上で関数を呼び出すときに、GDB は浮動小数点レジスタの内容を退避する必要がなくなります。R4650 プロセッサ上にあるような、単精度浮動小数だけをサポートする浮動小数点コプロセッサを使っている場合には、'`set mipsfpu single`' コマンドを使います。デフォルトの倍精度浮動小数点コプロセッサは、'`set mipsfpu double`' によって選択することができます。

以前のバージョンでは、有効な選択肢は、倍精度浮動小数コプロセッサを使う設定と浮動小数点コプロセッサを使わない設定だけでした。したがって、'`set mipsfpu on`' で倍精度浮動小数コプロセッサが選択され、'`set mipsfpu off`' で浮動小数点コプロセッサを使わないという設定が選択されていました。

他の場合と同様、`mipsfpu` 変数に関する設定は、'`show mipsfpu`' によって問い合わせることができます。

```
set remotedebug n
```

```
show remotedebug
```

remotedebug変数を設定することによって、ボードとの通信に関するいくつかのデバッグ用の情報を見ることができます。‘set remotedebug 1’によって値 1を設定すると、すべてのパケットが表示されます。値を 2に設定すると、すべての文字が表示されます。‘show remotedebug’コマンドによって、いつでも現在の設定値を調べることができます。

```
set timeout seconds
```

```
set retransmit-timeout seconds
```

```
show timeout
```

```
show retransmit-timeout
```

MIPS リモート・プロトコルにおけるパケット待ちの状態でのタイムアウト時間を、set timeout *seconds*コマンドで制御することができます。デフォルトは 5 秒です。同様に、パケットに対する確認 (ACK) を待っている状態でのタイムアウト時間を、set retransmit-timeout *seconds*コマンドで制御することができます。デフォルトは 3 秒です。それぞれの値を show timeoutと show retransmit-timeoutで調べることができます (どちらのコマンドも、GDBが‘--target=mips-idx-ecoff’用に構成されている場合のみ使用可能です)。

set timeoutで設定されたタイムアウト時間は、ユーザ・プログラムが停止するのをGDBが待っている間は適用されません。この場合には、GDBは永遠に待ち続けます。これは、停止するまでにプログラムがどの程度長く実行を継続するのかわかる方法がないからです。

### 13.4.10 シミュレートされた CPU ターゲット

構成によっては、ユーザ・プログラムをデバッグする際にハードウェア CPU の代わりに使うことのできる CPU シミュレータが、GDB の中に組み込まれています。現在のところ、ARM、D10V、D30V、FR30、H8/300、H8/500、i960、M32R、MIPS、MN10200、MN10300、PowerPC、SH、Sparc、V850、W65、Z8000 用のシミュレータが利用できます。

Z8000 系については、‘target sim’によって、Z8002 (Z8000 アーキテクチャの、セグメントを持たない変種) または Z8001 (セグメントを持つ変種) をシミュレートします。シミュレータは、オブジェクト・コードを調べることで、どちらのアーキテクチャが適切であるかを認識します。

```
target sim args
```

シミュレートされた CPU 上でプログラムをデバッグします。シミュレータがセットアップ・オプションをサポートしている場合は、それを *args* の部分に指定します。

このターゲットを指定した後は、ホスト・コンピュータ上のプログラムをデバッグするのと同様の方法で、シミュレートされた CPU 用のプログラムをデバッグすることができます。新しいプログラムのイメージをロードするには file コマンドを使い、ユーザ・プログラムを実行するには run コマンドを使う、という具合です。

Z8000 シミュレータでは、通常のマシン・レジスタ (info reg を参照) がすべて利用可能であるだけでなく、特別な名前を持つレジスタとして、3 つの追加情報が提供されています。

**cycles**      シミュレータ内のクロック・ティックをカウントします。

**insts**        シミュレータ内で実行された命令をカウントします。

**time**         1/60 秒を単位とする実行時間を示します。

これらの変数は、GDBの式の中で普通に参照することができます。例えば、`'b fputc if $cycles>5000'` は、シミュレートされたクロック・ティックが最低 5,000 回発生した後に停止するような、条件付きブレイクポイントを設定します。



## 14 GDB の制御

`set` コマンドによって GDB の操作方法を変更することができます。GDB によるデータの表示方法を変更するコマンドについては、see Section 8.7 [Print settings], page 65。この章では、その他の設定について説明します。

### 14.1 プロンプト

GDB は、プロンプトと呼ばれる文字列を表示することで、コマンドを受け付ける用意ができたことを示します。通常、この文字列は `(gdb)` です。`set prompt` コマンドによって、プロンプトの文字列を変更することができます。例えば、GDB を使って GDB 自体をデバッグしているときには、どちらか一方の GDB セッションのプロンプトを変更して、どちらの GDB とやりとりしているのか区別できるようにすると便利です。

注：以前のバージョンとは異なり、現在の `set prompt` は、ユーザが設定したプロンプトの後ろに空白を追加しません。ユーザは、空白で終わるプロンプト、空白で終わらないプロンプトのいずれでも設定することができます。

```
set prompt newprompt
```

今後は `newprompt` をプロンプトとして使用するよう、GDB に指示します

```
show prompt
```

`'Gdb's prompt is: your-prompt'` という形式の 1 行を表示します

### 14.2 コマンド編集

GDB は入力コマンドを `readline` インターフェイスによって読み込みます。この GNU ライブラリを使うことで、ユーザに対してコマンドライン・インターフェイスを提供するプログラムは、統一された振る舞いをするようになります。これを使うことの利点としては、GNU Emacs スタイルまたは `vi` スタイルによるコマンドのインライン編集、`csh` スタイルのヒストリ代替、複数のデバッグ・セッションにまたがるコマンド・ヒストリの保存と呼び出しができるようになることが挙げられます。

`set` コマンドによって、GDB におけるコマンドライン編集の振る舞いを制御することができます。

```
set editing
```

```
set editing on
```

コマンドライン編集を使用可能にします (コマンドライン編集は、デフォルトの状態で使用可能です)。

```
set editing off
```

コマンドライン編集を使用不可にします。

```
show editing
```

コマンドライン編集が使用可能かどうかを示します。

### 14.3 コマンド・ヒストリ

デバッグ・セッション中にユーザが入力したコマンドを GDB に記録させることができるため、ユーザは実際に何が実行されたかを確実に知ることができます。以下のコマンドを使って、GDB のコマンド・ヒストリ機能を管理します。

`set history filename fname`

GDB コマンド・ヒストリ・ファイルの名前を *fname* に設定します。GDB は、最初にこのファイルからコマンド・ヒストリ・リストの初期値を読み込み、終了時には、このファイルにセッション中のコマンド・ヒストリを書き込みます。コマンド・ヒストリ・リストには、ヒストリ展開機能、あるいは、後に列挙するヒストリ・コマンド編集文字によってアクセスすることができます。このファイル名は、デフォルトでは環境変数 GDBHISTFILE の値になりますが、この変数が設定されていない場合には `./.gdb_history` になります。

`set history save`

`set history save on`

コマンド・ヒストリをファイルの中に記録します。ファイルの名前は `set history filename` コマンドで指定可能です。デフォルトでは、このオプションは使用不可の状態になっています。

`set history save off`

コマンド・ヒストリをファイルの中に記録するのを停止します。

`set history size size`

GDB がヒストリ・リストの中に記録するコマンドの数を設定します。デフォルトでは、この値は環境変数 HISTSIZE の値に設定されますが、この変数が設定されていない場合は 256 になります。

ヒストリ展開機能により、文字 `!` には特別な意味が割り当てられます。

`!` は、C 言語における論理 NOT の演算子でもあるので、ヒストリ展開機能はデフォルトでは off になっています。`set history expansion on` コマンドによってヒストリ展開を利用できるようにした場合には、( `!` を式の中で論理 NOT として使うのであれば ) `!` の後ろに空白かタブを入れることによって、それが展開されないようにする必要のある場合があります。ヒストリ展開が有効になっている場合でも、`readline` のヒストリ機能は、`!=` や `!` ( という文字列を置き換えようとはしません )。

ヒストリ展開を制御するコマンドには、以下のようなものがあります。

`set history expansion on`

`set history expansion`

ヒストリ展開を使用可能にします。ヒストリ展開はデフォルトでは使用不可です。

`set history expansion off`

ヒストリ展開を使用不可にします。

`readline` のコードには、ヒストリ編集機能やヒストリ展開機能に関する、より完全なドキュメントが付属しています。GNU Emacs や `vi` のことをよく知らない人は、このドキュメントを読むとよいでしょう。

```
show history
show history filename
show history save
show history size
show history expansion
```

これらのコマンドは、GDB のヒストリ・パラメータの状態を表示します。単に `show history` を実行すると、4 つのパラメータの状態がすべて表示されます。

```
show commands
```

コマンド・ヒストリ中の最後の 10 個のコマンドを表示します。

```
show commands n
```

コマンド番号 *n* のコマンドを中心に、その前後の 10 個のコマンドを表示します。

```
show commands +
```

最後に表示されたコマンドに続く 10 個のコマンドを表示します。

## 14.4 画面サイズ

GDB のコマンドは、大量の情報を画面上に出力することがあります。大量の情報をすべて読むのを支援するために、GDB は 1 ページ分の情報を出力するたびに、出力を停止してユーザからの入力を求めます。出力を継続したい場合は `(RET)` キーを押し、残りの出力を破棄したい場合は *q* を入力します。また、画面幅の設定によって、どこで行を折り返すかが決まります。GDB は、単純に次の行に折り返すのではなく、出力の内容に応じて読みやすいところで折り返すよう試みます。

通常 GDB は、`termcap` データベースと `TERM` 環境変数の値、さらに、`stty rows`、`stty cols` の設定から、画面の大きさを知っています。この結果が正しくない場合、`set height` コマンドと `set width` コマンドで画面の大きさの設定を変更することができます。

```
set height lpp
show height
set width cpl
show width
```

これらの `set` コマンドは、画面の高さを *lpp* 行に、幅を *cpl* 桁に指定します。関連する `show` コマンドが、現在の設定を表示します。

ゼロ行の高さを指定すると、GDB は出力がどんなに長くても、出力途中で一時停止することをしません。これは、出力先がファイルやエディタのバッファである場合に便利です。

同様に、`'set width 0'` を指定することによって、GDB に行の折り返しを行わせないようにすることもできます。

## 14.5 数値

GDB に対して 8 進、10 進、16 進の数値を慣例にしたがって入力することはいつでも可能です。8 進数は `'0'` で始まります。10 進数は `'.'` で終わります。16 進数は `'0x'` で始まります。このどれにも該当しないものは、デフォルトで 10 進数として入力されます。同様に、数値を表示するときも、特定のフォーマットが指定されていなければ、デフォルトで 10 進数として表示されます。`set radix` コマンドによって、入力、出力の両方のデフォルトを変更することができます。

`set input-radix base`

数値入力デフォルトの基数を設定します。サポートされる選択肢は 10 進数の 8、10、16 です。*base* 自身はあいまいにならないように指定するか、あるいは、現在のデフォルトの基数を使用して指定します。例えば、

```
set radix 012
set radix 10.
set radix 0xa
```

は基数を 10 進数に設定します。一方、`'set radix 10'` は、現在の基数を（それがどれであれ）変更しません。

`set output-radix base`

数値の表示に使うデフォルトの基数を設定します。サポートされる *base* の選択肢は 10 進数の 8、10、16 です。*base* 自身はあいまいにならないように指定するか、あるいは、現在のデフォルトの基数を使用して指定します。

`show input-radix`

数値の入力に現在使われているデフォルトの基数を表示します。

`show output-radix`

数値の表示に現在使われているデフォルトの基数を表示します。

## 14.6 オプションの警告およびメッセージ

デフォルトでは、GDB は内部の動作に関する情報を表示しません。性能の遅いマシンで実行している場合には、`set verbose` コマンドを使うとよいでしょう。これによって、GDB は、長い内部処理を実行するときにメッセージを出力することで、クラッシュと勘違いされないようにします。

現在のところ、`set verbose` コマンドによって制御されるメッセージは、ソース・ファイルのシンボル・テーブルを読み込み中であることを知らせるメッセージです。Section 12.1 [Commands to specify files], page 99 の `symbol-file` を参照してください。

`set verbose on`

GDB が特定の情報メッセージを出力するようにします。

`set verbose off`

GDB が特定の情報メッセージを出力しないようにします。

`show verbose`

`set verbose` が `on`、`off` のどちらの状態であることを表示します。

デフォルトでは、オブジェクト・ファイルのシンボル・テーブルに問題を検出しても、GDB はメッセージを出力しません。しかし、コンパイラをデバッグしているようなときには、このような情報があると便利かもしれません（see Section 12.2 [Errors reading symbol files], page 102 ）。

`set complaints limit`

異常な型のシンボルを検出するたびに GDB が出力するメッセージの総数を *limit* 個とします。*limit* 個のメッセージを表示すると、その後は問題を検出してもメッセージを表示しないようになります。メッセージを 1 つも出力させないようにするには、*limit* にゼロを指定してください。メッセージの出力が抑止されないようにするには、*limit* に大きな値を設定してください。



`show complaints`

GDB が何個までシンボル異常に関するメッセージを出力できるよう設定されているかを表示します。

デフォルトでは、GDB は慎重に動作し、コマンドを本当に実行するのか確認するために、ときには馬鹿げているとさえ思えるような質問を多く尋ねてきます。例えば、既に実行中のプログラムを実行しようとする、次のように質問してきます。

```
(gdb) run
```

```
The program being debugged has been started already.
```

```
Start it from the beginning? (y or n)
```

ユーザが、実行したコマンドの結果を何がなんでも見てみたいのであれば、この「機能」を抑止することができます。

`set confirm off`

確認要求を行わないようにします。

`set confirm on`

確認要求を行うようにします (デフォルト)。

`show confirm`

確認要求の現在の設定を表示します。



## 15 一連のコマンドのグループ化

ブレイクポイント・コマンド ( see Section 5.1.7 [Breakpoint command lists], page 39 ) とは別に、一連のコマンドを一括して実行するために保存する 2 つの方法を、GDB は提供しています。ユーザ定義コマンドとコマンド・ファイルがそれです。

### 15.1 ユーザ定義コマンド

ユーザ定義コマンドとは、一連の GDB コマンドに単一コマンドとしての名前を新たに割り当てたものです。これは、define コマンドによって行われます。ユーザ・コマンドは、空白で区切られた引数を最高で 10 個まで受け取ることができます。引数は、ユーザ・コマンドの中で、\$arg0...\$arg9 としてアクセスすることができます。簡単な例を以下に示します。

```
define adder
  print $arg0 + $arg1 + $arg2
```

このコマンドを実行するには、以下のようにします。

```
adder 1 2 3
```

上の例では、adder というコマンドを定義しています。このコマンドは、3 つの引数の合計を表示します。引数は文字列で代用されますので、変数を参照することもできますし、複雑な式を使うこともできます。また、下位関数の呼び出しを行うこともできます。

`define commandname`

`commandname` という名前のコマンドを定義します。同じ名前のコマンドが既に存在する場合は、再定義の確認を求められます。

コマンドの定義は、define コマンドに続いて与えられる、他の GDB コマンド行から構成されます。これらのコマンドの末尾は、end を含む行によって示されます。

`if`      引数として、評価の対象となる式を 1 つだけ取ります。その後一連のコマンドが続きますが、これらのコマンドは、式の評価結果が真 (ゼロ以外の値) である場合にだけ実行されます。さらに、else 行が続くことがあり、この場合は、else 行の後に、式の評価結果が偽であった場合にだけ実行される一連のコマンドが続きます。末尾は、end を含む行によって示されます。

`while`    構文は if と似ています。引数として、評価の対象となる式を 1 つだけ取ります。その後には、実行されるべきコマンドが 1 行に 1 つずつ続き、最後に end がなければなりません。コマンドは、式の評価結果が真である限り、繰り返し実行されます。

`document commandname`

ユーザ定義コマンド `commandname` のドキュメントを記述します。このドキュメントは help コマンドによってアクセスできます。コマンド `commandname` は既に定義済みでなければなりません。このコマンドは、define コマンドが一連のコマンド定義を読み込むのと同様に、end で終わる一連のドキュメントを読み込みます。document コマンドの実行が完了すると、コマンド `commandname` に対して help コマンドを実行すると、ユーザの記述したドキュメントが表示されます。

document コマンドを再度実行することによって、コマンドのドキュメントを変更することができます。define コマンドによってコマンドを再定義しても、ドキュメントは変更されません。

```
help user-defined
```

すべてのユーザ定義コマンドを一覧表示します。個々のコマンドにドキュメントがあれば、その1行目が表示されます。

```
show user
```

```
show user commandname
```

*commandname* で指定されるコマンドを定義するのに使われた GDB コマンドを表示します(ドキュメントは表示されません)。 *commandname* を指定しないと、すべてのユーザ定義コマンドの定義が表示されます。

ユーザ定義コマンドが実行されるときに、定義内のコマンドは表示されません。定義内のコマンドがどれか1つでもエラーになると、ユーザ定義コマンドの実行が停止されます。

対話的に使われている場合には確認を求めてくるようなコマンドも、ユーザ定義コマンドの内部で使われている場合には確認を求めることなく処理を継続します。通常は実行中の処理に関してメッセージを表示する GDB コマンドの多くが、ユーザ定義コマンドの中から呼び出されている場合にはメッセージを表示しません。

## 15.2 ユーザ定義コマンド・フック

特別な種類のユーザ定義コマンドである、フックを定義することができます。‘hook-foo’というユーザ定義コマンドが存在すると、‘foo’というコマンドを実行するときにはいつも、‘foo’コマンドが実行される前に(引数のない)‘hook-foo’が実行されます。

また、仮想コマンドである‘stop’が存在します。(‘hook-stop’を)定義すると、ユーザ・プログラムの実行が停止するたびに、その定義内のコマンドが実行されます。実行タイミングは、ブレイクポイント・コマンドの実行、自動表示対象の表示、および、スタック・フレームの表示の直前です。

例えば、シングル・ステップ実行をしている際には SIGALRMシグナルを無視し、通常の実行時には通常どおり処理したい場合には、以下のように定義します。

```
define hook-stop
handle SIGALRM nopass
end
```

```
define hook-run
handle SIGALRM pass
end
```

```
define hook-continue
handle SIGLARM pass
end
```

GDB のコマンドのうち、その名前が1つの単語から成るものには、フックを定義することができます。ただし、コマンド・エイリアスにフックを定義することはできません。フックは、コマンドの基本名に対して定義しなければなりません。例えば、btではなく backtraceを使います。フックの実行中にエラーが発生すると、GDB コマンドは停止します。(ユーザが実際に入力したコマンドが実行する機会を与えられる前に) GDB はプロンプトを表示します。

既知のコマンドのいずれにも対応しないフックを定義しようとすると、defineコマンドは警告メッセージを表示します。

### 15.3 コマンド・ファイル

GDB のコマンド・ファイルとは、各行が GDB コマンドとなっているファイルのことです。(行の先頭が#の) コメントも含めることができます。コマンド・ファイル内の空行は何も実行しません。それは、端末上での実行の場合とは異なり、最後に実行されたコマンドの繰り返しを意味しません。

GDB を起動すると、自動的に初期化ファイルからコマンドを読み込んで実行します。これは、Unix 上では `.gdbinit` という名前のファイルであり、DOS/Windows 上では `gdb.ini` という名前のファイルです。GDB は、ユーザのホーム・ディレクトリに初期化ファイルがあればまずそれを読み込み、続いてコマンドライン・オプションとオペランドを処理した後、カレントな作業ディレクトリに初期化ファイルがあればそれを読み込みます。このように動くのは、ユーザのホーム・ディレクトリに初期化ファイルを置くことで、コマンドライン上のオプションやオペランドの処理に影響を与える ( `set complaints` のような ) オプションを設定することができるようにするためです。 `-nx` オプションを使用すると、初期化ファイルは実行されません。 see Section 2.1.2 [Choosing modes], page 11.

GDB のいくつかの構成では、初期化ファイルは異なる名前で知られています (このような環境では、特別な形式の GDB が他の形式の GDB と共存する必要がある、そのために特別なバージョンの GDB の初期化ファイルには異なる名前が付けられます)。特別な名前の初期化ファイルを持つ環境には、以下のようなものがあります。

- VxWorks ( Wind River Systems 社のリアルタイム OS ): `.vxgdbinit`
- OS68K ( Enea Data Systems 社のリアルタイム OS ): `.os68gdbinit`
- ES-1800 ( Ericsson Telecom 社の AB M68000 エミュレータ ): `.esgdbinit`

また、`source` コマンドによって、コマンド・ファイルの実行を要求することもできます。

`source filename`

コマンド・ファイル `filename` を実行します。

コマンド・ファイルの各行は順番に実行されます。コマンドの実行時に、そのコマンドは表示されません。どれか 1 つでもコマンドがエラーになると、コマンド・ファイルの実行は停止されます。

対話的に使われている場合には確認を求めてくるようなコマンドも、コマンド・ファイル内で使われている場合は確認を求めることなく処理を継続します。通常は実行中の処理についてメッセージを表示する GDB コマンドの多くが、コマンド・ファイルの中から呼び出されている場合にはメッセージを表示しません。

### 15.4 制御された出力を得るためのコマンド

コマンド・ファイルやユーザ定義コマンドの実行中には、通常の GDB の出力は抑止されます。唯一出力されるのは、定義内のコマンドが明示的に表示するメッセージだけです。ここでは、ユーザが希望するとおりの出力を生成するのに役に立つ、3 つのコマンドについて説明します。

`echo text` `text` を表示します。通常は表示されない文字も、C のエスケープ・シーケンスを使うことで `text` の中に含めることができます。例えば、改行コードを表示するには `'\n'` を使います。明示的に指定しない限り、改行コードは表示されません。標準的な C のエスケープ・シーケンスに加えて、バックスラッシュの後ろに空白を置くことで、空白が表わされます。これは、先頭や末尾に空白のある文字列を表示するのに便利です。というのは、こうしないと、引数の先頭や末尾の空白は削除されるからです。 `' and foo = '` を表示するには、 `'echo \ and foo = \'` を実行してください。

C と同様、`text` の末尾にバックスラッシュを置くことで、コマンドを次の行以降に継続することができます。例えば、

```
echo This is some text\n\
which is continued\n\
onto several lines.\n
```

は

```
echo This is some text\n
echo which is continued\n
echo onto several lines.\n
```

と同じ出力をもたらします。

`output expression`

*expression* の値を表示し、それ以外には何も表示しません。改行コードも、`'$nn = '`も表示されません。*expression* の値は値ヒストリには入りません。式の詳細については、See Section 8.1 [Expressions], page 59。

`output/fmt expression`

*expression* の値を、*fmt* で指定されるフォーマットで表示します。print コマンドと同じフォーマットを指定することができます。詳細については、See Section 8.4 [Output formats], page 62。

`printf string, expressions...`

*string* で指定された文字列にしたがって *expressions* の値を表示します。複数の *expressions* はカンマで区切られ、数値かポインタのいずれかを指定できます。これらの値は、ユーザ・プログラムから C のサブルーチン

```
printf (string, expressions...);
```

を実行した場合と同様に、*string* の指定にしたがって表示されます。

例えば、次のようにして 2 つの値を 16 進数で表示することができます。

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

フォーマットを指定する文字列の中で使えるバックスラッシュ・エスケープ・シーケンスは、バックスラッシュとそれに続く単一文字から構成される簡単なものだけです。

## 16 GNU Emacs 中での GDB の使用

GDB でデバッグ中のプログラムのソース・ファイルを GNU Emacs を使って参照（および編集）するための、特別なインターフェイスが提供されています。

このインターフェイスを使うには、Emacs 中で *M-x gdb* コマンドを使います。デバッグしたい実行ファイルを引数として指定してください。このコマンドは、GDB を Emacs のサブプロセスとして起動し、新しく作成した Emacs バッファを通じて入出力を行います。

Emacs 中での GDB の使い方は、通常の GDB の使い方とほぼ同様ですが、2 つ相違点があります。

- 「端末」へのすべての入出力は Emacs バッファへ送られる。

これは、GDB コマンドとその出力、および、デバッグ対象のユーザ・プログラムによる入出力の両方に適用されます。

以前に実行したコマンド・テキストをコピーして再入力することができるので便利です。出力された部分に関しても同様のことができます。

Emacs の Shell モードで利用可能なすべての機能を、ユーザ・プログラムとのやりとりで使うことができます。特に、通常どおりにシグナルを送信することができます。例えば、*C-c C-c* で割り込みシグナルを、*C-c C-z* でストップ・シグナルを発生させることができます。

- GDB は Emacs を使ってソース・コードを表示する。

GDB がスタック・フレームを表示するときにはいつでも、Emacs がそのフレームのソース・ファイルを自動的に見つけて、カレント行の左側の余白に矢印（`=>`）を表示します。Emacs はソース・コードを別バッファに表示し、スクリーンを 2 つに分けて、GDB セッションとソースをともに表示します。

GDB の `list` コマンドや `search` コマンドを明示的に使えば、通常どおりの出力を生成することもできますが、これらを Emacs から使う理由はおそらくないでしょう。

注意：ユーザ・プログラムの存在するディレクトリがユーザのカレント・ディレクトリでない場合、ソース・ファイルの存在場所について Emacs は簡単に混乱に陥ります。このような場合、ソースを表示するための追加のディスプレイ・バッファは表示されません。GDB は、ユーザの環境変数 `PATH` のリストの中にあるディレクトリを探索してプログラムを見つけ出しますので、GDB の入出力セッションは通常どおり進行します。しかし Emacs は、このような状況においてソース・ファイルを見つけ出すのに十分な情報を GDB から受け取っていません。この問題を回避するには、ユーザ・プログラムの存在するディレクトリから GDB モードを開始するか、*M-x gdb* の引数の入力を求められたときに、絶対パスでファイル名を指定します。

Emacs の既存の GDB バッファから、デバッグ対象をどこかほかの場所にあるプログラムに変更する目的で GDB の `file` コマンドを使うと、同様の混乱の発生することがあります。

デフォルトでは、*M-x gdb* は `'gdb'` という名前のプログラムを呼び出します。別の名前で GDB を呼び出す必要がある場合（例えば、異なる構成の GDB を別の名前で持っているような場合）は、Emacs の `gdb-command-name` という変数を設定します。例えば

```
(setq gdb-command-name "mygdb")
```

（を `ESC ESC` に続けて入力するか、あるいは、`*scratch*` バッファまたは `'.emacs'` ファイルに入力することで）Emacs は `gdb` の代わりに `'mygdb'` という名前のプログラムを呼び出します。

GDB の I/O バッファでは、標準的な Shell モードのコマンドに加えて、以下のような特別な Emacs コマンドを使うことができます。

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>C-h m</i>         | Emacs の GDB モードの機能に関する説明を表示します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <i>M-s</i>           | GDB の <code>step</code> コマンドのように、ソース行を 1 行実行します。さらに、カレントなファイルと其中における位置を示すために、表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <i>M-n</i>           | GDB の <code>next</code> コマンドのように、関数呼び出しをすべてスキップして、現在の関数内の次のソース行まで実行を進めます。さらに、カレントなファイルと其中における位置を示すために、表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <i>M-i</i>           | GDB の <code>stepi</code> コマンドのように、1 命令を実行します。必要に応じて表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <i>M-x gdb-nexti</i> | GDB の <code>nexti</code> コマンドを使って、次の命令まで実行します。必要に応じて表示ウィンドウを更新します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>C-c C-f</i>       | GDB の <code>finish</code> コマンドのように、選択されたスタック・フレームを終了するまで実行を継続します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <i>M-c</i>           | GDB の <code>continue</code> コマンドのように、ユーザ・プログラムの実行を継続します。<br>注意: Emacs v19 では、このコマンドは <i>C-c C-p</i> です。                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>M-u</i>           | GDB の <code>up</code> コマンドのように、数値引数によって示される数だけ上位のフレームに移動します ( see section “Numeric Arguments” in <i>The GNU Emacs Manual</i> <sup>1</sup> )。<br>注意: Emacs v19 では、このコマンドは <i>C-c C-u</i> です。                                                                                                                                                                                                                                                                                                                                                                             |
| <i>M-d</i>           | GDB の <code>down</code> コマンドのように、数値引数によって示される数だけ下位のフレームに移動します。<br>注意: Emacs v19 では、このコマンドは <i>C-c C-d</i> です。                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <i>C-x &amp;</i>     | カーソルの位置にある数値を読み取り、GDB の I/O バッファの末尾に挿入します。例えば、以前に表示されたアドレスの前後のコードを逆アセンブルしたいとしましょう。この場合、まず <code>disassemble</code> と入力し、次に表示されたアドレスのところにカーソルを移動し、 <code>disassemble</code> への引数を <i>C-x &amp;</i> で読み取ります。<br><br>gdb-print-command リストの要素を定義することによって、これをさらにカスタマイズすることができます。これが定義されていると、 <i>C-x &amp;</i> で入手した数値が挿入される前に、それをフォーマットしたり、処理したりすることができるようになります。 <i>C-x &amp;</i> に数値引数を指定すると、特別なフォーマット処理を必要としているという意味になり、その数値がリストの要素を取得するためのインデックスとなります。リストの要素が文字列の場合は、挿入される数値は Emacs の <code>format</code> 関数によってフォーマットされます。リストの要素が文字列以外の場合は、その数値が、対応するリスト要素への引数として渡されます。 |

どのソース・ファイルが表示されている場合でも、Emacs の *C-x SPC* ( `gdb-break` ) コマンドは、ポイントの置かれているソース行にブレイクポイントを設定するよう GDB に対して指示します。

ソースを表示中のバッファを誤って削除してしまった場合に、それを再表示させる簡単な方法は、GDB バッファの中で `f` コマンドを入力して、フレーム表示を要求することです。Emacs 配下では、カレント・フレームのコンテキストを表示するために必要であれば、ソース・バッファが再作成されます。

<sup>1</sup> 訳注: *GNU Emacs 19 マニュアル* ( 星雲社 ) の「ニューメリック引数」、*GNU Emacs マニュアル* ( 共立出版 ) の「数引数」に、日本語訳があります。



Emacs によって表示されるソース・ファイルは、通常どおりの方法でソース・ファイルにアクセスする、普通の Emacs バッファによって表示されます。そうしたいのであれば、これらのバッファの中でファイルの編集を行うこともできますが、GDB と Emacs の間で行番号に関する情報が交換されていることを頭に入れておいてください。テキストに行を挿入したり、削除したりすると、GDB の認識しているソース行番号は、実際のコードと正しく対応しなくなってしまいます。



## 17 GDB のバグ報告

ユーザからのバグ報告は、GDB の信頼性を向上させるのに重要な役割を果たしています。

バグを報告することで、その問題の解決につながり、結果として報告者自ら利益を得ることができるかもしれませんが、もちろん、何の解決にもつながらないこともあります。しかし、いずれにしても、バグ報告の主要な意義は、次のバージョンの GDB をより良いものにすることで、コミュニティ全体の役に立つという点にあります。バグ報告は、GDB の保守作業へのユーザからの貢献です。

バグ報告がその目的とするところを首尾よく達成できるようにするためには、バグを修正することを可能にするような情報が提供されなければなりません。

### 17.1 本当にバグを見つけたのかどうかを知る方法

発見した現象がバグかどうかよく分からない場合には、以下のガイドラインを参照してください。

- 入力された情報が何であれ、デバッガが致命的なシグナルを受信するのであれば、それは GDB のバグです。信頼性のあるデバッガは決してクラッシュなどしません。
- 正当な入力に対して GDB がエラー・メッセージを出力するのであれば、それはバグです。(クロス・デバッグを行っている場合には、ターゲットへの接続に問題がある可能性もあるということに注意してください。)
- 不正な入力に対して GDB がエラー・メッセージを出力しないのであれば、それはバグです。ただし、ユーザにとって「不正な入力」に思えるものが、実は「拡張機能」であったり「古くから使われている用法のサポート」であったりすることもあります。
- デバッグ・ツールに関する経験が豊富なユーザからの GDB の改善提案は、どのような場合でも歓迎です。<sup>1</sup>

### 17.2 バグの報告方法

いくつかの企業や個人が GNU のソフトウェアをサポートしています。こうしたサポート組織から GDB を入手したのであれば、まずその組織に連絡することをお勧めします。

サポートを提供している多くの企業、個人の連絡先情報が、GNU Emacs ディストリビューションの `etc/SERVICE` ファイルに記載されています。

どのような場合でも、GDB のバグ報告を ( 英語で ) 以下のアドレスに送ることをお勧めします。<sup>2</sup>

`bug-gdb@prep.ai.mit.edu`

`'info-gdb'`、`'help-gdb'`、および、いかなるニュースグループにもバグ報告を送ることはしないでください。GDB ユーザのほとんどは、バグ報告を受け取りたいと考えてはいません。バグ報告を受け取りたいと思っている人は、`'bug-gdb'` の配信を受けるようにしているはずです。

メーリング・リスト `'bug-gdb'` には、リピータとして機能する `'gnu.gdb.bug'` というニュースグループがあります。このメーリング・リストとニュースグループは、全く同一のメッセージを配信しています。メーリング・リストではなくニュースグループにバグ報告を流そうと考える人がよくいます。これはうまく機能するようには見えますが、1 つ重大な問題があります。ニュースグループへの投稿では、送信者へのメール・パスが分からないことがよくあります。したがって、もっと多くの情報

<sup>1</sup> 訳注: この日本語の翻訳マニュアルへの改善提案は、`ki@home.email.ne.jp` に送ってください。

<sup>2</sup> 訳注: この日本語の翻訳マニュアルのバグは、日本語 ( か英語 ) で、`ki@home.email.ne.jp` に報告してください。

が必要になったときに、バグの報告者と連絡を取ることができない可能性があります。こういうことがあるので、メーリング・リストへのバグ報告の方が望ましいのです。

最後の手段として、バグ報告を（英語で）紙に書いて下記に郵送するという方法があります。

GNU Debugger Bugs  
Free Software Foundation Inc.  
59 Temple Place - Suite 330  
Boston, MA 02111-1307  
USA

役に立つバグ報告を行うための最も根本的な原則は、すべての事実を報告することです。ある事実を書くべきか省くべきかよく分からない場合は、書くようにしてください。

事実が省略されてしまうことがよくありますが、これはバグ報告者が、自分には問題の原因は既に分かっていると考え、いくつかの細かい点は関係がないと仮定してしまうからです。したがって、例の中で使った変数の名前などは重要ではないと、報告者は考えます。おそらくそうかもしれません。しかし、完全にそうであるとも言い切れません。メモリの参照がデタラメな場所を指しているというバグで、それがたまたまメモリ上においてその名前が置かれている箇所から値を取り出しているということがあるかもしれません。名前が異なれば、その内容は、バグが存在するにもかかわらずデバッグが正しく動作してしまうような値になるかもしれません。このようなことがないよう、特定の完全な実例を提供してください。バグの報告者にとっては、このようにするのが最も簡単なはずであり、かつ、それが最も役に立つのです。

バグ報告の目的は、そのバグを修正することができるようにすることにある、という点を頭に入れておいてください。そのバグが、以前に報告されたものと同じであるという可能性もありますが、バグ報告が完全なもので、必要な情報がすべて含まれたものでなければ、バグの報告者にも私たちにもそのことを知ることはできません。

ときどき、2、3の大雑把な事実だけを記述して、「何か思い当たることはありますか?」と聞いてくる人がいます。このようなバグ報告は役に立ちません。このような報告には、より適切なバグ報告を送るよう報告者に注意する場合を除いて、返事をするを拒否するよう強くお願いします。

バグを修正できるようにするためには、報告者は以下の情報をすべて含めるべきです。

- GDB のバージョン。GDB のバージョンは、引数を指定せずに GDB を起動すると、表示されます。また、いつでも `show version` コマンドで表示させることができます。  
この情報がないと、カレント・バージョンの GDB を使ってバグを探すことに意味があるのかどうかを知ることができません。
- 使っているマシンのタイプ、オペレーティング・システムの名前とバージョン番号。
- GDB をコンパイルするのに使われたコンパイラ（および、そのバージョン）。例えば、`gcc-2.8.1`。
- デバッグ対象のプログラムをコンパイルするのに使われたコンパイラ（および、そのバージョン）。例えば、`gcc-2.8.1`、あるいは、`HP92453-01 A.10.32.03 HP C コンパイラ`。GCC については、`gcc --version`によってこの情報を知ることができます。他のコンパイラについては、そのドキュメントを参照してください。
- バグを見つけたプログラムをコンパイルする際に、コンパイラに渡したコマンド引数。例えば、`'-O'` オプションを使ったか否かなど。何か重要な点を省いてしまうことがないよう、すべての引数を記述してください。`'Makefile'`のコピー（あるいは、`make`からの出力）を添付すれば十分でしょう。  
引数が何であったのかを私たちが推測しようとしても、おそらく誤った推測をしてしまうでしょう。そうすると、バグは再現しないかもしれません。
- バグを再現することのできる、完全な入力スクリプトとすべての必要なソース・ファイル。

- 発見された、正しくないと思われる動作の説明。例えば、「致命的なシグナルを受信する」など。もちろん、GDB が致命的なシグナルを受信するというバグであれば、私たちも間違いなくそれに気がつくでしょう。しかし、出力が正しくないというバグであれば、紛れもない誤りでなければ、私たちはそれに気付かないかもしれません。私たちが間違いをする可能性を排除するようにしてください。

たとえ致命的なシグナルを受信するような問題であっても、報告者はそのことを明示的に報告すべきです。何か奇妙なことが起こっていると仮定しましょう。例えば、報告者が使っている GDB にちぐはぐなところがあるとか、報告者のシステム上にある C ライブラリのバグだった、というような場合です（こういうことは、実際にありました！）。このような場合、報告者の GDB はクラッシュしても、私たちのところではクラッシュしません。クラッシュするはずであると報告されていれば、私たちの GDB がクラッシュしなくても、「私たちのところではバグが発生しない」ということを知ることができます。クラッシュするはずであるという報告がなければ、実際の現象から何も結論を引き出すことができません。

- もし GDB のソースへの修正を提案したいのであれば、コンテキスト付きの差分情報を送ってください。GDB のソースについて何か議論する場合も、行番号に言及するのではなく、コンテキストに言及してください。

私たちが開発中のソースの行番号は、報告者の持っているソースの行番号とは一致しないでしょう。報告者から見たソースの行番号は、私たちにとって役に立つ情報を提供してくれません。

以下に、バグ報告に必要なではない情報をいくつか列挙します。

- バグの包括的な説明。

バグを見つけると、多くの時間をかけて、入力ファイルをどのように変更するとバグが発生しなくなり、どのように変更した場合はバグが発生し続けるかを調べる人がよくいます。

これは多くの場合、時間のかかる作業であり、しかもあまり役に立ちません。というのは、私たちがバグを見つけるのは、デバッガでブレイクポイントを使いながら 1 つの実例を実行させることによってであり、一連の実例からの純粋な演繹によってではないからです。時間を無駄にせず、何かほかのことに使うようお勧めします。

もちろん、一番最初にバグを見つけたときの実例の代わりとなる、もっと単純な実例を見つけることができるのであれば、私たちにとっても便利です。出力におけるエラーはより発見しやすいものですし、デバッガ配下で実行させる方が時間がかかりません。

しかし、単純化は絶対に必要というわけでもありません。こういうことをしたくないのであれば、バグを発見したときのテスト・ケース全体を送って、バグの報告を行ってください。

- バグに対するパッチ。

バグに対するパッチは、それが良いものであれば、役に立ちます。しかし、パッチがあれば十分であるとみなして、テスト・ケースのような必要な情報を送るのを省かないでください。提供されたパッチに問題があり、別の方法で問題を修正することにする場合もありますし、提供されたパッチを全く理解できないということもあるかもしれません。

GDB のような複雑なプログラムでは、コード中のある特定のパスを通るような実例を作成するのは困難なことがあります。報告者が実例を送ってくれなければ、私たちには実例を作成することができず、したがって、バグが修正されたことを検証することができなくなってしまいます。

また、報告者の送ってくれたパッチがどのような問題を修正しようとしているのか私たちに理解できない場合、あるいは、なぜそのパッチが改善になるのか私たちが理解できない場合、そのパッチを組み込むことはしません。テスト・ケースが 1 つでもあれば、そうしたことを理解するのに役立つでしょう。

- バグが何であるか、あるいは、何に依存しているかに関する推測。

このような推測は普通は間違っているものです。私たちですら、デバッガを使って事実を見出すまでは、このような点に関して正しく推測することはできないのです。

## 18 コマンドライン編集

この章では、GNU のコマンドライン編集インターフェイスの基本的な特徴について説明します。

### 18.1 行編集入門

以下のパラグラフでは、キー・ストロークを表わすために使用される表記法について説明します。

**C-k** は、Control-K という意味です。これは、コントロール・キーが押されたままの状態でも **k** が押されたときに生成される文字を表わします。

**M-k** は、Meta-K という意味です。これは、メタ・キー (があるものとして、それ) が押されたままの状態でも **k** が押されたときに生成される文字を表わします。メタ・キーがない場合、最初に **ESC** キーを押す、次に **k** を押すことで、同等のキー・ストロークを生成することができます。どちらの手順も、**k** をメタ化する、といえます。

**M-C-k** は、Meta-Control-K という意味です。これは、**C-k** をメタ化することにより生成される文字を指します。

さらに、いくつかのキーには名前があります。**DEL**、**ESC**、**LFD**、**SPC**、**RET**、**TAB** は、この文章の中でも、初期化ファイルの中でも、各々のキーを表わします (see Section 18.3 [Readline Init File], page 152)。

### 18.2 Readline の操作

対話的なセッションにおいて、長いテキストを 1 行に記述した後で、その行の先頭の単語のスペルが間違っていたことに気が付くことがよくあります。Readline ライブラリは、入力したテキストを操作するための一連のコマンドを提供しており、これによって、その行の大部分を入力し直すことなく、タイプ・ミスしたところだけを修正することができます。これらの編集コマンドを使って、修正が必要なところにカーソルを移動させ、テキストを削除したり、修正テキストを挿入したりします。その行の修正が終われば、単に **RET** を押します。**RET** を押すのに、行末にいる必要はありません。カーソルが行内のどこにあらうと、その行全体が入力として受け付けられます。

#### 18.2.1 Readline の基本

行内に文字を入力するには、単にその文字をタイプします。タイプされた文字はカーソルの位置に表示され、カーソルは 1 桁分右へ移動します。1 文字打ち間違えた場合は、削除文字 (erase character) を使って、後退しながら打ち間違えた文字を削除することができます。

ときには、本当は入力しなかった文字を入力せず、その誤りに気が付くことなく、さらに数文字を入力してしまうことがあります。このような場合には、**C-b** によってカーソルを左に移動し、誤りを訂正することができます。訂正後、**C-f** によってカーソルを右に移動することができます。

行の途中にテキストを追加すると、挿入されたテキストのためのスペースを空けるために、カーソルの右側にある文字が右方向に押しやられることに気がつくでしょう。同様に、カーソル位置にあるテキストを削除すると、文字が削除されたために生じる空白を埋めるために、カーソルの右側にある文字が左方向に引き戻されます。入力行のテキストを編集するための最も基本的な操作の一覧を以下に示します。

**C-b**      1 文字戻ります。

**C-f**      1 文字進みます。

**DEL** カーソルの左にある文字を削除します。

**C-d** カーソル位置にある文字を削除します。

表示可能な文字

行内のカーソル位置にその文字を挿入します。

**C-** 最後の編集コマンドを取り消して元に戻します。行内に文字が無くなるまで取り消しを繰り返すことが可能です。

### 18.2.2 Readline 移動コマンド

上記の一覧は、ユーザが入力行を編集するのに必要な、最も基本的なキー・ストロークを説明したものです。ユーザの利便を考慮して、**C-b**、**C-f**、**C-d**、**DEL**に加えて多くのコマンドが追加されてきました。以下に、行内をより迅速に動きまわるためのコマンドをいくつか示します。

**C-a** 行の先頭に移動します。

**C-e** 行の末尾に移動します。

**M-f** 1 単語分先に進みます。単語は、文字と数字から構成されます。

**M-b** 1 単語分前に戻ります。

**C-l** 画面上の情報を消去し、カレント行が画面の一番上にくるようにして再表示します。

**C-f**が1文字分先に進むのに対して、**M-f**が1単語分先に進む点に注意してください。大まかな慣例として、コントロール・キーを使うと文字単位の操作になり、メタ・キーを使うと単語単位の操作になります。

### 18.2.3 Readline キル (kill) コマンド

テキストをキル (kill) するとは、行からテキストを削除し、その際に、そのテキストを後に引き出して行内に再挿入 (yank) することができるように退避しておくことを指します。あるコマンドの説明に「テキストをキル (kill) する」という記述があれば、後に別の箇所 (あるいは同じ箇所) において、そのテキストを再入手することができると考えて間違いありません。

キル (kill) コマンドを使うと、テキストはキル・リング (kill-ring) に退避されます。キル (kill) コマンドを任意の回数連続して実行すると、キル (kill) されたテキストはすべて連結されて退避されます。したがって、再挿入 (yank) を行くと、そのすべてを入手することができます。キル・リング (kill-ring) は個々の行に固有のものではありません。以前入力した行においてキル (kill) したテキストを、後になって別の行を入力しているときに再挿入 (yank) することができます。

以下に、テキストをキル (kill) するためのコマンドを一覧で示します。

**C-k** カレントなカーソル位置から行末までのテキストをキル (kill) します。

**M-d** カーソル位置から、カーソルの置かれている単語の末尾までをキル (kill) します。カーソルが2つの単語の間にあるときは、次の単語の末尾までをキル (kill) します。

**M-DEL** カーソル位置から、カーソルの置かれている単語の先頭までをキル (kill) します。カーソルが2つの単語の間にあるときは、前の単語の先頭までをキル (kill) します。

**C-w** カーソル位置から、それより前にある最初の空白までをキル (kill) します。単語間の境界が異なるので、これは**M-DEL**とは異なります。



キル ( kill ) されたテキストを引き出して行内へ再挿入 ( yank ) する方法を、以下に示します。再挿入 ( yank ) とは、最後にキルされたテキストを、キル・バッファからコピーすることを意味しています。

- Ⓒ-ⓤ バッファ内のカーソル位置に、最後にキル ( kill ) されたテキストを再挿入 ( yank ) します。
- Ⓖ-ⓤ キル・リング ( kill-ring ) を回転させ、新たに一番上にきたテキストを再挿入 ( yank ) します。このコマンドを実行できるのは、1 つ前に実行したコマンドが Ⓒ-ⓤ または Ⓖ-ⓤ の場合だけです。

#### 18.2.4 Readline の引数

Readline コマンドには数値引数を渡すことができます。数値引数は、繰り返し回数として使われたり、引数の符号として使われたりします。通常は先に進むようなコマンドに負の数を引数として指定すると、前に戻るようになります。例えば、行の先頭までのテキストをキル ( kill ) するには、'M--C-k'としてもよいでしょう。

コマンドに数値引数を渡す通常の方法は、コマンドの前にメタ化された数字を入力することです。入力された最初の「数字」がマイナス記号 ( - ) の場合、引数の符号は負になります。引数を開始するためには、メタ化された数字を 1 つだけ入力すればよく、残りの数字はそのまま入力することができます。そして最後にコマンドを入力します。例えば、Ⓒ-Ⓓ コマンドに引数として 10 を渡すためには、'M-1 0 C-d'と入力します。

#### 18.2.5 ヒストリ中のコマンドの検索

readline は、コマンド・ヒストリの中から、指定された文字列を含む行を検索するコマンドを提供しています。インクリメンタル ( incremental ) と非インクリメンタル ( non-incremental ) の 2 つの検索モードがあります。

インクリメンタル ( incremental ) ・モードでは、ユーザが検索文字列を入力し終わる前から検索が始まります。検索文字列の中の文字が 1 つ入力されるたびに、readline は、それまで入力された文字列にマッチする、ヒストリの中の次のエントリを表示します。インクリメンタル・モードの検索では、検索したいヒストリ・エントリを見つけるのに本当に必要となる文字だけを入力するだけで済みます。インクリメンタル・モードの検索を中止するには、ⒺⒸ文字を使います。Ⓒ-Ⓙでも、検索は中止されます。Ⓒ-Ⓖは、インクリメンタル・モードの検索を終了させて、元の行を表示します。検索が中止されると、検索文字列を含むヒストリ・エントリがカレント行となります。検索文字列にマッチする他のエントリをヒストリ・リストからを見つけるためには、必要に応じてⒸ-ⓈまたはⒸ-Ⓣを入力します。これによって、それまでに入力された検索文字列にマッチする次のエントリをヒストリからを見つけるために、下の方向、または、上の方向に検索が行われます。Readline コマンドにバインドされているキー・シーケンスのうち上記以外のものを入力すると、検索は中止され、そのコマンドが実行されます。例えばⒺⒼが入力されると、検索は中止され、そのときの行が受け入れられたことになります。したがって、ヒストリ・リストの中のそのコマンドが実行されます。

非インクリメンタル ( non-incremental ) ・モードでは、マッチするヒストリ行の検索を開始する前に、検索文字列全体を読み込みます。検索文字列は、ユーザによって入力されたものでも構いませんし、カレント行の内容の一部であっても構いません。

## 18.3 Readline 初期化ファイル

Readline ライブラリには、emacsスタイルのキー・バインディングがデフォルトで組み込まれていますが、異なるキー・バインディングを使うこともできます。ホーム・ディレクトリ内のファイル *inputrc* にコマンドを記述することで、誰でも Readline を使うプログラムをカスタマイズすることができます。このファイルの名前は、環境変数 INPUTRC の値から取られます。この変数に値がセットされていない場合のデフォルトは、`~/ .inputrc` です。

Readline ライブラリを使うプログラムが起動されると、初期化ファイルが読み込まれ、キー・バインディングが設定されます。

さらに、`C-x C-r` コマンドを実行すると、この初期化ファイルが再読み込みされます。初期化ファイルに変更が加えられていれば、その変更が反映されます。

### 18.3.1 Readline 初期化ファイルの構文

Readline 初期化ファイルの中では、ほんの少数の基本的な構文だけが使用できます。空行は無視されます。`#` で始まる行はコメントです。`$` で始まる行は、条件構文を表わします ( see Section 18.3.2 [Conditional Init Constructs], page 156 )。その他の行は、変数設定とキー・バインディングを示します。

**変数設定**      初期化ファイルの中で `set` コマンドを使用して Readline の変数の値を変更することによって、Readline の実行時の振る舞いを変更することができます。デフォルトの Emacs スタイルのキー・バインディングを変更して、`vi` の行編集コマンドを使用できるようにするには、以下のようにします。

```
set editing-mode vi
```

以下の変数によって、実行時の振る舞いのかなりの部分が変更可能です。

**bell-style**

Readline が端末のベル音を鳴らしたいと判断した場合に、何が起るかを制御します。`'none'` がセットされると、Readline はベル音を鳴らしません。`'visible'` がセットされると、視覚的なベル<sup>1</sup> が利用可能であれば、それを使います。`'audible'` ( デフォルト ) がセットされると、Readline は、端末のベル音を鳴らそうと試みます。

**comment-begin**

`insert-comment` コマンドが実行されたときに、行の先頭に挿入される文字列です。デフォルトの値は `"#"` です。

**completion-ignore-case**

`'on'` がセットされると、Readline は、大文字・小文字を区別せずに、ファイル名のマッチングや補完を行います。デフォルトの値は `'off'` です。

**completion-query-items**

ユーザに対して補完候補の一覧を見たいかどうか問い合わせるタイミングを決定する、補完候補の数です。補完候補の数がこの値よりも多いと、Readline は、補完候補の一覧を見たいかどうかをユーザに対して問い合わせることになります。この値よりも少ない場合は、問い合わせを行うことなく一覧を表示します。デフォルトの境界は 100 です。

<sup>1</sup> 訳注：ベル音を鳴らす代わりに、画面表示をフラッシュさせることを表わしています。

**convert-meta**

‘on’がセットされると、Readline は、第 8 ビットがセットされている文字を ASCII のキー・シーケンスに変換します。これは、該当文字の第 8 ビットを落として、その前に `(ESC)` 文字を付加することで、メタ・プレフィックス・キー・シーケンス ( meta-prefixed key sequence ) に変換することによって行われます。デフォルトの値は ‘on’ です。

**disable-completion**

‘On’がセットされると、Readline は単語補完を抑制します。補完文字 ( completion character ) は、あたかも self-insert にマップされたかのように、行内に挿入されます。デフォルトは ‘off’ です。

**editing-mode**

editing-mode 変数は、デフォルトで使用するキー・バインディングの種類を制御します。Readline は、デフォルトの状態では、Emacs 編集モードで起動します。このモードは、キー・ストロークが Emacs に非常に良く似ています。この変数は、emacs と vi のどちらかに設定することができます。

**enable-keypad**

‘on’がセットされると、Readline は、呼び出されたときに、アプリケーション・キーパッド ( application keypad ) を有効にすることを試みます。システムによっては、矢印キーを使用できるようにするために、これが必要となります。デフォルトは ‘off’ です。

**expand-tilde**

‘on’がセットされると、Readline が単語補完を試みる際に、チルダの展開が行われます。デフォルトは ‘off’ です。

**horizontal-scroll-mode**

この変数は、‘on’ と ‘off’ のどちらかに設定することができます。これを ‘on’ に設定すると、1 行のテキストの長さがスクリーン幅よりも長い場合に、編集中の行のテキストが次の行に折り返すことなく、同じ行の上で水平方向にスクロールするようになります。デフォルトでは、この変数には ‘off’ がセットされています。

**keymap**

Readline が認識している、キー・バインディング・コマンドのカレントなキーマップをセットします。セットすることのできる keymap 名は、emacs、emacs-standard、emacs-meta、emacs-ctlx、vi、vi-command、vi-insert です。vi は vi-command と同等です。また、emacs は emacs-standard と同等です。デフォルトの値は、emacs です。editing-mode 変数の値も、デフォルトのキーマップに影響を及ぼします。

**mark-directories**

‘on’がセットされると、補完されたディレクトリ名の後ろにスラッシュが付加されます。デフォルトは ‘on’ です。

**mark-modified-lines**

この変数に ‘on’ がセットされると、Readline は、変更されたヒストリ行の先頭にアスタリスク ( ‘\*’ ) を表示します。この変数は、デフォルトでは ‘off’ です。

**input-meta**

‘on’がセットされると、Readline は、8 ビット入力に対する端末側のサポートがどうであれ、8 ビット入力を有効にします（読み込まれた文字の第 8 ビットを落としません）。デフォルト値は ‘off’ です。meta-flag は、この変数の別名です。

**output-meta**

‘on’がセットされると、Readline は、第 8 ビットがセットされている文字を、メタ・プレフィックス・エスケープ・シーケンス ( meta-prefixed escape sequence ) としてではなく、直接表示します。デフォルトは ‘off’ です。

**print-completions-horizontally**

‘on’がセットされると、Readline は、マッチする補完候補をアルファベット順にソートして、画面の下向きにではなく、水平方向に並べて表示します。デフォルトは ‘off’ です。

**show-all-if-ambiguous**

補完関数のデフォルトの振る舞いを変更します。‘on’がセットされると、複数の補完候補を持つ単語は、ベル音を鳴らすことなく、直ちに補完候補を一覧表示させます。デフォルト値は ‘off’ です。

**visible-stats**

‘on’がセットされると、補完候補を一覧表示する際に、ファイル・タイプを示す文字がファイル名の後ろに付加されます。デフォルトは ‘off’ です。

**キー・バインディング**

初期化ファイルの中でキー・バインディングを制御するための構文は単純です。まず、キー・バインディングを変更したいコマンドの名前を知っている必要があります。以下のセクションにおいて、コマンドの名前、そのコマンドにデフォルトのキー・バインディングがある場合はそのバインディング、および、そのコマンドが何をするものであるかについての簡単な説明を、一覧にして示します。

コマンドの名前を知っていれば、初期化ファイルの中で、コマンドにバインドしたいキーの名前、コロン、そして最後にコマンドの名前を、1 行にして記述するだけです。キーの名前は、好みに応じて異なる方法で表現することができます。

**keyname: function-name or macro**

keyname は、英語で記述されたキーの名前です。例えば、以下のようになります。

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: "> output"
```

上の例では、`(C-u)` が関数 universal-argument にバインドされ、`(C-o)` がその右側に記述されたマクロ（行内に ‘> output’ というテキストを挿入するマクロ）を実行するようバインドされます。

**"keyseq": function-name or macro**

前の例の keyname とは異なり、keyseq には、キー・シーケンス全体を示す文字列を指定することができます。これは、キー・シーケンスを二重引用符で囲むことによって実現されます。以下の例に示すように、いくつかの GNU Emacs スタイルのキー・エスケープを使うことができますが、特殊文字の名前は認識されません。

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

上の例では、`(C-u)`が(最初の例と同様)関数 `universal-argument` に、`(C-x) (C-r)`が関数 `re-read-init-file`に、`(ESC) (F1) (F1) (F1)`が `'Function Key 1'`というテキストを挿入するよう、それぞれバインドされています。

キー・シーケンスを指定する際には、以下の GNU Emacs スタイルのエスケープ・シーケンスが利用できます。

|                  |                |
|------------------|----------------|
| <code>\C-</code> | コントロール・プレフィックス |
| <code>\M-</code> | メタ・プレフィックス     |
| <code>\e</code>  | エスケープ文字        |
| <code>\\</code>  | バックスラッシュ       |
| <code>\"</code>  | <code>"</code> |
| <code>\'</code>  | <code>'</code> |

GNU Emacs スタイルのエスケープ・シーケンスに加えて、別のバックスラッシュ・エスケープ群が利用できます。

|                    |                                                              |
|--------------------|--------------------------------------------------------------|
| <code>\a</code>    | 警告 (ベル)                                                      |
| <code>\b</code>    | バックスペース                                                      |
| <code>\d</code>    | 削除                                                           |
| <code>\f</code>    | フォーム・フィード                                                    |
| <code>\n</code>    | 改行                                                           |
| <code>\r</code>    | 復帰 (carriage return)                                         |
| <code>\t</code>    | 水平タブ                                                         |
| <code>\v</code>    | 垂直タブ                                                         |
| <code>\nnn</code>  | ASCII コードが 8 進数値の <code>nnn</code> (1 個以上 3 個以下の数字) に相当する文字  |
| <code>\xnnn</code> | ASCII コードが 16 進数値の <code>nnn</code> (1 個以上 3 個以下の数字) に相当する文字 |

マクロのテキストを入力する際には、マクロ定義であることを示すために、単一引用符または二重引用符を使わなければなりません。引用符に囲まれないテキストは、関数名であると見なされます。マクロ本体においては、上記のバックスラッシュ・エスケープは展開されます。バックスラッシュとそれに続く文字の組み合わせがバックスラッシュ・エスケープに該当しない場合、マクロのテキストの中のバックスラッシュは、`"`や`'`も含めて、直後にある文字を引用します。例えば、以下のバインディングによって、`\C-x \`は、行内に`\`を 1 つ挿入することになります。

```
"\C-x\\": "\\\""
```

### 18.3.2 条件初期化構文

Readline は、C のプリプロセッサにおける条件コンパイル機能と質的に類似した機能を実装しています。これによって、あるテストの結果に応じてキー・バインディングや変数設定が実行されるようにすることができます。4 種類のパーサ指示子が使われます。

**\$if**            \$if は、編集モード、使用されている端末、あるいは、Readline を使用しているアプリケーションに応じてバインディングが行われるようにすることを可能にします。\$if の後ろに、テストされる内容が行末まで続きます。テストされる内容をほかのものと分離するために特別に文字を使う必要はありません。

**mode**            Readline が emacs モードと vi モードのどちらで動作しているかをテストするために、\$if 指示子の一形式である mode= が使用されます。例えば、Readline が emacs モードで開始されている場合にのみ、emacs-standard や emacs-ctlx のキーマップでバインディングをセットするようにするために、これを 'set keymap' コマンドと組み合わせて使用することができます。

**term**            term= という形式は、端末のファンクション・キーによって特定のキー・シーケンスが出力されるようなバインディングを行うなどの目的で、端末固有のキー・バインディングを組み込むために使用することができます。 '=' の右側の単語は、端末の完全名と、端末の名前のうち最初の '-' までの部分の両方に対してテストされます。これにより、例えば sun は、sun と sun-cmd の両方にマッチすることになります。

**application**    application は、アプリケーション固有の設定を組み込むために使用されます。Readline ライブラリを使用する個々のプログラムがセットする application name (アプリケーション名) をテストすることができます。特定のプログラムにとって役に立つ関数に対してキー・シーケンスをバインドするために、これを使用することができます。例えば以下のコマンドは、Bash において、カレントな単語、または、1 つ前の単語を引用符で囲むキー・シーケンスを追加します。

```
$if Bash
# カレントな単語、または、1 つ前の単語を引用符で囲む
"\C-xq": "\eb"\ef\"
$endif
```

**\$endif**          このコマンドは、前の例が示すように、\$if コマンドを終わらせます。

**\$else**           \$if 指示子から枝分かれしたこの部分に記述されたコマンドは、テスト結果が偽であった場合に実行されます。

**\$include**        この指示子は、引数としてファイル名を 1 つ取り、そのファイルからコマンドとバインディングを読み込みます。

```
$include /etc/inputrc
```

### 18.3.3 初期化ファイルのサンプル

以下に、inputrc ファイルの実例を示します。この中では、キー・バインディング、変数割り当て、条件構文の例が示されています。



```

# このファイルは、Gnu Readline ライブラリを使うプログラムの行入力編集
# の振る舞いを制御する。Gnu Readline ライブラリを使うプログラムには、
# FTP、Bash、Gdb などがある。
#
# inputrc ファイルは、C-x C-r によって再読み込みすることができる。
# '##' で始まる行は、コメントである。
#
# 最初に、/etc/Inputrc からシステム全体のバインディングと変数割り当て
# を取り込む。
$include /etc/Inputrc

#
# emacs モードにおける種々のバインディングをセットする。

set editing-mode emacs

$if mode=emacs

Meta-Control-h: backward-kill-word 関数名の後ろのテキストは無視される。

#
# キーパッド・モードにおける矢印キー
#
#"M-OD":      backward-char
#"M-OC":      forward-char
#"M-OA":      previous-history
#"M-OB":      next-history
#
# ANSI モードにおける矢印キー
#
"\M-[D":      backward-char
"\M-[C":      forward-char
"\M-[A":      previous-history
"\M-[B":      next-history
#
# 8ビット・キーパッド・モードにおける矢印キー
#
#"M-\C-OD":   backward-char
#"M-\C-OC":   forward-char
#"M-\C-OA":   previous-history
#"M-\C-OB":   next-history
#
# 8ビット ANSI モードにおける矢印キー
#
#"M-\C-[D":   backward-char
#"M-\C-[C":   forward-char
#"M-\C-[A":   previous-history
#"M-\C-[B":   next-history

```



```
C-q: quoted-insert

$endif

# 旧スタイルのバインディング。これがたまたまデフォルトでもある。
TAB: complete

# シェルとのやりとりにおいて便利なマクロ
$if Bash
# パス ( PATH ) の編集
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# 引用符で囲まれた単語を入力するための準備 -- 先頭と末尾の二重引用符
# を挿入して、先頭の引用符の直後に移動
"\C-x\"": "\""\C-b"
# バックスラッシュを挿入
# ( シーケンスやマクロにおいて、バックスラッシュ・エスケープをテストする )
"\C-x\\": "\\\"
# カレントな単語、または、1 つ前の単語を引用符で囲む
"\C-xq": "\eb"\ef\"
# バインドされていない行再表示コマンドにバインディングを追加
"\C-xr": redraw-current-line
# カレント行において変数を編集
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# 視覚的なベルが利用可能であれば、それを使う
set bell-style visible

# 読み込みの際に、文字の第 8 ビットを落とさない
set input-meta on

# iso-latin1 文字は、プレフィックス・メタ・シーケンスに変換せず、
# そのまま挿入する
set convert-meta off

# 第 8 ビットがセットされている文字を、メタ・プレフィックス文字として
# ではなく、直接表示する
set output-meta on

# ある単語について、150 を超える補完候補が存在する場合、ユーザに対して
# すべてを表示させたいかどうかを問い合わせる
set completion-query-items 150

# FTP 用
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg
```

\$endif

## 18.4 バインド可能な Readline コマンド

このセクションでは、キー・シーケンスにバインドすることが可能な Readline コマンドについて説明します。

### 18.4.1 移動のためのコマンド

beginning-of-line (C-a)

カレント行の先頭に移動します。

end-of-line (C-e)

行の末尾に移動します。

forward-char (C-f)

1 文字分先に進みます。

backward-char (C-b)

1 文字分後へ戻ります。

forward-word (M-f)

次の単語の末尾へ移動します。単語は、文字と数字により構成されます。

backward-word (M-b)

現在カーソルが指している単語、または、1 つ前の単語の先頭に移動します。単語は、文字と数字により構成されます。

clear-screen (C-l)

画面を消去し、カレント行を再表示します。その際、カレント行が画面の一番上になるようにします。

redraw-current-line ( )

カレント行を再表示します。デフォルトでは、このコマンドはバインドされていません。

### 18.4.2 ヒストリを操作するためのコマンド

accept-line (Newline, Return)

カーソルの位置がどこにあっても、その行を受け取ります。この行が空行ではない場合、それをヒストリ・リストに追加します。この行がヒストリ行である場合は、そのヒストリ行を最初の状態に復元します。

previous-history (C-p)

ヒストリ・リストを 1 つ上に移動します。

next-history (C-n)

ヒストリ・リストを 1 つ下に移動します。

beginning-of-history (M-<)

ヒストリの最初の行に移動します。

`end-of-history (M->)`

入力履歴の最後の行、すなわち、現在入力中の行に移動します。

`reverse-search-history (C-r)`

カレント行から始めて上の方向へ検索を行います。必要に応じて履歴の上の方へ移動します。インクリメンタルな検索を行います。

`forward-search-history (C-s)`

カレント行から始めて下の方向へ検索を行います。必要に応じて履歴の下の方へ移動します。インクリメンタルな検索を行います。

`non-incremental-reverse-search-history (M-p)`

カレント行から始めて、必要に応じて履歴の上の方へ移動しつつ、非インクリメンタルな検索を使って、ユーザによって提供された文字列を上の方へ検索します。

`non-incremental-forward-search-history (M-n)`

カレント行から始めて、必要に応じて履歴の下の方へ移動しつつ、非インクリメンタルな検索を使って、ユーザによって提供された文字列を下の方へ検索します。

`history-search-forward ( )`

カレント行の先頭からカレントなカーソル位置 (ポイント) までの間の文字列を、履歴の中で下の方向へ検索します。これは、非インクリメンタルな検索です。デフォルトでは、このコマンドはバインドされていません。

`history-search-backward ( )`

カレント行の先頭からポイントまでの間の文字列を、履歴の中で上の方向へ検索します。これは、非インクリメンタルな検索です。デフォルトでは、このコマンドはバインドされていません。

`yank-nth-arg (M-C-y)`

1 つ前に実行されたコマンドの最初の引数 (通常は、1 つ前の行の 2 つめの単語) を挿入します。引数  $n$  を指定すると、1 つ前に実行されたコマンドの  $n$  番目の単語を挿入します (1 つ前に実行されたコマンドの中の最初の単語を、0 番目の単語とします)。負の値を引数に指定すると、1 つ前に実行されたコマンドの後ろから数えて  $n$  番目の単語を挿入します。

`yank-last-arg (M-., M-_)`

1 つ前に実行されたコマンドの最後の引数 (1 つ前の履歴・エントリの最後の単語) を挿入します。引数を指定すると、`yank-nth-arg` と同じように動作します。`yank-last-arg` を連続して実行すると、履歴・リストを遡って移動していきます。したがって、各行の最後の引数が順番に挿入されていきます。

### 18.4.3 テキストを変更するためのコマンド

`delete-char (C-d)`

カーソル位置にある文字を削除します。カーソルが空行の先頭にあり、最後に入力された文字が `delete-char` にバインドされていない場合は、EOFを返します。

`backward-delete-char (Rubout)`

カーソル位置の前にある文字を削除します。数値引数を指定すると、文字を削除するのではなくキル (kill) するよう指示したことになります。

`quoted-insert (C-q, C-v)`

このコマンドに続けて入力する文字をそのまま行に追加します。これが、例えば `(C-q)` のようなキー・シーケンスを挿入する方法です。

`tab-insert (M-TAB)`

タブを挿入します。

`self-insert (a, b, A, 1, !, ...)`

その文字自身を挿入します。

`transpose-chars (C-t)`

カーソルの前にある文字をドラッグして、カーソル位置にある文字の後ろに持っていきます。カーソル自身も同様に前進させます。挿入ポイントが行末にある場合には、行の最後の 2 文字を入れ替えます。負の引数を与えても機能しません。

`transpose-words (M-t)`

カーソルの前にある単語をドラッグして、カーソルの後ろにある単語の後ろに持っていきます。カーソル自身も、カーソルの後ろにある単語の後ろに移動します。

`upcase-word (M-u)`

カレントな (あるいは、その 1 つ後ろの) 単語の中のすべての文字を大文字に変換します。負の引数を指定すると、1 つ前の単語の中のすべての文字を大文字に変換しますが、カーソルは移動しません。

`downcase-word (M-l)`

カレントな (あるいは、その 1 つ後ろの) 単語の中のすべての文字を小文字に変換します。負の引数を指定すると、1 つ前の単語の中のすべての文字を小文字に変換しますが、カーソルは移動しません。

`capitalize-word (M-c)`

カレントな (あるいは、その 1 つ後ろの) 単語の先頭文字を大文字に、それ以外の位置にある文字を小文字に変換します。負の引数を指定すると、1 つ前の単語に対して同様の変換を行いますが、カーソルは移動しません。

#### 18.4.4 キル (kill) と再挿入 (yank)

`kill-line (C-k)`

カレントなカーソル位置から行末までのテキストをキル (kill) します。

`backward-kill-line (C-x Rubout)`

行の先頭までのテキストをキルします。

`unix-line-discard (C-u)`

カーソル位置から逆方向にカレント行の先頭までをキルします。キルされたテキストは、キル・リング (kill-ring) に退避されます。

`kill-whole-line ()`

カーソルの位置にかかわらず、カレント行のすべての文字をキルします。デフォルトでは、バインドされていません。

**kill-word (M-d)**

カーソル位置からカレントな単語の末尾までをキルします。カーソルが単語の間にある場合は、次の単語の末尾までをキルします。単語の境界は、forward-wordの場合と同様です。

**backward-kill-word (M-DEL)**

カーソルの前にある単語をキルします。単語の境界は、backward-wordの場合と同様です。

**unix-word-rubout (C-w)**

空白類<sup>2</sup>を単語の境界として、カーソルの前にある単語をキルします。キルされた単語は、キル・リングに退避されます。

**delete-horizontal-space ( )**

ポイントの前後にある、すべての空白 (スペース) とタブを削除します。デフォルトでは、バインドされていません。

**kill-region ( )**

ポイントとマーク (待避されたカーソル位置) の間のテキストをキルします。このテキストは、領域 (region) と呼ばれます。デフォルトでは、このコマンドはバインドされていません。

**copy-region-as-kill ( )**

領域 (region) 内のテキストを、直ちに再挿入 (yank) できるよう、キル・バッファ (kill buffer) にコピーします。デフォルトでは、このコマンドはバインドされていません。

**copy-backward-word ( )**

ポイントの前にある単語をキル・バッファにコピーします。単語の境界は、backward-wordの場合と同様です。デフォルトでは、このコマンドはバインドされていません。

**copy-forward-word ( )**

ポイントの後ろにある単語をキル・バッファにコピーします。単語の境界は、forward-wordの場合と同様です。デフォルトでは、このコマンドはバインドされていません。

**yank (C-y)**

キル・リングの一番上の位置にあるテキストを、バッファ内のカレントなカーソル位置に再挿入 (yank) します。

**yank-pop (M-y)**

キル・リングを回転させ、新しく一番上の位置にきたテキストを再挿入 (yank) します。1 つ前に実行したコマンドが、yank または yank-pop であった場合のみ、このコマンドを実行することができます。

### 18.4.5 数値引数の指定

**digit-argument (M-0, M-1, ... M--)**

既に蓄積済みの引数にこの数字を追加するか、または、この数字によって新しい引数を開始します。負の引数を指定するには、先頭を(M-)とします。

---

<sup>2</sup> 訳注: 空白 (スペース)、水平タブ、改行、垂直タブ、フォーム・フィード

`universal-argument ()`

これは、引数を指定する別の方法です。このコマンドの後に、場合によって先頭にマイナス記号の付く、1つ以上の数字が続く場合には、それらの数字が引数を定義します。このコマンドの後ろに数字が続く場合には、`universal-argument`を再実行することによって、その数字引数を終わらせることができます。しかし、このコマンドの後ろに数字が続かない場合の再実行は、無視されます。特殊なケースとして、このコマンドの直後に数字でもマイナス記号でもない文字が続く場合、次に実行されるコマンドの引数カウントは4倍されます。引数カウントの初期値は1です。したがって、この関数を最初に実行した後は、引数カウントは4になり、2回目に実行した後は16になります。以下、同様です。デフォルトでは、キーへのバインドはされていません。

### 18.4.6 Readline による入力補完

`complete (TAB)`

カーソルの前にあるテキストの補完を試みます。これは、アプリケーション固有の動作をします。通常、引数としてファイル名を入力しているときには、ファイル名を補完することができます。コマンド名を入力しているときには、コマンド名を補完することができます。GDB に対してシンボル名を入力しているときには、シンボル名を補完することができます。Bash に対して変数名を入力しているときには、変数名を補完することができます。

`possible-completions (M-?)`

カーソルの前にあるテキストの補完候補を一覧表示します。

`insert-completions (M-*)`

`possible-completions`を実行すれば生成されたであろうテキストの補完候補をすべて、ポイントの前に挿入します。

`menu-complete ()`

`complete`に似ていますが、補完されるべき単語を、補完候補の一覧の中の1つと置き換えます。`menu-complete`を繰り返し実行すると、補完候補の一覧から順番に1つずつ補完候補が挿入されていきます。候補一覧の終端に達すると、ベル音が鳴らされ、補完前のテキストが復元されます。引数  $n$  を指定すると、補完候補の一覧の中で  $n$  個先に移動します。一覧を逆方向に戻るために、負の引数を指定することができます。このコマンドは、TABにバインドすることを意図したのですが、デフォルトではバインドされていません。

### 18.4.7 キーボード・マクロ

`start-kbd-macro (C-x )`

カレントなキーボード・マクロの構成要素として入力される文字の保存を開始します。

`end-kbd-macro (C-x )`

カレントなキーボード・マクロの構成要素として入力された文字の保存を終了して、そのキーボード・マクロの定義を保存します。

`call-last-kbd-macro (C-x e)`

最後に定義されたキーボード・マクロを再実行します。マクロの中の文字群が、あたかもキーボードから入力されたかのように、現われます。

### 18.4.8 その他のコマンド

`re-read-init-file (C-x C-r)`

`inputrc` ファイルの内容を読み込み、その中にあるバインディングや変数割り当てをすべて組み込みます。

`abort (C-g)`

カレントな編集コマンドの実行を停止し、(`bell-style`の設定次第では)端末のベル音を鳴らします。

`do-uppercase-version (M-a, M-b, M-x, ...)`

メタ化された文字 `x` が小文字である場合、対応する大文字にバインドされているコマンドを実行します。

`prefix-meta (ESC)`

次に入力される文字をメタ化します。これは、メタ・キーのないキーボード用のコマンドです。‘ESC f’を入力するのは、‘M-f’を入力するのと同じことです。

`undo (C-_, C-x C-u)`

インクリメンタルな取り消し処理を実行します。取り消す内容は、各行ごとに別々に記憶されています。

`revert-line (M-r)`

行に加えられたすべての変更を取り消します。これは、`undo`コマンドを、行を元の状態に戻すのに必要な回数繰り返して実行するようなものです。

`tilde-expand (M-~)`

カレントな単語に対して、チルダ展開を実行します。

`set-mark (C-@)`

カレントなポイントにマークをセットします。数値引数があれば、その位置にマークがセットされます。

`exchange-point-and-mark (C-x C-x)`

ポイントとマークを交換します。待避されていた位置がカレントなカーソル位置としてセットされ、元のカーソル位置はマークとして待避されます。

`character-search (C-])`

文字を1つ読み込み、その文字が次に現われるところにポイントを移動します。負の数を指定すると、その文字が以前に現われたところを探します。

`character-search-backward (M-C-])`

文字を1つ読み込み、その文字が前に現われたところにポイントを移動します。負の数を指定すると、その文字が次に現われるところを探します。

`insert-comment (M-#)`

カレント行の先頭に `comment-begin`変数の値が挿入され、挿入後の行が、あたかも改行が入力されたかのように、受け付けられます。

**dump-functions ()**

Readline の出力ストリームに、すべての関数とそのキー・バインディングを出力します。数値引数が指定されると、*inputrc* ファイルの一部として使用することのできる形式に、出力がフォーマットされます。このコマンドは、デフォルトではバインドされていません。

**dump-variables ()**

Readline の出力ストリームに、値をセットすることのできるすべての変数とその値を出力します。数値引数が指定されると、*inputrc* ファイルの一部として使用することのできる形式に、出力がフォーマットされます。このコマンドは、デフォルトではバインドされていません。

**dump-macros ()**

マクロにバインドされているすべての Readline キー・シーケンスと、そのキー・シーケンスが出力する文字列を出力します。数値引数が指定されると、*inputrc* ファイルの一部として使用することのできる形式に、出力がフォーマットされます。このコマンドは、デフォルトではバインドされていません。

## 18.5 Readline の vi モード

Readline ライブラリは、vi の編集機能のフルセットを提供してはいませんが、簡単な行編集を行うのに十分な機能は備えています。Readline の vi モードは、POSIX 1003.2 標準にしたがって動作します。

emacs 編集モードと vi 編集モードを対話的に切り替えるには、コマンド M-C-j ( *toggle-editing-mode* ) を使用してください。Readline のデフォルトは emacs モードです。

vi モードで行入力を行うときには、あたかも 'i' を入力したかのように、最初から「挿入」モードになっています。⏏ (ESC) を押すと「コマンド」モードになり、標準的な vi の移動キーによって行のテキストを編集することができます。すなわち、'k' により前のヒストリ行に移動すること、'j' によって後ろのヒストリ行に移動すること、などが可能です。



## Appendix A ヒストリの対話的な使用

ここでは、ユーザの見地から、GNU ヒストリ・ライブラリの対話的な使い方を説明します。

### A.1 ヒストリの操作

ヒストリ・ライブラリは、`csch`のヒストリ展開機能に似た機能を提供します。以下において、ヒストリ情報を操作するための構文を説明します。

ヒストリ展開は 2 つの部分から構成されます。第 1 の部分で、過去のヒストリのどの行が代替処理に使用されるかが決まります。この行をイベントと呼びます。第 2 の部分で、この行のうちどの部分がカレント行に挿入されるかが決まります。この部分のことをワードと呼びます。GDB は、Bash シェルと同様の方法によって行をワードに分割します。したがって、引用符によって囲まれた複数の英単語 (あるいは UNIX 用語) は 1 つのワードとみなされます。

#### A.1.1 イベント指定子

イベント指定子とは、ヒストリ・リスト内のコマンド行エントリへの参照です。

- !            ヒストリ代替を開始します。ただし次に続く文字が、空白、タブ、行末、`␣`、`␣`である場合は例外です。
- !!            1 つ前のコマンドを参照します。これは、`!-1`と同義です。
- !n            コマンド行番号 *n* のコマンド行を参照します。
- !-n            *n* 行前のコマンド行を参照します。
- !string      コマンドの最初の部分が文字列 *string* で始まるコマンドのうち、最後に実行されたものを参照します。
- !?string[?]      文字列 *string* を含むコマンドのうち、最後に実行されたものを参照します。

#### A.1.2 ワード指定子

コロン ( `:` ) が、イベント指定子とワード指定子の区切り文字になります。ワード指定子が `␣`、`␣`、`␣`、`␣` で始まる場合は、この区切り文字は省略することができます。ワードは行の先頭から番号が付与され、最初のワードは 0 (ゼロ) 番になります。

- 0 (zero)      ゼロ番目のワードです。多くのアプリケーションにおいて、これはコマンド・ワードです。
  - n            *n* 番目のワードです。
  - ^            最初の引数、すなわち 1 番目のワードです。
  - \$            最後の引数です。
  - %            最後に実行された `?string?` 検索にマッチしたワードです。
  - x-y          ある範囲のワードを指します。-y は、0-y の省略形です。
  - \*
- ゼロ番目のワードを除く、すべてのワードです。これは、`1-$`と同義です。イベントの内部にワードが 1 つしかなくても、`␣`の使用はエラーにはなりません。この場合には、空の文字列が返されます。

### A.1.3 修飾子

必須ではないワード指定子に続けて、以下の修飾子を 1 つ以上連続して追加することができます。個々の修飾子の前にコロンの (:) を付けます。

- #           それまで入力されたコマンド行全体です。これは、1 つ前のコマンドではなく、カレントなコマンドを意味します。
- h           パス名の末尾の部分を削除したヘッド部です。
- r           `‘.suffix’` 形式の拡張子を削除したベース名です。
- e           拡張子以外のすべての部分を削除します。
- t           パス名の末尾の部分を残し、それより前にある部分をすべて削除します。
- p           新しいコマンドを表示するだけで実行しません。

## Appendix B ドキュメントのフォーマット

GDB 4 には、PostScript または GhostScript でそのまま印刷できる、フォーマット済みのリファレンス・カードが含まれています。<sup>1</sup> これは、メインのソース・ディレクトリの下に 'gdb' サブディレクトリにあります。PostScript または Ghostscript を使えるプリンタがあれば、'refcard.ps' を使ってすぐにリファレンス・カードを印刷することができます。

GDB 4 には、リファレンス・カードのソースも含まれています。TeX を使えば、以下のようにしてこれをフォーマットすることができます。

```
make refcard.dvi
```

GDB のリファレンス・カードは、米国のレター・サイズ用の紙にランドスケープ・モードで印刷するようにデザインされています。レター・サイズは、横幅が 11 インチ、高さが 8.5 インチです。DVI 出力プログラムへのオプションとして、この印刷形式を指定する必要があります。

すべての GDB ドキュメントは、マシン上で読むことのできるディストリビューションの一部として提供されます。ドキュメントは Texinfo フォーマットで記述されています。これは、単一のソースからオンライン・マニュアルとハードコピー・マニュアルの両方を生成するドキュメント・システムです。Info フォーマット・コマンドの 1 つを使ってオンライン・ドキュメントを作成することができ、TeX (または texi2roff) を使ってハード・コピーの組版ができます。

GDB には、このマニュアルのフォーマット済みのオンライン Info バージョンも含まれています。これは、'gdb' サブディレクトリにあります。メインの Info ファイルは 'gdb-4.18/gdb/gdb.info' で、同じディレクトリにある 'gdb.info\*' にマッチする従属ファイルを参照します。必要であれば、これらのファイルを印刷したり、任意のエディタで表示して読むこともできます。しかし、これらのファイルは、GNU Emacs の info サブシステムや GNU Texinfo の一部として配布されるスタンドアロンの info プログラムを使った方が読みやすいでしょう。

これらの Info ファイルを自分でフォーマットしたいのであれば、texinfo-format-buffer や makeinfo のような Info フォーマット・プログラムが必要になります。

makeinfo がインストールされていて、GDB ソース・ディレクトリのトップ・レベル (バージョン 4.18 では 'gdb-4.18') にいる場合は、以下のようにして Info ファイルを作成することができます。

```
cd gdb
make gdb.info
```

このマニュアルのコピーの組版を行って印刷するには、TeX、TeX の DVI 出力ファイルを印刷するプログラム、および、Texinfo 定義ファイル 'texinfo.tex' が必要です。

TeX は組版プログラムです。TeX は直接ファイルを印刷しませんが、DVI ファイルと呼ばれるものを生成します。組版されたドキュメントを印刷するには、DVI ファイルを印刷するプログラムが必要です。システム上に TeX がインストールされていれば、DVI ファイルを印刷するプログラムも入っている可能性があります。印刷に使われるコマンドの正確な名前はシステムにより異なります。lpr -d が一般によく使われます。また (PostScript プリンタでは) dvips がよく使われます。DVI プリント・コマンドを使う際には、ファイル名に拡張子を付けないか、あるいは、'.dvi' という拡張子を付ける必要があるかもしれません。

また、TeX は 'texinfo.tex' という名のマクロ定義ファイルを必要とします。このファイルは TeX に対して、Texinfo フォーマットで記述されたドキュメントをどのようにして組版するかを教えます。TeX は自分自身では、Texinfo ファイルを読むことも組版することもできません。'texinfo.tex' は GDB とともに配布されていて、'gdb-version-number/texinfo' ディレクトリにあります。

<sup>1</sup> 原注: バージョン 4.18 では 'gdb-4.18/gdb/refcard.ps' です。

$\text{\TeX}$  と  $\text{\DVI}$  印刷プログラムがインストールされていれば、このマニュアルを組版して、印刷することができます。メインのソース・ディレクトリの下に `'gdb'` サブディレクトリ (例えば、`'gdb-4.18/gdb'`) に移動して、以下のように実行します。

```
make gdb.dvi
```

その後、`'gdb.dvi'` を  $\text{\DVI}$  印刷プログラムに渡します。

## Appendix C GDB のインストール

GDB には、インストールのための準備作業を自動化する `configure` スクリプトが付属しています。`configure` を実行した後に `make` を実行することで、`gdb` をビルドすることができます。

<sup>1</sup>

GDB ディストリビューションには、GDB をビルドするのに必要なすべてのソース・コードが、単一のディレクトリの下に収められています。このディレクトリの名前は通常、`'gdb'` の後ろにバージョン番号を付加したものです。

例えば、バージョン 4.18 の GDB ディストリビューションは、`'gdb-4.18'` というディレクトリに収められています。このディレクトリには、以下のものが含まれます。

`gdb-4.18/configure` ( およびサポート・ファイル )

GDB、および、GDB が必要とするすべてのライブラリの構成を行うためのスクリプト

`gdb-4.18/gdb`

GDB 自身に固有のソース

`gdb-4.18/bfd`

Binary File Descriptor ライブラリのソース

`gdb-4.18/include`

GNU インクルード・ファイル

`gdb-4.18/libiberty`

`'-liberty'` フリー・ソフトウェア・ライブラリのソース

`gdb-4.18/opcodes`

opcode テーブルおよび逆アセンブラのライブラリのソース

`gdb-4.18/readline`

GNU コマンドライン・インターフェイスのソース

`gdb-4.18/glob`

GNU ファイル名パターン・マッチング・サブルーチンのソース

`gdb-4.18/malloc`

メモリにマップされる GNU malloc パッケージのソース

GDB の構成とビルドを行う最も簡単な方法は、`'gdb-version-number'` ソース・ディレクトリから `configure` を実行することです。ここでの例では、このディレクトリは `'gdb-4.18'` です。

もしまだ `'gdb-version-number'` ソース・ディレクトリにいないのであれば、まずそこに移動してください。続いて `configure` を実行します。GDB が実行されるプラットフォームの識別子を引数として渡します。

例えば、以下のようにします。

```
cd gdb-4.18
./configure host
make
```

---

<sup>1</sup> 原注：バージョン 4.18 よりさらに新しい GDB を持っている場合には、ソースの中に含まれる `'README'` ファイルを参照してください。このマニュアルの出版後、インストール手順が改善されていることがあるかもしれません。

`host` は、GDB が実行されるプラットフォームを識別する識別子です。例えば `'sun4'` や `'decstation'` などです（多くの場合 `host` は省略することができます。この場合 `configure` は、ユーザのシステムを調べることによって正しい値を推定しようとします）。

`'configure host'` を実行した後に `make` を実行することで、`'bfd'`、`'readline'`、`'mmap'`、`'libiberty'` の各ライブラリがビルドされ、最後に `gdb` 自体がビルドされます。構成されたソース・ファイルやバイナリは、対応するソース・ディレクトリに残されます。

`configure` は Bourne シェル（`/bin/sh`）のスクリプトです。ユーザが別のシェルを実行していて、システムがこのことを自動的に認識してくれない場合は、明示的に `sh` にスクリプトを実行させる必要があるかもしれません。

```
sh configure host
```

バージョン 4.18 のソース・ディレクトリである `'gdb-4.18'` のように、配下に複数のライブラリやプログラムのソース・ディレクトリを含むディレクトリから `configure` を実行すると、`configure` は配下にあるそれぞれのディレクトリのための構成ファイルを作成します（`'--norecursion'` オプションによって、そうしないよう指定した場合は別です）。

GDB ディストリビューションの中のある特定のサブディレクトリを構成したいだけの場合には、そのサブディレクトリから `configure` スクリプトを実行することができます。ただし、`configure` スクリプトへのパスを必ず指定してください。

例えば、バージョン 4.18 では、`bfd` サブディレクトリだけを構成するには以下のようにします。

```
cd gdb-4.18/bfd
../configure host
```

`gdb` はどこにでもインストールできます。あらかじめ固定されたパスは 1 つもありません。ただし、ユーザのパスにある（`'SHELL'` 環境変数により指定される）シェルが誰にでも読み込み可能であることを確かめる必要があります。GDB はシェルを使ってユーザ・プログラムを起動するというのを憶えておいてください。子プロセスが読み込み不可のプログラムである場合、システムによっては、GDB がそれをデバッグするのを拒否します。

## C.1 異なるディレクトリでの GDB のコンパイル

いくつかのホスト・マシンおよびターゲット・マシン用の GDB を実行したい場合、ホストとターゲットの個々の組み合わせ用にコンパイルされた異なる `gdb` が必要になります。`configure` には、個々の構成をソース・ディレクトリではなく個別のサブディレクトリに生成する機能があり、このようなことが簡単にできるように設計されています。ユーザの使っている `make` プログラムに `'VPATH'` 機能があれば（GNU `make` にはあります）、これら個々のディレクトリにおいて `make` を実行することで、そこで指定されている `gdb` プログラムをビルドすることができます。

個別のディレクトリにおいて `gdb` をビルドするには、ソースの置かれている場所を指定するために、`'--srcdir'` オプションを使って `configure` を実行します（同時に、ユーザの作業ディレクトリから `configure` を見つけるためのパスも指定する必要があります。もし、`configure` へのパスが `'--srcdir'` への引数として指定するものと同じであれば、`'--srcdir'` オプションは指定しなくてもかまいません。指定されなければ、同じであると仮定されます）。

例えば、バージョン 4.18 で Sun 4 用の GDB を別のディレクトリにおいて構築するには、以下のようになります。

```
cd gdb-4.18
mkdir ../gdb-sun4
cd ../gdb-sun4
../gdb-4.18/configure sun4
make
```

`configure`が、別の場所にあるソース・ディレクトリを使って、ある構成を作成する際には、ソース・ディレクトリ配下のディレクトリ・ツリーと同じ構造のディレクトリ・ツリーを(同じ名前で)バイナリ用に作成します。この例では、Sun 4 用のライブラリ `'libiberty.a'` は `'gdb-sun4/libiberty'` ディレクトリに、GDB 自身は `'gdb-sun4/gdb'` ディレクトリにそれぞれ作成されます。

いくつかの GDB の構成を別々のディレクトリにおいてビルドする理由としてよくあるのが、クロス・コンパイル( GDB はホストと呼ばれるあるマシン上で動作し、ターゲットと呼ばれる別のマシンで実行されているプログラムをデバッグする ) 環境用に GDB を構成する場合です。クロス・デバッグのターゲットは、`configure`に対する `'--target=target'` オプションを使って指定します。

プログラムやライブラリをビルドするために `make` を実行するときには、構成されたディレクトリにいないければなりません。これは、`configure` を実行したときにいたディレクトリ(または、そのサブディレクトリの1つ)です。

`configure`が個別のソース・ディレクトリに生成した Makefile は再帰的に呼び出されます。`'gdb-4.18'` (あるいは、`'--srcdir=dirname/gdb-4.18'`により構成された別のディレクトリ)などのソース・ディレクトリにおいて `make` を実行すると、必要とされるすべてのライブラリがビルドされ、その後に GDB がビルドされることになります。

複数のホストまたはターゲットの構成が、異なる複数のディレクトリに存在する場合、(例えば、それらが個々のホスト上に NFS マウントされている場合) 並行して `make` を実行することができます。複数の構成が互いに干渉し合うということはありません。

## C.2 ホストとターゲットの名前の指定

`configure` スクリプトにおけるホストおよびターゲットの指定方法は、3 つの名称部分を持ちますが、あらかじめ定義された短い別名もいくつかサポートされています。完全名は、以下のようなパターンの 3 つの情報部分を持ちます。

*architecture-vendor-os*

例えば、ホストを指定する引数 *host* として、あるいは、`--target=target` オプションの *target* の部分に、`sun4` という別名を使うことができます。これと同等の完全名は `'sparc-sun-sunos4'` です。

GDB に付属している `configure` スクリプトには、サポートされているすべてのホスト名、ターゲット名、および、別名を問い合わせするための機能はありません。`configure` は、Bourne シェル・スクリプトの `config.sub` を呼び出すことによって、省略名を完全名に対応付けします。このスクリプトを使って、省略名の意味が推測と合っているかどうかをテストすることもできます。以下に例を示します。

```
% sh config.sub i386-linux
i386-pc-linux-gnu
% sh config.sub alpha-linux
alpha-unknown-linux-gnu
% sh config.sub hp9k700
hppa1.1-hp-hpux
% sh config.sub sun4
sparc-sun-sunos4.1.1
% sh config.sub sun3
m68k-sun-sunos4.1.1
% sh config.sub i986v
Invalid configuration 'i986v': machine 'i986v' not recognized
```

`config.sub` も、GDB ディストリビューションの一部としてソース・ディレクトリ(バージョン 4.18 では、`'gdb-4.18'`)に入っています。

### C.3 configureオプション

以下に、GDBをビルドする上でほとんどの場合に役に立つconfigureのオプションと引数の要約を示します。configureには、ここには挙げられていないオプションもいくつかあります。configureに関する完全な説明については、See Info file ‘configure.info’, node ‘What Configure Does’.

```
configure [--help]
          [--prefix=dir]
          [--exec-prefix=dir]
          [--srcdir=dirname]
          [--norecursion] [--rm]
          [--target=target]
          host
```

そうしたいのであれば、‘--’ではなく単一の‘-’でオプションを始めることもできますが、‘--’を使うとオプション名を省略することができます。

`--help`      configureの実行方法の簡単な要約を表示します。

`--prefix=dir`  
プログラムおよびファイルをディレクトリ ‘*dir*’ にインストールするようソースを構成します。

`--exec-prefix=dir`  
プログラムをディレクトリ ‘*dir*’ にインストールするようソースを構成します。

`--srcdir=dirname`  
注意: このオプションを使うには、GNU make、あるいは、VPATH機能を持つ他の make を使用する必要があります。  
GDB ソース・ディレクトリとは別のディレクトリに構成を作成する場合に、このオプションを使用します。特に、いくつかの構成を別々のディレクトリにおいて同時に作成（かつ維持）する場合に、このオプションを使うことができます。configureは、構成に固有のファイルをカレント・ディレクトリに書き込みますが、*dirname* ディレクトリにあるソースを使うように、それらのファイルを調整します。configureは、*dirname* ディレクトリ配下のソース・ディレクトリ・ツリーと同じ構造を持つディレクトリ・ツリーを、作業ディレクトリの下に作成します。

`--norecursion`  
configureが実行されたディレクトリ・レベルだけを構成します。サブディレクトリまで含めて構成することはしません。

`--target=target`  
指定されたターゲット *target* で実行するプログラムをクロス・デバッグするために、GDBを構成します。このオプションを指定しないと、GDB と同じマシン（ホスト）で実行されるプログラムをデバッグするよう、GDB は構成されます。  
利用可能なすべてのターゲットの一覧を生成する、便利な方法はありません。

*host* ...      指定されたホスト *host* 上で実行されるよう GDB を構成します。  
利用可能なすべてのホストの一覧を生成する、便利な方法はありません。

ほかにも利用可能な多くのオプションがありますが、これは通常、特殊な目的にのみ必要とされるものです。



## インデックス

|                                                                                          |     |                                                                  |     |
|------------------------------------------------------------------------------------------|-----|------------------------------------------------------------------|-----|
| リモート・スタブ、サポート・ルーチン . . . . .                                                             | 112 | 要約、リモート・シリアル・デバッグ処理の [よ<br>うやく、リモート・シリアル・デバッグし<br>よりの] . . . . . | 113 |
| リモート・シリアル・プロトコル . . . . .                                                                | 114 | 概要、リモート・シリアル・デバッグ処理の [が<br>いよう、リモート・シリアル・デバッグし<br>よりの] . . . . . | 110 |
| リファレンス・カード . . . . .                                                                     | 169 | リモート・デバッグ . . . . .                                              | 110 |
| プロンプト . . . . .                                                                          | 131 | メモリにマップされたシンボル・ファイル . .                                          | 100 |
| ブレイクポイントとスレッド . . . . .                                                                  | 45  | オーバーロードされている関数のブレイク [オー<br>バーロードされているかんすうのブレイク]<br>. . . . .     | 84  |
| ブレイクポイント・コマンド . . . . .                                                                  | 39  | リモート・デバッグ処理、スタブの実例 [リモー<br>ト・デバッグしより、スタブのじつれい]<br>. . . . .      | 114 |
| フレーム . . . . .                                                                           | 47  | 呼び出し、オーバーロードされた関数の [よびだ<br>し、オーバーロードされたかんすうの] . .                | 83  |
| フォーマットのオプション . . . . .                                                                   | 65  | 選択、ターゲットのバイト・オーダの [せんた<br>く、ターゲットのバイト・オーダの] . . .                | 109 |
| ヒストリ・ファイル . . . . .                                                                      | 132 | バイト・オーダの選択、ターゲットの [バイト・<br>オーダのせんたく、ターゲットの] . . . . .            | 109 |
| ハードウェア・ウォッチポイント . . . . .                                                                | 33  | オーバーロードされた関数の呼び出し [オーバ<br>ーロードされたかんすうのよびだし] . . . . .            | 83  |
| ドキュメント . . . . .                                                                         | 169 | リモート・シリアル・スタブの一覧 [リモート・<br>シリアル・スタブのいちらん] . . . . .              | 111 |
| デバッグのクラッシュ . . . . .                                                                     | 145 | 割り込み、リモート・プログラムの [わりこみ、<br>リモート・プログラムの] . . . . .                | 114 |
| ソフトウェア・ウォッチポイント . . . . .                                                                | 33  |                                                                  |     |
| スレッドとウォッチポイント . . . . .                                                                  | 34  |                                                                  |     |
| シンボルのオーバーロード . . . . .                                                                   | 40  |                                                                  |     |
| シミュレータ . . . . .                                                                         | 128 |                                                                  |     |
| コマンド・フック . . . . .                                                                       | 138 |                                                                  |     |
| コマンド・ファイル . . . . .                                                                      | 139 |                                                                  |     |
| クラッシュ、デバッグの . . . . .                                                                    | 145 |                                                                  |     |
| インストール . . . . .                                                                         | 171 |                                                                  |     |
| 消去、ブレイクポイント、ウォッチポイント、<br>キャッチポイントの [しょうきょ、ブレイク<br>ポイント、ウォッチポイント、キャッチポイ<br>ントの] . . . . . | 36  |                                                                  |     |
| 削除、ブレイクポイント、ウォッチポイント、<br>キャッチポイントの [さくじょ、ブレイクポ<br>イント、ウォッチポイント、キャッチポイン<br>トの] . . . . .  | 36  |                                                                  |     |
| ブレイクポイント、ウォッチポイント、キャッチ<br>ポイントの削除 [ブレイクポイント、ウォッ<br>チポイント、キャッチポイントのさくじょ]<br>. . . . .     | 36  |                                                                  |     |

|                                               |     |                                        |     |
|-----------------------------------------------|-----|----------------------------------------|-----|
| デバッグ・ターゲット .....                              | 105 | シリアル接続、デバッグ処理 [シリアルせつぞく、デバッグしより] ..... | 115 |
| ダイナミック・リンク .....                              | 101 | 識別子、システムのスレッド [しきべつし、システムのスレッド] .....  | 25  |
| シンボル・ファイル、メモリにマップされた .....                    | 100 | シリアル装置、日立マイクロ [シリアルそうち、日立マイクロ] .....   | 125 |
| ターゲットのバイト・オーダの選択 [ターゲットのバイト・オーダのせんたく] .....   | 109 | 実例、リモート・スタブの [じつれい、リモート・スタブの] .....    | 114 |
| シリアル回線速度、日立マイクロ [シリアルかいせんそくど、ひたちマイクロ] .....   | 125 | 実例、スタブのデバッグの [じつれい、スタブのデバッグの] .....    | 114 |
| リモート・プログラムの割り込み [リモート・プログラムのわりこみ] .....       | 114 | アクティブ・ターゲット .....                      | 105 |
| リモート・ターゲットの割り込み [リモート・ターゲットのわりこみ] .....       | 112 | 保存、シンボル・テーブルの [ぼぞん、シンボル・テーブルの] .....   | 100 |
| リモート・シリアル・スタブ .....                           | 111 | 書き込み、実行コードへの [かきこみ、じっこうコードへの] .....    | 97  |
| リモート・シリアル・スタブ、メイン・ルーチン .....                  | 111 | 削除、ブレイクポイントの [さくじょ、ブレイクポイントの] .....    | 36  |
| 初期化、リモート・シリアル・スタブの [しよきか、リモート・シリアル・スタブの] ...  | 111 | 読み込み、シンボルの即時 [よみこみ、シンボルのそくじ] .....     | 100 |
| エラー、正当な入力にたいする [エラー、せいとうなにゅうりょくにたいする] .....   | 145 |                                        |     |
| ブレイクポイント、条件付きの [ブレイクポイント、じょうけんつきの] .....      | 37  |                                        |     |
| スタブを使わないリモート接続 [スタブをつかわないリモートせつぞく] .....      | 115 |                                        |     |
| 引数、ユーザ・プログラムへの [ひきすう、ユーザ・プログラムへの] .....       | 20  |                                        |     |
| パケット、標準出力への報告 [パケット、ひょうじゅんしゅつりょくへのほうこく] ..... | 115 |                                        |     |
| 条件付きのブレイクポイント [じょうけんつきのブレイクポイント] .....        | 37  |                                        |     |

- シンボルの即時読み込み [シンボルのそくじよみ  
こみ] ..... 100  
シンボル・テーブル ..... 99  
コア・ダンプ・ファイル ..... 99
- 書き込み、コア・ファイルへの [かきこみ、コ  
ア・ファイルへの] ..... 97
- コア・ファイルへの書き込み [コア・ファイルへ  
のかきこみ] ..... 97  
コロンの、スコープ演算子の 2 重 [コロンの、スコ  
プえんざんしの 2 じゅう] ..... 88
- 部分的シンボル・ダンプ [ぶぶんてきシンボル・  
ダンプ] ..... 93
- 実行コードへの書き込み [じっこうコードへのか  
きこみ] ..... 97
- シリアル・プロトコル、GDB リモート ..... 114
- 浮動小数ハードウェア [ふどうしょうすうハード  
ウェア] ..... 74
- 再ロード、シンボルの [さいロード、シンボルの]  
..... 92
- レジスタ・スタック、AMD29K の ..... 73  
オブジェクトの形式と C++ [オブジェクトのけい  
しきと C++] ..... 82
- 浮動小数点レジスタ [ふどうしょうすうてんレジ  
スタ] ..... 72
- 併用、ターゲットの [へいよう、ターゲットの]  
..... 105
- レジスタ ..... 72  
リダイレクト ..... 22  
ユーザ定義コマンド [ユーザていぎコマンド]  
..... 137  
マルチスレッド ..... 24  
プロセスのイメージ ..... 24  
バクトレース ..... 48  
デマングル ..... 69  
ディレクトリ、ソース・ファイルの ..... 55  
ディレクトリ、コンパイルする ..... 55  
シンボリック形式のアドレス解釈 [シンボリック  
けいしきのアドレスかいしゃく] ..... 67  
フレームを持たない関数の実行 [フレームをもた  
ないかんすうのじっこう] ..... 47
- 参照する位置、ポインタの [さんしょうするい  
ち、ポインタの] ..... 67
- 命令の表示、アセンブリ [めいれいのひょうじ、  
アセンブリ] ..... 56

|                                          |     |                                       |    |
|------------------------------------------|-----|---------------------------------------|----|
| ポインタの参照する位置 [ポインタのさんしょうするいち] .....       | 67  | デバッグ、最適化コードの [デバッグ、さいてきかコードの] .....   | 19 |
| フレーム・ポインタ .....                          | 47  | システムのスレッド識別子 [システムのスレッドしきべつし] .....   | 25 |
| ダウンロード、日立 SH への [ダウンロード、ひたち SH への] ..... | 109 | 自動的選択、スレッドの [じどうてきせんたく、スレッドの] .....   | 26 |
| ソースのパス .....                             | 55  | 最適化コードのデバッグ [さいてきかコードのデバッグ] .....     | 19 |
| スレッドのブレイクポイント .....                      | 45  | ウォッチポイントの設定 [ウォッチポイントのせってい] .....     | 33 |
| スタック・フレーム .....                          | 47  | 最後のブレイクポイント [さいごのブレイクポイント] .....      | 30 |
| コンパイルするディレクトリ .....                      | 55  | メモリのトレース .....                        | 29 |
| シミュレータ、日立 SH [シミュレータ、ひたち SH] .....       | 128 | スレッドの自動的選択 [スレッドのじどうてきせんたく] .....     | 26 |
| キャッチポイント .....                           | 34  | カレント・スレッド .....                       | 25 |
| キャストしたメモリ .....                          | 59  | アセンブリ命令の表示 [アセンブリめいれいのひょうじ] .....     | 56 |
| カレントなディレクトリ .....                        | 55  | アセンブリ言語の選択 [アセンブリげんごのせんたく] .....      | 57 |
| ウォッチポイントとスレッド .....                      | 34  | ディレクトリ、現在の [ディレクトリ、げんざいの] .....       | 55 |
| 内部のブレイクポイント番号 [ないぶのブレイクポイントばんごう] .....   | 32  | エンコードされた形式 [エンコードされたけいしき] .....       | 69 |
| ユーザ・プログラムへの引数 [ユーザ・プログラムへのひきすう] .....    | 20  | パイプ .....                             | 20 |
| 負のブレイクポイント番号 [ふのブレイクポイントばんごう] .....      | 32  | スレッド識別子、システムの [スレッドしきべつし、システムの] ..... | 25 |
| 設定、ウォッチポイントの [せってい、ウォッチポイントの] .....      | 33  | 繰り返し、コマンドの [くりかえし、コマンドの] .....        | 13 |
| 一時停止、出力の [いちじていし、しゅつりょくの] .....          | 133 |                                       |    |

|                                    |    |                                   |     |
|------------------------------------|----|-----------------------------------|-----|
| 切り替え、スレッドの [きりかえ、スレッドの]<br>.....   | 24 | コンビニエンス変数 [コンビニエンスへんすう]<br>.....  | 71  |
|                                    |    | スレッド、停止した [スレッド、ていしした]..          | 45  |
|                                    |    | コマンドの繰り返し [コマンドのくりかえし]<br>.....   | 13  |
| シェル・エスケープ .....                    | 12 | スレッドの切り替え [スレッドのきりかえ]...          | 24  |
| 表示、マシン命令の [ひょうじ、マシンめいれいの]<br>..... | 56 | 機械語命令の表示 [きかいごめいれいのひょうじ]<br>..... | 56  |
|                                    |    | 機械語命令の選択 [きかいごめいれいのせんたく]<br>..... | 57  |
| 選択、マシン命令の [せんたく、マシンめいれいの]<br>..... | 57 |                                   |     |
|                                    |    | コマンド行の編集 [コマンドぎょうのへんしゅう]<br>..... | 149 |
| フォーマット、出力 [フォーマット、しゅつりょく]<br>..... | 62 |                                   |     |
|                                    |    | 出力フォーマット [しゅつりょくフォーマット]<br>.....  | 62  |
| 変数値、正しくない [へんすうち、ただしくない]<br>.....  | 60 |                                   |     |
|                                    |    | マシン命令の表示 [マシンめいれいのひょうじ]<br>.....  | 56  |
| 引用、コマンド内の [いんよう、コマンドないの]<br>.....  | 14 | マシン命令の選択 [マシンめいれいのせんたく]<br>.....  | 57  |
|                                    |    |                                   |     |
| メモリ、型変換した [メモリ、かたへんかんした]<br>.....  | 59 | 対象、デバッグの [たいしょう、デバッグの]..          | 25  |
|                                    |    |                                   |     |
| 選択されたフレーム [せんたくされたフレーム]<br>.....   | 47 | 最上位のフレーム [さいじょういのフレーム]<br>.....   | 47  |
|                                    |    |                                   |     |
| 現在のディレクトリ [げんざいのディレクトリ]<br>.....   | 55 | 型変換したメモリ [かたへんかんしたメモリ]<br>.....   | 59  |

|                                 |     |                                                   |     |
|---------------------------------|-----|---------------------------------------------------|-----|
| コマンド内の引用 [コマンドないのいんよう]<br>..... | 14  | マルチプロセス .....                                     | 26  |
| 初期化ファイル名 [しょきかファイルめい] ..        | 139 | ブレイクポイント .....                                    | 29  |
| 作業ディレクトリ [さぎょうディレクトリ] ...       | 55  | キャッチポイント .....                                    | 29  |
| 再開、スレッドの [さいかい、スレッドの] ...       | 45  | オーバーロード、C++での .....                               | 84  |
|                                 |     | シンボルの表現、C++の [シンボルのひょうげん、<br>C++の] .....          | 69  |
|                                 |     | メモリ・モデル、H8/500 .....                              | 126 |
|                                 |     | 初期化ファイル、readline[しょきかファイル、<br>readline] .....     | 152 |
| ハンドラ、例外の [ハンドラ、れいがいの] ...       | 50  | キル・リング .....                                      | 150 |
| 名前、シンボルの [なまえ、シンボルの] .....      | 91  | テキストの再挿入 ( yank ) [テキストのさいそう<br>にゅう] .....        | 150 |
| 履歴の大きさ [履歴のおおきさ] .....          | 132 | テキストのキル ( kill ) .....                            | 150 |
| トレースバック .....                   | 48  | オーバーロード .....                                     | 40  |
| スレッド、実行の [スレッド、じっこうの] ...       | 24  | ブレイクポイント、メモリ・アドレスの [ブレイ<br>クポイント、メモリ・アドレスの] ..... | 29  |
| キャッチ、例外の [キャッチ、れいがいの] ...       | 50  | メモリ・アドレスのブレイクポイント [メモリ・<br>アドレスのブレイクポイント] .....   | 29  |
|                                 |     | イベントに対するブレイクポイント [イベントに<br>たいするブレイクポイント] .....    | 29  |
|                                 |     | ブレイクポイント、変数変化の [ブレイクポイン<br>ト、へんすうへんかの] .....      | 29  |
| 停止したスレッド [ていししたスレッド] .....      | 45  | 変数変化のブレイクポイント [へんすうへんかの<br>ブレイクポイント] .....        | 29  |
| 処理、シグナルの [しょり、シグナルの] .....      | 44  | 番号、ブレイクポイント [ばんごう、ブレイクポ<br>イント] .....             | 29  |
| 最下位のフレーム [さいかいのフレーム] .....      | 47  | ブレイクポイント番号 [ブレイクポイントばんご<br>う] .....               | 29  |
| 呼び出しスタック [よびだしスタック] .....       | 47  | ウォッチポイント .....                                    | 29  |
|                                 |     | スレッド識別子、GDB の [スレッドしきべつし、<br>GDB の] .....         | 25  |

|                                 |                                                     |
|---------------------------------|-----------------------------------------------------|
| 変更、変数値の [へんこう、へんすうちの] ... 95    | フレームの番号 [フレームのばんごう] ..... 47                        |
|                                 | スレッドの再開 [スレッドのさいかい] ..... 45                        |
|                                 | スコープ演算子 [スコープえんざんし] ..... 88                        |
| 設定、変数値の [せってい、へんすうちの] ... 95    |                                                     |
|                                 | 初期化ファイル [しょきがファイル] ..... 139                        |
| 出力、データの [しゅつりょく、データの] ... 59    |                                                     |
|                                 | 検査、データの [けんさ、データの] ..... 59                         |
| 更新、変数値の [こうしん、へんすうちの] ... 95    |                                                     |
|                                 | 履歴の記録 [履歴のきろく] ..... 132                            |
|                                 | チェック、範囲 [チェック、はんい] ..... 79                         |
| 共有ライブラリ [きょうゆうライブラリ] .... 102   | シンボルの名前 [シンボルのなまえ] ..... 91                         |
|                                 | イベント指定子 [イベントしていし] ..... 167                        |
| デバッグの対象 [デバッグのたいしょう] .... 25    | 未知のアドレス [みちのアドレス] ..... 62                          |
| 例外のハンドラ [れいがいのハンドラ] ..... 50    | チェックサム、GDB リモートの ..... 114                          |
| 例外のキャッチ [れいがいのキャッチ] ..... 50    | シミュレータ、H8/300 または H8/500 ..... 128                  |
|                                 | ダウンロード、H8/300 や H8/500 への ..... 109                 |
| 表示、データの [ひょうじ、データの] ..... 59    |                                                     |
|                                 | 組み込み機能、Modula-2 の [くみこみきのう、<br>Modula-2 の] ..... 86 |
| 調査、メモリの [ちょうさ、メモリの] ..... 63    |                                                     |
|                                 | ダウンロード、Nindy-960 への ..... 109                       |
|                                 | シミュレータ、Z8000 ..... 128                              |
| 致命的シグナル [ちめいてきシグナル] ... 44, 145 |                                                     |
|                                 | 値履歴と\$_や\$__[あたい履歴と\$_や\$__] ..... 64               |
| 多重ターゲット [たじゅうターゲット] ..... 105   |                                                     |

|                                  |                                                        |
|----------------------------------|--------------------------------------------------------|
| 変数名の衝突 [へんすうめいのしょうとつ] ... 60     | 範囲チェック [はんいチェック] ..... 79                              |
| 整形した出力 [せいけいしたしゅつりょく] ... 62     | 初期フレーム [しょきフレーム] ..... 47                              |
| 変数への代入 [へんすうへのだいにゅう] ..... 95    | ヒストリ置換 [ヒストリちかん] ..... 132                             |
|                                  | データの検査 [データのけんさ] ..... 59                              |
|                                  | サイズ、画面 [サイズ、がめん] ..... 133                             |
|                                  | プロトコル、GDB リモート・シリアル ..... 114                          |
| 入力、数値の [にゅうりょく、すうちの] .... 133    |                                                        |
| 停止、出力の [ていし、しゅつりょくの] .... 133    | 浮動小数点、MIPS リモートの [ふどうしょうす<br>うてん、MIPS リモートの] ..... 127 |
|                                  | 不正な入力 [ふせいなにゅうりょく] ..... 145                           |
| 正しくない値 [ただしくないあたい] ..... 60      | 慎重な動作 [しんちょうなどうさ] ..... 135                            |
| コマンド編集 [コマンドへんしゅう] .... 131, 149 | 命令セット [めいれいセット] ..... 57                               |
| 馬鹿げた質問 [ばかげたしつもん] ..... 135      | 動的リンク [どうてきリンク] ..... 101                              |
| 実行ファイル [じっこうファイル] ..... 99       | リロード ..... 92                                          |
|                                  | リターン ..... 97                                          |
| 逆アセンブル [ぎゃくアセンブル] ..... 56       | パッチ、バイナリの ..... 97                                     |
|                                  | バイナリのパッチ ..... 97                                      |
|                                  | シンボル・ダンプ ..... 93                                      |
| 遠隔デバッグ [えんかくデバッグ] ..... 110      | 復帰、関数からの [ふっき、かんすうからの] .. 97                           |
| ヒストリ番号 [ヒストリばんごう] ..... 70       | 呼び出し、関数の [よびだし、かんすうの] ... 97                           |
| ヒストリ展開 [ヒストリてんかい] ..... 132      |                                                        |



|                                     |     |                                                 |     |
|-------------------------------------|-----|-------------------------------------------------|-----|
| 戻る、関数から [もどる、かんすうから] .....          | 97  | バグ報告、GDB の [バグほうこく、GDB の] .....                 | 145 |
| 返る、関数から [かえる、かんすうから] .....          | 97  | 応答時間、MIPS デバッグの [おうとうじかん、MIPS デバッグの] .....      | 51  |
| 関数の呼び出し [かんすうのよびだし] .....           | 97  | スタック、MIPS の .....                               | 51  |
| デフォルト、C/C++ の .....                 | 83  | チェック、Modula-2 の .....                           | 88  |
| デフォルト、Modula-2 の .....              | 88  | 差異、標準 Modula-2 との [さい、ひょうじゅん Modula-2 との] ..... | 88  |
| 値ヒストリ [あたひヒストリ] .....               | 70  | シグナル .....                                      | 43  |
| メンバ関数 [メンバかんすう] .....               | 82  | 実行のスレッド [じっこうのスレッド] .....                       | 24  |
| バグの報告 [バグのほうこく] .....               | 145 | プロセス、多重 [プロセス、たじゅう] .....                       | 26  |
| バグの基準 [バグのきじゅん] .....               | 145 | バージョン番号 [バージョンばんごう] .....                       | 16  |
| 型チェック [かたチェック] .....                | 78  | シグナルの処理 [シグナルのしより] .....                        | 44  |
| 画面サイズ [がめんサイズ] .....                | 133 | 発生、例外の [はっせい、れいがいの] .....                       | 35  |
| スタック、Alpha の .....                  | 51  | 例外ハンドラ [れいがいハンドラ] .....                         | 34  |
| チェック、C や C++ の .....                | 83  | 多重スレッド [たじゅうスレッド] .....                         | 24  |
| コマンド、C++ 専用の [コマンド、C++ せんようの] ..... | 84  |                                                 |     |
| 名前空間、C++ の [なまえくうかん、C++ の] ..       | 82  |                                                 |     |

|                                                                    |     |                                  |    |
|--------------------------------------------------------------------|-----|----------------------------------|----|
| ステップ実行 [ステップじっこう].....                                             | 41  | 引用文字列の補完 [いんようもじれつのほかん]<br>..... | 14 |
| 通過カウント [つうかカウント].....                                              | 38  | オンライン文書 [オンラインぶんしょ] .....        | 15 |
| 例外の発生 [れいがいはっせい].....                                              | 35  | 説明文字列 [せつめいもじれつ] .....           | 15 |
| 実行の再開 [じっこうのさいかい].....                                             | 41  | ヘルプ情報 [ヘルプじょうほう] .....           | 15 |
| コマンド、STDBUG ( ST2000 ) に対する [コマン<br>ド、STDBUG ( ST2000 ) にたいする] ... | 121 | 単語の補完 [たんごのほかん] .....            | 13 |
| クオート .....                                                         | 14  | コメント .....                       | 13 |
| アタッチ .....                                                         | 23  | 呼び出し、makeの [よびだし、makeの] .....    | 12 |
| 作業ディレクトリ、ユーザ・プログラムの [さ<br>ぎょうディレクトリ、ユーザ・プログラムへ<br>の] .....         | 22  | スタック/トレース.....                   | 48 |
| ユーザ・プログラムの作業ディレクトリ [ユーザ・<br>プログラムのさぎょうディレクトリ] ....                 | 22  | 参照宣言 [さんしょうせんげん] .....           | 83 |
| 環境、ユーザ・プログラムの [かんきょう、ユー<br>ザ・プログラムの] .....                         | 21  | 表示設定 [ひょうじせってい] .....            | 65 |
| ユーザ・プログラムの環境 [ユーザ・プログラム<br>のかんきょう] .....                           | 21  | 正規表現 [せいきひょうげん] .....            | 31 |
| 補完、引用文字列の [ほかん、いんようもじれつ<br>の] .....                                | 14  | 制御端末 [せいぎょたんまつ] .....            | 22 |

- 数値表現 [すうちひょうげん] ..... 133  
人工配列 [じんこうはいれつ] ..... 61
- 継続実行 [けいぞくじっこう] ..... 41
- 例外処理 [れいがいしより] ..... 34
- 式の表示 [しきのひょうじ] ..... 64  
自動表示 [じどうひょうじ] ..... 64
- 作業言語 [さぎょうげんご] ..... 75
- 割り込み [わりこみ] ..... 12
- 演算子、C や C++ の [えんざんし、C や C++ の]  
..... 80
- 型変換、C++ での [かたへんかん、C++ での] .. 83
- 識別子、GDB のスレッド [しきべつし、GDB の  
スレッド] ..... 25
- 演算子、Modula-2 の [えんざんし、Modula-2 の]  
..... 85
- 表記法、readline [ひょうきほう、readline] ... 149
- 入出力 [にゅうしゅつりょく] ..... 22
- 省略形 [しょうりゃくけい] ..... 13
- 短縮形 [たんしゅくけい] ..... 13
- 行指定 [ぎょうしてい] ..... 53
- 富士通 [ふじつう] ..... 111
- 定数、C/C++ の [ていすう、C や C++ の] ..... 82
- 報告、GDB のバグ [ほうこく、GDB のバグ]  
..... 145
- 終了、GDB の [しゅうりょう、GDB の] ..... 12
- 設定、GDB の [せってい、GDB の] ..... 171
- 定数、Modula-2 の [ていすう、Modula-2 の] .. 87
- 操作、readline の [操作、readline の] ..... 149
- 略称 [りゃくしょう] ..... 13

|                  |     |                                            |     |
|------------------|-----|--------------------------------------------|-----|
| 編集 [へんしゅう] ..... | 131 | 起動 [きどう] .....                             | 19  |
| 代入 [だいにゅう] ..... | 95  | 開始 [かいし] .....                             | 19  |
| 継承 [けいしょう] ..... | 84  | 式、C や C++ の [しき、C や C++ の] .....           | 80  |
| 展開 [てんかい] .....  | 167 | 式、C++ の [しき、C++ の] .....                   | 82  |
| 端末 [たんまつ] .....  | 22  | 式、Modula-2 での [しき、Modula-2 での] ....        | 85  |
| 実行 [じっこう] .....  | 19  | 日立 SH へのダウンロード [ひたち SH へのダウ<br>ンロード] ..... | 109 |
| 検索 [けんさく] .....  | 54  | 日立 SH シミュレータ [ひたち SH シミュレータ]<br>.....      | 128 |
| 確認 [かくにん] .....  | 135 | 式 [しき] .....                               | 59  |
| 引用 [いんよう] .....  | 91  | # .....                                    | 13  |
| 補完 [ほかん] .....   | 13  | # in Modula-2 .....                        | 89  |
| 日立 [ひたち] .....   | 111 | \$ .....                                   | 70  |
| 言語 [げんご] .....   | 75  | \$ .....                                   | 70  |
|                  |     | \$ .....                                   | 72  |
|                  |     | \$ _ and info breakpoints .....            | 32  |
|                  |     | \$ _ and info line .....                   | 56  |
|                  |     | \$ _ , \$ _ _ , and value history .....    | 64  |
|                  |     | \$ _ _ .....                               | 72  |
|                  |     | \$ _ exitcode .....                        | 72  |
|                  |     | \$ bpnun .....                             | 30  |
|                  |     | \$ cdir .....                              | 55  |
|                  |     | \$ cwd .....                               | 55  |

. ..... 88  
 .esgdbinit ..... 139  
 '.gdbinit' ..... 139  
 .os68gdbinit ..... 139  
 .vxgdbinit ..... 139

/  
 /proc ..... 24

:  
 :: ..... 60, 88

@  
 @ ..... 61

{  
 {type} ..... 59

**2**  
 2 つのコロン [2 つのコロン] ..... 60  
 2 重コロン [2 じゅうコロン] ..... 60  
 29K プログラムの実行 [29K プログラムのじっ  
 こ] ..... 119

**A**  
 a.out and C++ ..... 82  
 abbreviation ..... 13  
 active targets ..... 105  
 add-shared-symbol-file ..... 101  
 add-symbol-file ..... 101  
 Alpha のスタック ..... 51  
 Alpha stack ..... 51  
 AMD 29K register stack ..... 73  
 AMD EB29K ..... 107  
 AMD29K via UDI ..... 119  
 arguments (to your program) ..... 20  
 artificial array ..... 61  
 assembly instructions ..... 56, 57  
 assignment ..... 95  
 attach ..... 23

automatic display ..... 64  
 automatic thread selection ..... 26  
 awatch ..... 33

## B

b ..... 30  
 backtrace ..... 48  
 backtraces ..... 48  
 bell-style ..... 152  
 break ..... 30  
 break ... thread threadno ..... 45  
 break in overloaded functions ..... 84  
 breakpoint サブルーチン、リモート ..... 111  
 breakpoint commands ..... 39  
 breakpoint conditions ..... 37  
 breakpoint numbers ..... 29  
 breakpoint on events ..... 29  
 breakpoint on memory address ..... 29  
 breakpoint on variable modification ..... 29  
 breakpoint subroutine, remote ..... 111  
 breakpoints ..... 29  
 breakpoints and threads ..... 45  
 bt ..... 48  
 bug criteria ..... 145  
 bug reports ..... 145  
 bugs in GDB ..... 145

## C

c ..... 41  
 C and C++ ..... 80  
 C and C++ checks ..... 83  
 C and C++ constants ..... 82  
 C and C++ defaults ..... 83  
 C and C++ operators ..... 80  
 C++ ..... 80  
 C++ とオブジェクトの形式 [C++ とオブジェクト  
 のけいしき] ..... 82  
 C++ のシンボルの表現 [C++ のシンボルのひょう  
 げん] ..... 69  
 C++ のスコープ解決 [C++ のスコープかいけつ]  
 ..... 60  
 C++ のシンボル表示 [C++ のシンボルひょうじ]  
 ..... 84  
 C++ のサポート、COFF でない ..... 82  
 C++ の例外処理 [C++ のれいがいしより] ..... 84  
 C++ と DWARF ..... 82

|                                                   |        |                                                |     |
|---------------------------------------------------|--------|------------------------------------------------|-----|
| C++と ECOFF .....                                  | 82     | completion .....                               | 13  |
| C++と ELF .....                                    | 82     | completion of quoted strings .....             | 14  |
| C++と XCOFF .....                                  | 82     | completion-query-items .....                   | 152 |
| C++ and object formats .....                      | 82     | condition .....                                | 38  |
| C++ exception handling .....                      | 84     | conditional breakpoints .....                  | 37  |
| C++ scope resolution .....                        | 60     | configuring GDB .....                          | 171 |
| C++ support, not in COFF .....                    | 82     | confirmation .....                             | 135 |
| C++ symbol decoding style .....                   | 69     | connect ( STDBUG に対する ) [connect               |     |
| C++ symbol display .....                          | 84     | ( STDBUG にたいする )] .....                        | 122 |
| call .....                                        | 97     | connect (to STDBUG) .....                      | 122 |
| call overloaded functions .....                   | 83     | continue .....                                 | 41  |
| call stack .....                                  | 47     | continuing .....                               | 41  |
| calling functions .....                           | 97     | continuing threads .....                       | 45  |
| calling make .....                                | 12     | control C, and remote debugging .....          | 112 |
| casts, to view memory .....                       | 59     | controlling terminal .....                     | 22  |
| catch .....                                       | 34     | convenience variables .....                    | 71  |
| catch catch .....                                 | 34     | convert-meta .....                             | 153 |
| catch exceptions .....                            | 50     | core .....                                     | 101 |
| catch exec .....                                  | 34     | core dump file .....                           | 99  |
| catch fork .....                                  | 35     | core-file .....                                | 101 |
| catch load .....                                  | 35     | CPU シミュレータ .....                               | 128 |
| catch throw .....                                 | 34     | CPU simulator .....                            | 128 |
| catch unload .....                                | 35     | crash of debugger .....                        | 145 |
| catch vfork .....                                 | 35     | Ctrl-C、リモート・デバッグ処理 [Ctrl-C、リ                   |     |
| catchpoints .....                                 | 29, 34 | モート・デバッグしよ]                                    | 112 |
| cd .....                                          | 22     | current directory .....                        | 55  |
| cdir .....                                        | 55     | current thread .....                           | 25  |
| checks, range .....                               | 79     | cwd .....                                      | 55  |
| checks, type .....                                | 78     |                                                |     |
| checksum, for GDB remote .....                    | 114    | <b>D</b>                                       |     |
| choosing target byte order .....                  | 109    | d .....                                        | 36  |
| clear .....                                       | 36     | debugger crash .....                           | 145 |
| clearing breakpoints, watchpoints, catchpoints .. | 36     | debugging optimized code .....                 | 19  |
| COFF versus C++ .....                             | 82     | debugging stub, example .....                  | 114 |
| colon, doubled as scope operator .....            | 88     | debugging target .....                         | 105 |
| colon-colon .....                                 | 60     | define .....                                   | 137 |
| command editing .....                             | 149    | delete .....                                   | 36  |
| command files .....                               | 139    | delete breakpoints .....                       | 36  |
| command hooks .....                               | 138    | delete display .....                           | 65  |
| command line editing .....                        | 131    | deleting breakpoints, watchpoints, catchpoints |     |
| commands .....                                    | 39     | .....                                          | 36  |
| commands for C++ .....                            | 84     | demangling .....                               | 69  |
| commands to STDBUG (ST2000) .....                 | 121    | detach .....                                   | 23  |
| comment .....                                     | 13     | device .....                                   | 125 |
| comment-begin .....                               | 152    | dir .....                                      | 55  |
| compilation directory .....                       | 55     | directories for source files .....             | 55  |
| Compiling .....                                   | 124    | directory .....                                | 55  |
| complete .....                                    | 16     |                                                |     |

directory, compilation ..... 55  
 directory, current ..... 55  
 dis ..... 37  
 disable ..... 37  
 disable breakpoints ..... 36, 37  
 disable display ..... 65  
 disable-completion ..... 153  
 disassemble ..... 56  
 display ..... 64  
 display of expressions ..... 64  
 do ..... 49  
 document ..... 137  
 documentation ..... 169  
 down ..... 49  
 down-silently ..... 50  
 download to H8/300 or H8/500 ..... 109  
 download to Hitachi SH ..... 109  
 download to Nindy-960 ..... 109  
 download to Sparclet ..... 124  
 download to VxWorks ..... 123  
 dynamic linking ..... 101

## E

eb.log ..... 121  
 EB29K 用のログ・ファイル [EB29K ようのログ・  
 ファイル] ..... 121  
 EB29K ボード ..... 119  
 EB29K board ..... 119  
 EBMON ..... 120  
 echo ..... 139  
 ECOFF and C++ ..... 82  
 editing ..... 131  
 editing command lines ..... 149  
 editing-mode ..... 153  
 ELF/DWARF and C++ ..... 82  
 ELF/stabs and C++ ..... 82  
 else ..... 137  
 Emacs ..... 141  
 enable ..... 37  
 enable breakpoints ..... 36, 37  
 enable display ..... 65  
 enable-keypad ..... 153  
 end ..... 39  
 entering numbers ..... 133  
 environment (of your program) ..... 21  
 error on valid input ..... 145  
 event designators ..... 167

event handling ..... 34  
 examining data ..... 59  
 examining memory ..... 63  
 exception handlers ..... 34, 50  
 exceptionHandler ..... 112  
 exec-file ..... 99  
 executable file ..... 99  
 exiting GDB ..... 12  
 expand-tilde ..... 153  
 expansion ..... 167  
 expressions ..... 59  
 expressions in C or C++ ..... 80  
 expressions in C++ ..... 82  
 expressions in Modula-2 ..... 85

## F

f ..... 49  
 fatal signal ..... 145  
 fatal signals ..... 44  
 fg ..... 41  
 file ..... 99  
 finish ..... 42  
 flinching ..... 135  
 floating point ..... 74  
 floating point registers ..... 72  
 floating point, MIPS remote ..... 127  
 flush\_i\_cache ..... 113  
 focus of debugging ..... 25  
 foo ..... 103  
 forkを呼び出す関数のデバッグ [fork をよびだす  
 かんすうのデバッグ] ..... 26  
 fork, debugging programs which call ..... 26  
 format options ..... 65  
 formatted output ..... 62  
 Fortran ..... 1  
 forward-search ..... 54  
 frame ..... 47  
 frame ..... 48, 49  
 frame number ..... 47  
 frame pointer ..... 47  
 frameless execution ..... 47  
 Fujitsu ..... 111

## G

|                                       |     |
|---------------------------------------|-----|
| g++ .....                             | 80  |
| GDB リファレンス・カード [GDB リファレンス・カード] ..... | 169 |
| GDB のスレッド識別子 [GDB のスレッドしきべつ] .....    | 25  |
| GDB のバグの報告 [GDB のバグのほうこく] .....       | 145 |
| GDB の終了 [GDB のしゅうりょう] .....           | 12  |
| GDB bugs, reporting .....             | 145 |
| GDB reference card .....              | 169 |
| GDBHISTFILE .....                     | 132 |
| gdbserve.nlm .....                    | 117 |
| gdbserver .....                       | 115 |
| getDebugChar .....                    | 112 |
| GNU C++ .....                         | 80  |
| GNU Emacs .....                       | 141 |

## H

|                                         |     |
|-----------------------------------------|-----|
| h .....                                 | 15  |
| H8/300 または H8/500 のシミュレータ .....         | 128 |
| H8/300 や H8/500 へのダウンロード .....          | 109 |
| H8/300 or H8/500 download .....         | 109 |
| H8/300 or H8/500 simulator .....        | 128 |
| handle .....                            | 44  |
| handle_exception .....                  | 111 |
| handling signals .....                  | 44  |
| hardware watchpoints .....              | 33  |
| hbreak .....                            | 31  |
| help .....                              | 15  |
| help target .....                       | 105 |
| help user-defined .....                 | 137 |
| heuristic-fence-post (Alpha,MIPS) ..... | 51  |
| history expansion .....                 | 132 |
| history file .....                      | 132 |
| history number .....                    | 70  |
| history save .....                      | 132 |
| history size .....                      | 132 |
| history substitution .....              | 132 |
| Hitachi .....                           | 111 |
| Hitachi SH download .....               | 109 |
| Hitachi SH simulator .....              | 128 |
| horizontal-scroll-mode .....            | 153 |

## I

|                                    |        |
|------------------------------------|--------|
| i .....                            | 16     |
| i/o .....                          | 22     |
| i386 .....                         | 111    |
| i386-stub.c .....                  | 111    |
| i960 .....                         | 117    |
| if .....                           | 137    |
| ignore .....                       | 38     |
| ignore count (of breakpoint) ..... | 38     |
| INCLUDE_RDB .....                  | 122    |
| info .....                         | 16     |
| info address .....                 | 91     |
| info all-registers .....           | 72     |
| info args .....                    | 50     |
| info breakpoints .....             | 32     |
| info catch .....                   | 50     |
| info display .....                 | 65     |
| info extensions .....              | 77     |
| info f .....                       | 50     |
| info files .....                   | 102    |
| info float .....                   | 74     |
| info frame .....                   | 50, 77 |
| info functions .....               | 92     |
| info line .....                    | 56     |
| info locals .....                  | 50     |
| info proc .....                    | 24     |
| info proc id .....                 | 24     |
| info proc mappings .....           | 24     |
| info proc status .....             | 24     |
| info proc times .....              | 24     |
| info program .....                 | 29     |
| info registers .....               | 72     |
| info s .....                       | 48     |
| info set .....                     | 16     |
| info share .....                   | 102    |
| info sharedlibrary .....           | 102    |
| info signals .....                 | 44     |
| info source .....                  | 77, 92 |
| info sources .....                 | 92     |
| info stack .....                   | 48     |
| info target .....                  | 102    |
| info terminal .....                | 22     |
| info threads .....                 | 25     |
| info types .....                   | 92     |
| info variables .....               | 92     |
| info watchpoints .....             | 33     |
| inheritance .....                  | 84     |



init file ..... 139  
 init file name ..... 139  
 initial frame ..... 47  
 initialization file, readline ..... 152  
 innermost frame ..... 47  
**input-meta** ..... 154  
**inspect** ..... 59  
 installation ..... 171  
 instructions, assembly ..... 56, 57  
 Intel ..... 111  
 interaction, readline ..... 149  
 internal GDB breakpoints ..... 32  
 interrupt ..... 12  
 interrupting remote programs ..... 114  
 interrupting remote targets ..... 112  
 invalid input ..... 145

## J

**jump** ..... 96

## K

**keymap** ..... 153  
**kill** ..... 23  
 kill ring ..... 150  
 killing text ..... 150

## L

**l** ..... 53  
 languages ..... 75  
 latest breakpoint ..... 30  
 leaving GDB ..... 12  
 linespec ..... 53  
**list** ..... 53  
 listing machine instructions ..... 56, 57  
**load filename** ..... 109  
 log file for EB29K ..... 121

## M

m680x0 ..... 111  
**m68k-stub.c** ..... 111  
 machine instructions ..... 56, 57  
**maint info breakpoints** ..... 33  
**maint print psymbols** ..... 93  
**maint print symbols** ..... 93

**make** ..... 12  
**makeの呼び出し** [make のよびだし] ..... 12  
**mapped** ..... 100  
**mark-modified-lines** ..... 153  
 member functions ..... 82  
 memory models, H8/500 ..... 126  
 memory tracing ..... 29  
 memory, viewing as typed object ..... 59  
 memory-mapped symbol file ..... 100  
**memset** ..... 113  
**meta-flag** ..... 154  
 MIPS リモート浮動小数点 [MIPS リモートふど  
 うしょうすうてん] ..... 127  
 MIPS ボード ..... 126  
 MIPS のスタック ..... 51  
 MIPS boards ..... 126  
 MIPS remote floating point ..... 127  
 MIPS **remotedebug** プロトコル ..... 128  
 MIPS **remotedebug** protocol ..... 128  
 MIPS stack ..... 51  
 Modula-2 ..... 85  
 Modula-2 の# ..... 89  
 Modula-2 built-ins ..... 86  
 Modula-2 checks ..... 88  
 Modula-2 constants ..... 87  
 Modula-2 defaults ..... 88  
 Modula-2 operators ..... 85  
 Modula-2, deviations from ..... 88  
 Motorola 680x0 ..... 111  
 multiple processes ..... 26  
 multiple targets ..... 105  
 multiple threads ..... 24

## N

**n** ..... 42  
 names of symbols ..... 91  
 namespace in C++ ..... 82  
 negative breakpoint numbers ..... 32  
**New systag** ..... 25  
**next** ..... 42  
**nexti** ..... 43  
**ni** ..... 43  
 Nindy ..... 117  
 Nindy-960 へのダウンロード ..... 109  
 notation, readline ..... 149  
 number representation ..... 133  
 numbers for breakpoints ..... 29

## O

|                                 |     |
|---------------------------------|-----|
| object formats and C++ .....    | 82  |
| online documentation .....      | 15  |
| optimized code, debugging ..... | 19  |
| outermost frame .....           | 47  |
| <b>output</b> .....             | 140 |
| output formats .....            | 62  |
| <b>output-meta</b> .....        | 154 |
| overloading .....               | 40  |
| overloading in C++ .....        | 84  |

## P

|                                    |     |
|------------------------------------|-----|
| packets, reporting on stdout ..... | 115 |
| partial symbol dump .....          | 93  |
| patching binaries .....            | 97  |
| <b>path</b> .....                  | 21  |
| pauses in output .....             | 133 |
| pipes .....                        | 20  |
| pointer, finding referent .....    | 67  |
| <b>print</b> .....                 | 59  |
| print settings .....               | 65  |
| <b>printf</b> .....                | 140 |
| printing data .....                | 59  |
| process image .....                | 24  |
| processes, multiple .....          | 26  |
| prompt .....                       | 131 |
| protocol, GDB remote serial .....  | 114 |
| <b>ptype</b> .....                 | 91  |
| <b>putDebugChar</b> .....          | 112 |
| <b>pwd</b> .....                   | 22  |

## Q

|                                         |    |
|-----------------------------------------|----|
| <b>q</b> .....                          | 12 |
| <b>quit</b> [ <i>expression</i> ] ..... | 12 |
| quotes in commands .....                | 14 |
| quoting names .....                     | 91 |

## R

|                                   |     |
|-----------------------------------|-----|
| raise exceptions .....            | 35  |
| range checking .....              | 79  |
| <b>rbreak</b> .....               | 31  |
| reading symbols immediately ..... | 100 |
| readline .....                    | 131 |
| <b>readnow</b> .....              | 100 |

|                                                 |     |
|-------------------------------------------------|-----|
| redirection .....                               | 22  |
| reference card .....                            | 169 |
| reference declarations .....                    | 83  |
| register stack, AMD29K .....                    | 73  |
| registers .....                                 | 72  |
| regular expression .....                        | 31  |
| reloading symbols .....                         | 92  |
| remote connection without stubs .....           | 115 |
| remote debugging .....                          | 110 |
| remote programs, interrupting .....             | 114 |
| remote serial debugging summary .....           | 113 |
| remote serial debugging, overview .....         | 110 |
| remote serial protocol .....                    | 114 |
| remote serial stub .....                        | 111 |
| remote serial stub list .....                   | 111 |
| remote serial stub, initialization .....        | 111 |
| remote serial stub, main routine .....          | 111 |
| remote stub, example .....                      | 114 |
| remote stub, support routines .....             | 112 |
| <b>remotedebug</b> , MIPS プロトコル .....           | 128 |
| <b>remotedebug</b> , MIPS protocol .....        | 128 |
| <b>remotetimeout</b> .....                      | 124 |
| repeating commands .....                        | 13  |
| reporting bugs in GDB .....                     | 145 |
| <b>reset</b> .....                              | 119 |
| response time, MIPS debugging .....             | 51  |
| resuming execution .....                        | 41  |
| <b>RET</b> .....                                | 13  |
| <b>retransmit-timeout</b> , MIPS プロトコル .....    | 128 |
| <b>retransmit-timeout</b> , MIPS protocol ..... | 128 |
| <b>return</b> .....                             | 97  |
| returning from a function .....                 | 97  |
| <b>reverse-search</b> .....                     | 54  |
| <b>run</b> .....                                | 19  |
| running .....                                   | 19  |
| <b>Running</b> .....                            | 124 |
| running 29K programs .....                      | 119 |
| running and debugging Sparclet programs ....    | 125 |
| running VxWorks tasks .....                     | 123 |
| <b>rwatch</b> .....                             | 33  |

## S

|                                         |        |                                            |          |
|-----------------------------------------|--------|--------------------------------------------|----------|
| <b>s</b> .....                          | 42     | <b>set print max-symbolic-offset</b> ..... | 67       |
| saving symbol table .....               | 100    | <b>set print null-stop</b> .....           | 67       |
| scope .....                             | 88     | <b>set print object</b> .....              | 70       |
| <b>search</b> .....                     | 54     | <b>set print pretty</b> .....              | 67       |
| searching .....                         | 54     | <b>set print sevenbit-strings</b> .....    | 68       |
| <b>section</b> .....                    | 101    | <b>set print static-members</b> .....      | 70       |
| <b>select-frame</b> .....               | 48     | <b>set print symbol-filename</b> .....     | 66       |
| selected frame .....                    | 47     | <b>set print union</b> .....               | 68       |
| serial connections, debugging .....     | 115    | <b>set print vtbl</b> .....                | 70       |
| serial device, Hitachi micros .....     | 125    | <b>set processor args</b> .....            | 127      |
| serial line speed, Hitachi micros ..... | 125    | <b>set prompt</b> .....                    | 131      |
| serial line, <b>target remote</b> ..... | 114    | <b>set remotedebug</b> .....               | 115, 128 |
| serial protocol, GDB remote .....       | 114    | <b>set retransmit-timeout</b> .....        | 128      |
| <b>set</b> .....                        | 16     | <b>set rstack_high_address</b> .....       | 73       |
| <b>set args</b> .....                   | 21     | <b>set symbol-reloading</b> .....          | 93       |
| <b>set assembly-language</b> .....      | 57     | <b>set timeout</b> .....                   | 128      |
| <b>set check</b> .....                  | 78, 79 | <b>set variable</b> .....                  | 95       |
| <b>set check range</b> .....            | 79     | <b>set verbose</b> .....                   | 134      |
| <b>set check type</b> .....             | 78     | <b>set width</b> .....                     | 133      |
| <b>set complaints</b> .....             | 134    | <b>set write</b> .....                     | 97       |
| <b>set confirm</b> .....                | 135    | <b>set_debug_traps</b> .....               | 111      |
| <b>set demangle-style</b> .....         | 69     | setting variables .....                    | 95       |
| <b>set editing</b> .....                | 131    | setting watchpoints .....                  | 33       |
| <b>set endian auto</b> .....            | 109    | <b>SH</b> .....                            | 111      |
| <b>set endian big</b> .....             | 109    | <b>sh-stub.c</b> .....                     | 111      |
| <b>set endian little</b> .....          | 109    | <b>share</b> .....                         | 102      |
| <b>set environment</b> .....            | 21     | shared libraries .....                     | 102      |
| <b>set extension-language</b> .....     | 77     | <b>sharedlibrary</b> .....                 | 102      |
| <b>set gnutarget</b> .....              | 106    | <b>shell</b> .....                         | 12       |
| <b>set height</b> .....                 | 133    | shell escape .....                         | 12       |
| <b>set history expansion</b> .....      | 132    | <b>show</b> .....                          | 16       |
| <b>set history filename</b> .....       | 132    | <b>show args</b> .....                     | 21       |
| <b>set history save</b> .....           | 132    | <b>show check range</b> .....              | 79       |
| <b>set history size</b> .....           | 132    | <b>show check type</b> .....               | 78       |
| <b>set input-radix</b> .....            | 133    | <b>show commands</b> .....                 | 133      |
| <b>set language</b> .....               | 76     | <b>show complaints</b> .....               | 134      |
| <b>set listsize</b> .....               | 53     | <b>show confirm</b> .....                  | 135      |
| <b>set machine</b> .....                | 126    | <b>show convenience</b> .....              | 72       |
| <b>set memory mod</b> .....             | 126    | <b>show copying</b> .....                  | 17       |
| <b>set mipsfpu</b> .....                | 127    | <b>show demangle-style</b> .....           | 69       |
| <b>set output-radix</b> .....           | 134    | <b>show directories</b> .....              | 56       |
| <b>set print address</b> .....          | 65     | <b>show editing</b> .....                  | 131      |
| <b>set print array</b> .....            | 67     | <b>show endian</b> .....                   | 109      |
| <b>set print asm-demangle</b> .....     | 69     | <b>show environment</b> .....              | 21       |
| <b>set print demangle</b> .....         | 69     | <b>show gnutarget</b> .....                | 106      |
| <b>set print elements</b> .....         | 67     | <b>show height</b> .....                   | 133      |
|                                         |        | <b>show history</b> .....                  | 132      |
|                                         |        | <b>show input-radix</b> .....              | 134      |

|                                                                        |          |
|------------------------------------------------------------------------|----------|
| <code>show language</code> .....                                       | 77       |
| <code>show listsize</code> .....                                       | 53       |
| <code>show machine</code> .....                                        | 126      |
| <code>show mipsfpu</code> .....                                        | 127      |
| <code>show output-radix</code> .....                                   | 134      |
| <code>show paths</code> .....                                          | 21       |
| <code>show print address</code> .....                                  | 66       |
| <code>show print array</code> .....                                    | 67       |
| <code>show print asm-demangle</code> .....                             | 69       |
| <code>show print demangle</code> .....                                 | 69       |
| <code>show print elements</code> .....                                 | 67       |
| <code>show print max-symbolic-offset</code> .....                      | 67       |
| <code>show print object</code> .....                                   | 70       |
| <code>show print pretty</code> .....                                   | 68       |
| <code>show print sevenbit-strings</code> .....                         | 68       |
| <code>show print static-members</code> .....                           | 70       |
| <code>show print symbol-filename</code> .....                          | 66       |
| <code>show print union</code> .....                                    | 68       |
| <code>show print vtbl</code> .....                                     | 70       |
| <code>show processor</code> .....                                      | 127      |
| <code>show prompt</code> .....                                         | 131      |
| <code>show remotedebug</code> .....                                    | 115, 128 |
| <code>show retransmit-timeout</code> .....                             | 128      |
| <code>show rstack_high_address</code> .....                            | 74       |
| <code>show symbol-reloading</code> .....                               | 93       |
| <code>show timeout</code> .....                                        | 128      |
| <code>show user</code> .....                                           | 138      |
| <code>show values</code> .....                                         | 71       |
| <code>show verbose</code> .....                                        | 134      |
| <code>show version</code> .....                                        | 16       |
| <code>show warranty</code> .....                                       | 17       |
| <code>show width</code> .....                                          | 133      |
| <code>show write</code> .....                                          | 98       |
| <code>show-all-if-ambiguous</code> .....                               | 154      |
| <code>si</code> .....                                                  | 43       |
| <code>signal</code> .....                                              | 96       |
| <code>signals</code> .....                                             | 43       |
| <code>silent</code> .....                                              | 39       |
| <code>sim</code> .....                                                 | 128      |
| <code>simulator</code> .....                                           | 128      |
| <code>simulator, H8/300 or H8/500</code> .....                         | 128      |
| <code>simulator, Hitachi SH</code> .....                               | 128      |
| <code>simulator, Z8000</code> .....                                    | 128      |
| <code>size of screen</code> .....                                      | 133      |
| <code>sleep</code> .....                                               | 26       |
| <code>software watchpoints</code> .....                                | 33       |
| <code>source</code> .....                                              | 139      |
| <code>source path</code> .....                                         | 55       |
| <code>Sparc</code> .....                                               | 111      |
| <code>sparc-stub.c</code> .....                                        | 111      |
| <code>sparc1-stub.c</code> .....                                       | 111      |
| <code>Sparclet</code> .....                                            | 123      |
| <code>Sparclet プログラムの実行とデバッグ</code> [Sparclet<br>プログラムのじっこうとデバッグ]..... | 125      |
| <code>Sparclet へのダウンロード</code> .....                                   | 124      |
| <code>SparcLite</code> .....                                           | 111      |
| <code>speed</code> .....                                               | 125      |
| <code>ST2000 用の補助的なコマンド</code> [ST2000 ようのほ<br>じょてきなコマンド].....         | 121      |
| <code>ST2000 auxiliary commands</code> .....                           | 121      |
| <code>st2000 cmd</code> .....                                          | 121      |
| <code>stack frame</code> .....                                         | 47       |
| <code>stack on Alpha</code> .....                                      | 51       |
| <code>stack on MIPS</code> .....                                       | 51       |
| <code>stack traces</code> .....                                        | 48       |
| <code>stacking targets</code> .....                                    | 105      |
| <code>starting</code> .....                                            | 19       |
| <code>STDEBUG コマンド ( ST2000 )</code> .....                             | 121      |
| <code>STDEBUG commands (ST2000)</code> .....                           | 121      |
| <code>step</code> .....                                                | 42       |
| <code>stepi</code> .....                                               | 43       |
| <code>stepping</code> .....                                            | 41       |
| <code>stopped threads</code> .....                                     | 45       |
| <code>stub example, remote debugging</code> .....                      | 114      |
| <code>stupid questions</code> .....                                    | 135      |
| <code>switching threads</code> .....                                   | 24       |
| <code>switching threads automatically</code> .....                     | 26       |
| <code>symbol decoding style, C++</code> .....                          | 69       |
| <code>symbol dump</code> .....                                         | 93       |
| <code>symbol names</code> .....                                        | 91       |
| <code>symbol overloading</code> .....                                  | 40       |
| <code>symbol table</code> .....                                        | 99       |
| <code>symbol-file</code> .....                                         | 99       |
| <code>symbols, reading immediately</code> .....                        | 100      |
| <b>T</b>                                                               |          |
| <code>target</code> .....                                              | 105      |
| <code>target abug</code> .....                                         | 106      |
| <code>target adapt</code> .....                                        | 106      |
| <code>target amd-eb</code> .....                                       | 107      |
| <code>target array</code> .....                                        | 107      |
| <code>target bug</code> .....                                          | 107      |
| <code>target byte order</code> .....                                   | 109      |
| <code>target core</code> .....                                         | 106      |
| <code>target cpu32bug</code> .....                                     | 107      |

target debug ..... 107  
 target ddb ..... 107  
 target ddb *port* ..... 127  
 target dink32 ..... 107  
 target e7000 ..... 107, 126  
 target es1800 ..... 107  
 target est ..... 107  
 target exec ..... 106  
 target hms ..... 107  
 target lsi ..... 107  
 target lsi *port* ..... 127  
 target m32r ..... 107  
 target mips ..... 107  
 target mips *port* ..... 126  
 target mon960 ..... 107  
 target nindy ..... 107  
 target nrom ..... 107  
 target op50n ..... 108  
 target pmon ..... 108  
 target pmon *port* ..... 127  
 target ppcbug ..... 108  
 target ppcbug1 ..... 108  
 target r3900 ..... 108  
 target rdi ..... 108  
 target rdp ..... 108  
 target remote ..... 106  
 target remote、シリアル回線 [target remote、  
   シリアルかいせん] ..... 114  
 target remote、TCP ポート ..... 114  
 target rom68k ..... 108  
 target rombug ..... 108  
 target sds ..... 108  
 target sh3 ..... 108  
 target sh3e ..... 108  
 target sim ..... 106, 128  
 target sparclite ..... 108  
 target st2000 ..... 108  
 target udi ..... 108  
 target vxworks ..... 108  
 target w89k ..... 108  
 tbreak ..... 31  
 TCP port, target remote ..... 114  
 terminal ..... 22  
 thbreak ..... 31  
 this ..... 82  
 thread apply ..... 26  
 thread breakpoints ..... 45  
 thread identifier (GDB) ..... 25

thread identifier (system) ..... 25  
 thread number ..... 25  
 thread *threadno* ..... 26  
 threads and watchpoints ..... 34  
 threads of execution ..... 24  
 threads, automatic switching ..... 26  
 threads, continuing ..... 45  
 threads, stopped ..... 45  
 timeout、MIPS プロトコル ..... 128  
 timeout, MIPS protocol ..... 128  
 tracebacks ..... 48  
 tty ..... 22  
 type casting memory ..... 59  
 type checking ..... 78  
 type conversions in C++ ..... 83

## U

u ..... 42  
 udi ..... 119  
 UDI ..... 119  
 UDI 経由の AMD29K [UDI けいゆの AMD29K]  
   ..... 119  
 undisplay ..... 65  
 unknown address, locating ..... 62  
 unset environment ..... 21  
 until ..... 42  
 up ..... 49  
 up-silently ..... 50  
 user-defined command ..... 137

## V

value history ..... 70  
 variable name conflict ..... 60  
 variable values, wrong ..... 60  
 variables, setting ..... 95  
 version number ..... 16  
 visible-stats ..... 154  
 VxWorks ..... 122  
 VxWorks へのダウンロード ..... 123  
 VxWorks タスクの実行 [VxWorks タスクの実行]  
   ..... 123  
 vxworks-timeout ..... 122

**W**

|                                           |     |
|-------------------------------------------|-----|
| <b>watch</b> .....                        | 33  |
| watchpoints .....                         | 29  |
| watchpoints and threads .....             | 34  |
| <b>whatis</b> .....                       | 91  |
| <b>where</b> .....                        | 48  |
| <b>while</b> .....                        | 137 |
| wild pointer, interpreting .....          | 67  |
| word completion .....                     | 13  |
| working directory .....                   | 55  |
| working directory (of your program) ..... | 22  |
| working language .....                    | 75  |
| writing into corefiles .....              | 97  |
| writing into executables .....            | 97  |

|                    |    |
|--------------------|----|
| wrong values ..... | 60 |
|--------------------|----|

**X**

|                     |    |
|---------------------|----|
| <b>x</b> .....      | 63 |
| XCOFF and C++ ..... | 82 |

**Y**

|                    |     |
|--------------------|-----|
| yanking text ..... | 150 |
|--------------------|-----|

**Z**

|                       |     |
|-----------------------|-----|
| Z8000 シミュレータ .....    | 128 |
| Z8000 simulator ..... | 128 |

The body of this manual is set in  
cmr10 at 10.95pt,  
with headings in **cmb10 at 10.95pt**  
and examples in cmtt10 at 10.95pt.  
*cmti10 at 10.95pt*,  
**cmb10 at 10.95pt**, and  
*cmsl10 at 10.95pt*  
are used for emphasis.





# Table of Contents

|                                      |           |
|--------------------------------------|-----------|
| <b>GDB の要約</b> .....                 | <b>1</b>  |
| フリー・ソフトウェア .....                     | 1         |
| GDB に貢献した人々 .....                    | 1         |
| <b>1 GDB セッションのサンプル</b> .....        | <b>5</b>  |
| <b>2 GDB の起動・終了</b> .....            | <b>9</b>  |
| 2.1 GDB の起動 .....                    | 9         |
| 2.1.1 ファイルの選択 .....                  | 10        |
| 2.1.2 モードの選択 .....                   | 11        |
| 2.2 GDB の終了 .....                    | 12        |
| 2.3 シェル・コマンド .....                   | 12        |
| <b>3 GDB コマンド</b> .....              | <b>13</b> |
| 3.1 コマンドの構文 .....                    | 13        |
| 3.2 コマンド名の補完 .....                   | 13        |
| 3.3 ヘルプの表示 .....                     | 15        |
| <b>4 GDB 配下でのプログラムの実行</b> .....      | <b>19</b> |
| 4.1 デバッグのためのコンパイル .....              | 19        |
| 4.2 ユーザ・プログラムの起動 .....               | 19        |
| 4.3 ユーザ・プログラムの引数 .....               | 20        |
| 4.4 ユーザ・プログラムの環境 .....               | 21        |
| 4.5 ユーザ・プログラムの作業ディレクトリ .....         | 22        |
| 4.6 ユーザ・プログラムの入出力 .....              | 22        |
| 4.7 既に実行中のプロセスのデバッグ .....            | 23        |
| 4.8 子プロセスの終了 .....                   | 23        |
| 4.9 プロセス情報 .....                     | 24        |
| 4.10 マルチスレッド・プログラムのデバッグ .....        | 24        |
| 4.11 マルチプロセス・プログラムのデバッグ .....        | 26        |
| <b>5 停止と継続</b> .....                 | <b>29</b> |
| 5.1 ブ레이크ポイント、ウォッチポイント、キャッチポイント ..... | 29        |
| 5.1.1 ブ레이크ポイントの設定 .....              | 30        |
| 5.1.2 ウォッチポイントの設定 .....              | 33        |
| 5.1.3 キャッチポイントの設定 .....              | 34        |
| 5.1.4 ブ레이크ポイントの削除 .....              | 36        |
| 5.1.5 ブ레이크ポイントの無効化 .....             | 36        |
| 5.1.6 ブ레이크ポイントの成立条件 .....            | 37        |
| 5.1.7 ブ레이크ポイント・コマンド・リスト .....        | 39        |
| 5.1.8 ブ레이크ポイント・メニュー .....            | 40        |

|          |                             |           |
|----------|-----------------------------|-----------|
| 5.2      | 継続実行とステップ実行 .....           | 41        |
| 5.3      | シグナル .....                  | 43        |
| 5.4      | マルチスレッド・プログラムの停止と起動 .....   | 45        |
| <b>6</b> | <b>スタックの検査 .....</b>        | <b>47</b> |
| 6.1      | スタック・フレーム .....             | 47        |
| 6.2      | バックトレース .....               | 48        |
| 6.3      | フレームの選択 .....               | 49        |
| 6.4      | フレームに関する情報 .....            | 50        |
| 6.5      | MIPS/Alpha マシンの関数スタック ..... | 51        |
| <b>7</b> | <b>ソース・ファイルの検査 .....</b>    | <b>53</b> |
| 7.1      | ソース行の表示 .....               | 53        |
| 7.2      | ソース・ファイル内の検索 .....          | 54        |
| 7.3      | ソース・ディレクトリの指定 .....         | 55        |
| 7.4      | ソースとマシン・コード .....           | 56        |
| <b>8</b> | <b>データの検査 .....</b>         | <b>59</b> |
| 8.1      | 式 .....                     | 59        |
| 8.2      | プログラム変数 .....               | 60        |
| 8.3      | 人工配列 .....                  | 61        |
| 8.4      | 出力フォーマット .....              | 62        |
| 8.5      | メモリの調査 .....                | 63        |
| 8.6      | 自動表示 .....                  | 64        |
| 8.7      | 表示設定 .....                  | 65        |
| 8.8      | 値ヒストリ .....                 | 70        |
| 8.9      | コンビニエンス変数 .....             | 71        |
| 8.10     | レジスタ .....                  | 72        |
| 8.11     | 浮動小数ハードウェア .....            | 74        |
| <b>9</b> | <b>異なる言語の使用 .....</b>       | <b>75</b> |
| 9.1      | ソース言語の切り替え .....            | 75        |
| 9.1.1    | ファイル拡張子と言語のリスト .....        | 75        |
| 9.1.2    | 作業言語の設定 .....               | 76        |
| 9.1.3    | GDB によるソース言語の推定 .....       | 76        |
| 9.2      | 言語の表示 .....                 | 77        |
| 9.3      | 型と範囲のチェック .....             | 77        |
| 9.3.1    | 型チェックの概要 .....              | 78        |
| 9.3.2    | 範囲チェックの概要 .....             | 79        |
| 9.4      | サポートされる言語 .....             | 80        |
| 9.4.1    | C/C++ .....                 | 80        |
| 9.4.1.1  | C/C++演算子 .....              | 80        |
| 9.4.1.2  | C/C++定数 .....               | 82        |
| 9.4.1.3  | C++式 .....                  | 82        |
| 9.4.1.4  | C/C++のデフォルト .....           | 83        |
| 9.4.1.5  | C/C++の型チェックと範囲チェック .....    | 83        |
| 9.4.1.6  | GDB と C .....               | 83        |

|           |                          |            |
|-----------|--------------------------|------------|
| 9.4.1.7   | C++用の GDB 機能             | 84         |
| 9.4.2     | Modula-2                 | 85         |
| 9.4.2.1   | Modula-2 演算子             | 85         |
| 9.4.2.2   | 組み込み関数と組み込みプロシージャ        | 86         |
| 9.4.2.3   | 定数                       | 87         |
| 9.4.2.4   | Modula-2 デフォルト           | 88         |
| 9.4.2.5   | 標準 Modula-2 との差異         | 88         |
| 9.4.2.6   | Modula-2 の型チェックと範囲チェック   | 88         |
| 9.4.2.7   | スコープ演算子::と               | 88         |
| 9.4.2.8   | GDB と Modula-2           | 89         |
| <b>10</b> | <b>シンボル・テーブルの検査</b>      | <b>91</b>  |
| <b>11</b> | <b>実行の変更</b>             | <b>95</b>  |
| 11.1      | 変数への代入                   | 95         |
| 11.2      | 異なるアドレスにおける処理継続          | 96         |
| 11.3      | ユーザ・プログラムへのシグナルの通知       | 96         |
| 11.4      | 関数からの復帰                  | 97         |
| 11.5      | プログラム関数の呼び出し             | 97         |
| 11.6      | プログラムへのパッチ適用             | 97         |
| <b>12</b> | <b>GDB ファイル</b>          | <b>99</b>  |
| 12.1      | ファイルを指定するコマンド            | 99         |
| 12.2      | シンボル・ファイル読み込み時のエラー       | 102        |
| <b>13</b> | <b>デバッグ・ターゲットの指定</b>     | <b>105</b> |
| 13.1      | アクティブ・ターゲット              | 105        |
| 13.2      | ターゲットを管理するコマンド           | 105        |
| 13.3      | ターゲットのバイト・オーダの選択         | 109        |
| 13.4      | リモート・デバッグ                | 110        |
| 13.4.1    | GDB リモート・シリアル・プロトコル      | 110        |
| 13.4.1.1  | スタブの提供する機能               | 111        |
| 13.4.1.2  | スタブに対する必須作業              | 112        |
| 13.4.1.3  | ここまでのまとめ                 | 113        |
| 13.4.1.4  | 通信プロトコル                  | 114        |
| 13.4.1.5  | gdbserverプログラムの使用        | 115        |
| 13.4.1.6  | gdbserve.nlmプログラムの使用     | 117        |
| 13.4.2    | GDB とリモート i960 ( Nindy ) | 117        |
| 13.4.2.1  | Nindy 使用時の起動方法           | 118        |
| 13.4.2.2  | Nindy 用のオプション            | 118        |
| 13.4.2.3  | Nindy reset コマンド         | 118        |
| 13.4.3    | AMD29K 用の UDI プロトコル      | 119        |
| 13.4.4    | AMD29K の EBMON プロトコル     | 119        |
| 13.4.4.1  | 通信セットアップ                 | 119        |
| 13.4.4.2  | EB29K クロス・デバッグ           | 121        |
| 13.4.4.3  | リモート・ログ                  | 121        |
| 13.4.5    | GDB と Tandem ST2000      | 121        |

|           |                                      |            |
|-----------|--------------------------------------|------------|
| 13.4.6    | GDB と VxWorks .....                  | 122        |
| 13.4.6.1  | VxWorks への接続 .....                   | 122        |
| 13.4.6.2  | VxWorks ダウンロード .....                 | 123        |
| 13.4.6.3  | タスクの実行 .....                         | 123        |
| 13.4.7    | GDB と Sparclet .....                 | 123        |
| 13.4.7.1  | デバッグするファイルの選択 .....                  | 124        |
| 13.4.7.2  | Sparclet への接続 .....                  | 124        |
| 13.4.7.3  | Sparclet ダウンロード .....                | 124        |
| 13.4.7.4  | 実行とデバッグ .....                        | 125        |
| 13.4.8    | GDB と日立的マイクロ・プロセッサ .....             | 125        |
| 13.4.8.1  | 日立ボードへの接続 .....                      | 125        |
| 13.4.8.2  | E7000 インサーキット・エミュレータの使用<br>.....     | 126        |
| 13.4.8.3  | 日立マイクロ・プロセッサ用の特別な GDB コ<br>マンド ..... | 126        |
| 13.4.9    | GDB とリモート MIPS ボード .....             | 126        |
| 13.4.10   | シミュレートされた CPU ターゲット .....            | 128        |
| <b>14</b> | <b>GDB の制御 .....</b>                 | <b>131</b> |
| 14.1      | プロンプト .....                          | 131        |
| 14.2      | コマンド編集 .....                         | 131        |
| 14.3      | コマンド・ヒストリ .....                      | 132        |
| 14.4      | 画面サイズ .....                          | 133        |
| 14.5      | 数値 .....                             | 133        |
| 14.6      | オプションの警告およびメッセージ .....               | 134        |
| <b>15</b> | <b>一連のコマンドのグループ化 .....</b>           | <b>137</b> |
| 15.1      | ユーザ定義コマンド .....                      | 137        |
| 15.2      | ユーザ定義コマンド・フック .....                  | 138        |
| 15.3      | コマンド・ファイル .....                      | 139        |
| 15.4      | 制御された出力を得るためのコマンド .....              | 139        |
| <b>16</b> | <b>GNU Emacs の中での GDB の使用 .....</b>  | <b>141</b> |
| <b>17</b> | <b>GDB のバグ報告 .....</b>               | <b>145</b> |
| 17.1      | 本当にバグを見つけたのかどうかを知る方法 .....           | 145        |
| 17.2      | バグの報告方法 .....                        | 145        |

|                   |                           |            |
|-------------------|---------------------------|------------|
| <b>18</b>         | <b>コマンドライン編集</b>          | <b>149</b> |
| 18.1              | 行編集入門                     | 149        |
| 18.2              | Readline の操作              | 149        |
| 18.2.1            | Readline の基本              | 149        |
| 18.2.2            | Readline 移動コマンド           | 150        |
| 18.2.3            | Readline キル ( kill ) コマンド | 150        |
| 18.2.4            | Readline の引数              | 151        |
| 18.2.5            | ヒストリ中のコマンドの検索             | 151        |
| 18.3              | Readline 初期化ファイル          | 152        |
| 18.3.1            | Readline 初期化ファイルの構文       | 152        |
| 18.3.2            | 条件初期化構文                   | 156        |
| 18.3.3            | 初期化ファイルのサンプル              | 156        |
| 18.4              | バインド可能な Readline コマンド     | 160        |
| 18.4.1            | 移動のためのコマンド                | 160        |
| 18.4.2            | ヒストリを操作するためのコマンド          | 160        |
| 18.4.3            | テキストを変更するためのコマンド          | 161        |
| 18.4.4            | キル ( kill ) と再挿入 ( yank ) | 162        |
| 18.4.5            | 数値引数の指定                   | 163        |
| 18.4.6            | Readline による入力補完          | 164        |
| 18.4.7            | キーボード・マクロ                 | 164        |
| 18.4.8            | その他のコマンド                  | 165        |
| 18.5              | Readline の vi モード         | 166        |
| <b>Appendix A</b> | <b>ヒストリの対話的な使用</b>        | <b>167</b> |
| A.1               | ヒストリの操作                   | 167        |
| A.1.1             | イベント指定子                   | 167        |
| A.1.2             | ワード指定子                    | 167        |
| A.1.3             | 修飾子                       | 168        |
| <b>Appendix B</b> | <b>ドキュメントのフォーマット</b>      | <b>169</b> |
| <b>Appendix C</b> | <b>GDB のインストール</b>        | <b>171</b> |
| C.1               | 異なるディレクトリでの GDB のコンパイル    | 172        |
| C.2               | ホストとターゲットの名前の指定           | 173        |
| C.3               | configure オプション           | 174        |
|                   | <b>インデックス</b>             | <b>175</b> |

