日本語マニュアルだよＧＤＢ Ver 4.12

The GNU Source-Level Debugger

第 3.0 版 平成 6 年 5 月 18 日

by Shiro Fujiwara — akashi@sy.ssl.fujitsu.co.jp

はじめに

本書は、『Debugging with GDB』から、C言語のデバッグに必要だと思われる部分を抽出しています。また、原書との対応をとれるように、章や節の表題を変更しませんでした。しかし!!原本とは記述方法がまるで違うかもしれません(^^;)。

本書の作成にあたり、以下のマニュアルを参照しました(参照したんだけどなぁ)。

- Debugging width GDB(Ver4.10pl1).
- Debugging width GDB(Ver4.12).

— お願い —

- 本書を無断で転載しないようお願いします.
- 本書は予告なしに変更されることがあります.

目 次

Part I

GNU GENERAL PUBLIC LICENSE

Chapter 1

日本語訳 GNU GENERAL PUBLIC LICENCE

このパート「GNU GENERAL PUBLIC LICENCE」は、「JGawk の日本語マニュアル」より転載したもので、この文書の配布形態を忠実に守っていることを明記しておきます。また、このパートに限り、この文書の配布条件に従うのであれば、転載を自由に行なうことができます。

1.1 注意

英文文書[1] を正式文書とする。この和文文書は弁護士の意見を採り入れて、できるだけ正確に英文文書を翻訳したものであるが、法律的に有効な契約書ではない。

1.2 和文文書自体の再配布に関して

いかなる媒体でも次の条件がすべて満たされている場合に限り、本和文文書をそのまま複写し配布することを許可する。また、あなたは第三者に対して本許可告知と同一の許可を与える場合に限り、再配布することが許可されています。

- 受領、配布されたコピーに著作権表示および本許諾告知が前もって載せられていること。

---

[1] GNU General Public Licence

- コピーの受領者がさらに再配布する場合、その配布者が本告知と同じ許可を与えていること。
- 和文文書の本文を改変しないこと。

1.3 GNU 一般公有使用許諾書

1989 年 2 月、バージョン 1

Copyright (C) 1989 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

何人も、以下の内容を変更しないでそのまま複写する場合に限り、本使用許諾書を複製したり頒布することができます。

1.3.1 はじめに

ソフトウェア会社の使用許諾契約書は、多くの場合、その企業の意のままにユーザを縛ろうとしています。それに対して、我々の一般公有使用許諾は、フリー・ソフトウェアを共有したり変更する自由をユーザに保証するためのもの、即ちフリー・ソフトウェアがそのユーザ全てにとってフリーであることを保証するためのものです。本使用許諾は、Free Software Foundation のソフトウェアに適用されるだけでなく、プログラムの作成者が本使用許諾に依るとした場合のそのプログラムにも適用することができます。また、ユーザのプログラムのためにも利用することができます。

我々がフリー・ソフトウェアについて言う場合は自由のことに言及しているのであって、価格のことではありません。特に、一般公有使用許諾の各条項は、次の事柄を確実に実現することを目的として立案されています。

- フリー・ソフトウェアの複製物を自由に頒布したり販売できること。
- 希望しさえすればソース・コードを現実に入手できるか、あるいはその入手が可能であること。
- 入手したソフトウェアを変更したり、新しいフリー・プログラムの一部として使用できること。

- 以上の各内容を行なうことができるということをユーザ自身が知っていること。

このようなユーザの権利を守るために、我々は、何人もこれらの権利を否定したり、あるいは放棄するようにユーザに求めることはできないという制限条項を設ける必要があります。これらの制限条項は、ユーザが、フリー・ソフトウェアの複製物を頒布したり変更しようとする場合には、そのユーザ自身が守るべき義務ともなります。

例えば、あなたがフリー・ソフトウェアの複製物を頒布する場合、有償か無償かにかかわらず、あなたは自分の持っている権利を全て相手に与えなければなりません。あなたは、相手もまたソース・コードを受け取ったり入手できるということを認めなければなりません。さらにあなたは、相手が受領者へそれらの権利を持っているということを、その相手に知らせなければなりません。

我々は次の二つの方法でユーザの権利を守ります。(1) ソフトウェアに著作権を主張し、(2) 本使用許諾の条項の下でソフトウェアを複製・頒布・変更する権利をユーザに与えます。

また、各作成者や我々自身を守るために、本フリー・ソフトウェアが無保証であることを全ての人々が了解している必要があります。更に、他の誰かによって変更されたソフトウェアが頒布された場合、受領者はそのソフトウェアがオリジナル・バージョンではないということを知らされる必要があります。それは、他人の関与によって原開発者に対する評価が影響されないようにするためです。

複写・頒布・変更に対する正確な条項と条件を次に示します。

GNU 一般公有使用許諾の下での複製、頒布、変更に関する条項と条件

1. 本使用許諾は、一般公有使用許諾の各条項に従って頒布されるという著作権者からの告知文が表示されているプログラムやその他の作成物に適用されます。以下において「プログラム」とは、そのようなプログラムや作成物を指すものとし、また、「プログラム生成物」とは、上述した「プログラム」自身、及びその「プログラム」の全部又は一部の作成を、そのまま又は変更して内部に組み込んだ作成物を意味するものとします。本使用許諾によって許諾を受ける者を「あなた」と呼びます。

2. あなたは、どのような媒体上へ複製しようとする場合であっても、入手した「プログラム」のソース・コードをそのままの内容で複写した上で適正な著作権表示と保証の放棄と明確、且つ適正に付記する場合に限り、複製又は頒布することができます。その場合、本一般公有使用許諾及び無保証に関する記載部分は、全て元のままの形で表示して下さい。また、「プログラム」の頒布先に対しては、「プログラム」と共に本一般公有使用許諾書のコピーを渡して下さい。複製物を引き渡す際の実費は請求することができます。

3. 次の各条件を満たしている限り、あなたは、「プログラム」又はその一部分を、変更することができます。更に、上記第 2 項を満たせば、その変更版を複製したり頒布することもできます。

    - ファイルを変更した旨とその変更日とを、変更したファイル上に明確に表示すること。
    - 変更したか否かを問わず、凡そ「プログラム」又はその一部分を内部に組み込んでいる作成物を頒布する場合には、本一般公有使用許諾の条項に従って無償で使用許諾すること。但し、頒布先の全て又はその一部の者に対して、あなたが独自に保証することは構いません。
    - プログラムが実行時に通常の対話的な方法でコマンドを読むようになっているとすれば、最も単純、且つ普通の方法で対話的にそのプログラムを実行する時に、次の内容を示す文言がプリント・アウトされるか、或は画面に表示されること。
    - 適切な著作権表示。
    - 無保障であること (あなたが独自に保証する場合は、その旨)。
    - 頒布を受ける者も、本一般公有使用許諾と同一の条項に従ってそのプログラムを再頒布できること。
    - 頒布を受ける者が本一般公有使用許諾書の写しを回覧する方法。
    - 複製物の譲渡に要する実費は請求できること。また、あなた独自の保証を行なう場合はそれを有償とすることができること。

    本「プログラム」(または、その派生物) と他の別個のプログラムとを、保管や頒布のために同一の媒体上にまとめて記録したとしても、本使用許諾の条項は他の別個のプログラムには適用されません。

4. あなたは、以下のうちいずれか 1 つを満たす限り、上記第 2 項及び第 3 項に従って「プログラム」(または、上記の条項 2 のもとの

その一部分あるいはその派生物) をオブジェクト・コードまたは実行可能な形式で複製および頒布することができます。

- 対応する機械読み取り可能なソース・コード一式を一緒に引き渡すこと。その場合、そのソース・コードの引き渡しは上記第2項及び第3項に従って行なわれること。
- 少なくとも3年間の有効期間を定め、且つその期間内であれば対応する機械読み取り可能なソース・コード一式を無償で(ただし、少額の頒布実費は請求できる) 提供する旨、及びその場合には上記第2項及び第3項に従って提供される旨を記載した書面を、一緒に引き渡すこと。
- 対応するソース・コードを入手できる所について、あなたが得た情報を提供すること (この選択肢は、営利を目的としない頒布であって、且つあなたがオブジェクト・コードあるいは実行可能形式のプログラムしか入手していない場合にのみ適用される選択項目です。)

上記においてソース・コードとは、変更作業に適した記述形式を指します。また、実行可能形式のファイルに対応するソース・コード一式とは、それに含まれる全モジュールに対応する全てのソース・コードを指しますが、例外として、実行可能なファイルが動作するオペレーティング・システムに付随する標準ライブラリのモジュールのソース・コードやそのオペレーティング・システムに付随する定義ファイルのソース・コードを含ませる必要はありません。

5. 本一般公有使用許諾が明示的に許諾している場合を除き、あなたは、「プログラム」を複製、変更、サブライセンス、頒布、譲渡することができません。本使用許諾に従わずに「プログラム」を複製、変更、サブライセンス、頒布、譲渡しようとする行為は、それ自体が無効であり、且つ、本使用許諾があなたに許諾している「プログラム」の使用権限を自動的に消滅させます。その場合、本使用許諾に従ってあなたから複製物やその使用許諾を得ている第三者は、本使用許諾に完全に従っている限り、引続き有効な使用権限を持つものとします。

6. あなたが「プログラム」(あるいはその「プログラム生成物」) の複製、頒布、変更を行なえば、それ自体で、それらの各行為を行なう権利と、本使用許諾が定める全ての条項に従うことを、あなたが受け入れたものとみなします。

7. あなたが「プログラム」(あるいはその「プログラム生成物」) を再頒布すると自動的に、その受領者は、元の使用許諾者から、本使用許諾の条項に従って「プログラム」を複製、頒布、変項することを内容とする使用許諾を受けたものとします。あなたは、受領者に許諾された権利の行使について、更に制約を加えることはできません。

8. Free Software Foundation は随時、一般公有使用許諾の改訂版、又は新版を公表することがあります。そのような新しいバージョンは、現行のバージョンと基本的に変わるところはありませんが、新しい問題や懸案事項に対応するために細部では異なるかもしれません。

   各バージョンは、バージョン番号によって区別します。「プログラム」中に使用許諾のバージョン番号の指定がある場合は、その指定されたバージョンか、又はその後に Free Software Foundation から公表されているいずれかのバージョンから 1 つを選択して、その条項と条件に従って下さい。「プログラム」中に使用許諾のバージョン番号の指定がない場合は、Free Software Foundation が公表したどのバージョンでも選択することができます。

9. 「プログラム」の一部を頒布条件の異なる他のフリー・プログラムに組み込みたい場合は、その開発者に書面で許可を求めて下さい。Free Software Foundation が著作権を持っているソフトウェアについては、Free Software Foundation へ書面を提出して下さい。このような場合へ対応するために我々は例外的処理をすることもありますが、その判断基準となるのは、次の 2 つの目標の実現に合致するか否かという点です。即ち、一つは我々のフリー・ソフトウェアの全ての派生物をフリーな状態に保つことであり、もう一つはソフトウェアの共有と再利用とを広く促進させることです。

無保証

10. プログラム」は無償で使用許諾されますので、適用法令の範囲内で、「プログラム」の保証は一切ありません。著作権者やその他の第三者は全く無保証で「そのまま」の状態で、且つ、明示か暗黙であるかを問わず一切の保証をつけないで提供するものとします。ここでいう保証とは、市場性や特定目的適合性についての暗黙の保証も含まれますが、それに限定されるものではありません。「プログラム」の品質や性能に関する全てのリスクはあなたが負うものとします。

「プログラム」に欠陥があるとわかった場合、それに伴う一切の派生費用や修理・訂正に要する費用は全てあなたの負担とします。

11. 適用法令の定め、又は書面による合意がある場合を除き、著作権者や上記許諾を受けて「プログラム」の変更・再頒布を為し得る第三者は、「プログラム」を使用したこと、または使用できないことに起因する一切の損害について何らの責任も負いません。著作権者や前記の第三者が、そのような損害の発生する可能性について知らされていた場合でも同様です。なお、ここでいう損害には、通常、損害、特別損害、偶発損害、間接損害が含まれます(データの消失、又はその正確さの喪失、あなたや第三者が被った損失、他のプログラムとのインタフェースの不適合化、等も含まれますが、これに限定されるものではありません)。

— — — — — — — — — — — — — — — — (以上) — — — —

Part II

Debbuging with GDB

Chapter 1

Summary of GDB

「GDBってなに？」と聞かれるたら、次のように答えましょう。

「自分のコーディング能力に自信がないか、コンパイラの吐き出すコードが信用できない人の心を癒してくれる魔法のアイテムだよ。」

「テスト工程の時には特に便利だね。疑似的ないろんな状況を簡単につくれるからね。」

デバッガは、使い方を間違えると、とんでもない問題をソースに埋め込んでしまいます。でも、使う人がちゃんとした認識で使いこなせるとしたら、作業効率を大幅にアップしてくれる魔法のアイテムだと思います。中でも、GDBはとても高機能なので、使い方さえ間違えなければ（ひつこいってか(^^;)）、あなたの障害検出＆動作テストのパートナーになってくれるでしょう。でも、コーディングの基本は机上デバッグだということをお忘れずに。あなたの、頭の中で動かないプログラムがコンピュータの上で動くわけないのですから。

それでは、原文からまじめにGDBの目的を紹介しましょう。

GDBデバッガの目的は、プログラム内部で何が起こっているか知る手段を提供することです。そのために、GDBは、以下の4つの主な機能を提供しています。

- プログラムの動作を詳細に伝える。
- プログラムの実行を詳細な条件で停止させる。

- プログラムが止まっているときの状況を調べる。
- 見つけた障害に一時的な修正を加えて別の部分をデバッグする。

この目的に興味を持たれた方は、このあとの章も続けて読んで見て下さい。sdb を使っていて dbx を見た時に感じたのと同じ感動を、この gdb でも得られると思います。

1.1 ドキュメントで使用している言葉

このドキュメントでは、著者が普段から、なにげなく使っている言葉を使って記述しています。そのため、本来の使い方と意味が違う恐れがあるので、ここで言葉の意味を定義をしておきます。

【カレントディレクトリ】	デバッグ対象プログラムの実行ディレクトリを意味します。また、GDB の作業ディレクトリと同じディレクトリを意味します。
【コアファイル】	プログラムがコアダンプした時に作成される `‘core’` ファイルのことを意味します。
【コマンド】	このドキュメントで、単にコマンドと書いてあれば、GDB の動作を指定する GDB の内部コマンドを意味します。また、GDB の内部コマンドの名前は文中では~コマンドと表記して、UNIX コマンドの名前と区別できるようにしました。（例 `step` コマンド）
【コマンドファイル】	GDB の動作を記述したものが入っているファイルを意味します。
【コマンドライン】	GDB がプロンプト `‘(gdb)’` を表示してあなたの入力をまっている行のことを意味します。

【実行アドレス】	プログラムがメモリにロードされた時に振られるアドレスのことを意味します。
【作業用ディレクトリ】	GDB が作業用ファイルを作成するディレクトリを意味します。また、GDB の実行ディレクトリと同じです。
【スタックフレーム】	ここでは、話しが長くなるので簡単に書きますが、このマニュアルの中では、スタック、フレーム、スタックフレームという似たような名称が使われています。正式な言葉の定義は「Stack frames」(49ページ) を参照して頂くとして、フレームがC言語の関数と同意語だとだけ覚えて下さいね。
【ソースコード】	実行形式ファイルを作成するために記述された、C言語のテキストプログラムのことを意味します。
【ソースファイル】	プログラムのソースコードが入っているファイルのことを意味します。
【プログラム】	単に、プログラムと記述していれば、それは、デバッグ対象プログラムのことです。
...	この表記は、改行しないと紙面に収まらない実行例の行の後ろ少しを削ったよという意味で使う場合と、コマンドで繰り返し指定できる引数に対しての表記として使っている場合があります。

この他にも曖昧な表現があるかも知れませんが、石をなげたりしないでくださいね(^^;)。

1.2 キー操作の表現

キーの表記は、Emacs で使われている表記を使用しています。例えば `C-c` と書いてあれば、コントロールキーと‘`c`’キーを同時に押すことを意味します。また、`M-c` と書いてあれば、メタキーと‘`c`’キーを同時に押すことを意味します。ここで、メタキーのないキーボードの場合は、メタキーの変わりにエスケープキー `[ESC]` を押してから‘`c`’を押すという操作をして下さい。文中のいたる所に改行キーの空打ちという言葉がでてきますが、これはコマンドラインに文字を何も入力しないで改行キーだけを押すという意味で使っています。

Chapter 2

Getting In and Out of GDB

この章では、GDB の起動の仕方、終了の仕方について説明していますが !!、簡単に書いちゃえば、“`gdb`” と入力すれば起動されて、“`quit`” もしくは “`C-d`” と入力すれば終了します。

2.1 Invoking GDB

GDB のプログラム名は gdb です。GDB は一度起動されたら、あなたが終了コマンドを入力するまでの間、端末からのあなたの指示を待っています(^^)。

GDB は、引数を指定して起動することもできます。

多くの場合、次のようにデバッグするプログラム名を指定して起動します ：

```
gdb program
```

また、次のように書くと、コアファイル名の指定ができます ：

```
gdb program core
```

デバッグするプロセスのプロセス ID を指定して起動したい時は、次のように指定します ：

```
gdb program pid
```

ここで、*pid* に 1234 を指定したとします。その時、1234 という名前のコアファイルが、そこに存在したとしましょう。GDB は、その 1234 という指定は、コアファイルのことだと判断します。

GDB でどんなオプションが使えるかを知りたい時は、次のように打って下さい：

```
gdb -help
```

GDB は、オプションやコマンドラインからの一連の入力をファイルで指定することができます。ファイルを使う時は、'-x' オプションを使用します。ここで、指定するファイルの書き方については、「GDB Files」(83ページ) を参照して下さい。

2.1.1 Choosing file

GDBには、多くのオプションがあります。そして、それぞれのオプション名は、略記することもできます(オプションは‘-’で指定しても良いし‘--’を使っても構いません)。GDBにオプションを省略して引数を与えた場合、‘`-se`’を第一引数に‘`-c`’を第二引数に与えたのと同じ動きをします。

GDBのオプションは次の通りです:

`-symbols=`*file* `-s` *file*	シンボルテーブルの入ったファイルを指定する。
`-exec=`*file* `-e` *file*	デバッグする実行形式ファイルを指定する。
`-se` *file*	シンボリック情報を含む実行形式ファイルを指定する。また、指定した実行形式ファイルがデバッグ対象プログラムとなる。
`-core=`*file* `-c` *file*	コアファイルを指定する(core dumpで作られたファイル)。
`-c` *number*	プロセスIDを指定する。もし、指定したIDと同じ名前のコアファイルが存在する場合は、コアファイルの指定となる。
`-command=`*file* `-x` *file*	コマンドファイルを指定する。詳しくは、「Command files」(99ページ)を参照のこと。
`-directory=`*directory* `-d` *directory*	ソースファイルを検索するディレクトリパスを追加する。

`-m`	
`-mapped`	GDB 独自のシンボルマップを作成して動作を機敏にする。注意：このオプションは、オペレーティングシステムに依存するため、全てのシステムでサポートされているわけではない。 `mmap` システムコールがインプリメントされたシステムでは、GDB のメモリマップ機構が使えるかも知れない。GDB メモリマップ機構により作成されるファイルは、デバッグプログラム名に拡張子‘`.sym`’をつけた名前を用いる。このファイルは、カレントディレクトリに作成される。この機構は、GDB の内部シンボルテーブルの形式と同じであるため、GDB の動作を迅速にする。この機構は、リモートデバッグ時に使用することはできない。
`-r`	
`-readnow`	シンボルテーブルの入ったファイルを直ちに読む。このオプションを指定すると、現在は起動時間が遅くなるがそのうち早く起動できるようにする。

`-mapped` と`-readnow` オプションについては、「Commands to specify files」（83ページ）も合わせて読んで下さい。ここには、拡張子‘`.sym`’のファイルについての説明があります。GDB で簡単に‘`.sym`’ファイルを作らせるには、以下のようにコマンドを入力します：

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

2.1.2 Choosing modes

GDB 起動時に、次の起動モードを指定することができます。

`-nx`	
`-n`	GDB の環境設定ファイル‘`.gdbinit`’を読み込まずに起動する。「Command files」（99ページ）を参照のこと。
`-quiet`	
`-q`	GDB の紹介やコピーライトに関する表示を抑止する。

`-batch`	前節で紹介した ‘`-x`’ オプションと同じ。バッチモードで動作する。処理を終えた時に 0 以外の値を返す場合は、コマンドファイルの記述に誤りがある。 バッチモードは、GDB をフィルターとして使う場合に、役に立つかも知れない。例えば、あるプログラムを実行し終えた時に、その処理過程を解析させることにより、より有用なメッセージを出力させることができる。 （バッチモードで動作させている時は、GDB のコマンドラインインターフェイスは使えない。）
`-cd=`*directory*	GDB が使う作業用ディレクトリを指定する。カレントディレクトリもそこになる。
`-fullname` `-f`	GNU Emacs が使う GDB の作業用ファイル名を指定する。
`-b` *bps*	リモートデバッグ時の端末ボーレートを指定する（シリアル回線使用時のみ）。
`-tty=`*device*	標準入出力に使うデバイスを指定する。

2.2 Quitting GDB

GDB から抜けるには、`quit` コマンド（略記 `q`）か、end-of-file を示すコントロールキー（普通は `C-d`）を入力して下さい。インターラプトキー（普通は `C-c`）では、抜けれません。

既に起動されているプロセスをデバッグしたいときは、`attach` コマンドを用います。attach したプロセスのデバッグをやめるには、`detach` コマンドを用います。詳細については、「Debuggung an already-running process」（31ページ）を参照して下さい。

2.3 Shell commands

GDB の中から UNIX コマンドが使いたい時は、GDB をサスペンドさせなくても UNIX コマンドを実行することができます。

`shell` *command string*

環境変数 SHELL で設定されているシェルコマンドの配下で指定されたコマンドを実行する（設定されていない場合は `/usr/bin/sh`）。

`make` に関しては、`shell` コマンド使う必要はありません。以下のように記述することで同じ動作をします。

`make` *make-args*

'`shell make` *make-args*' と同じ動作をする。

Chapter 3

GDB Commands

GDB は、強力なコマンド補完機能を持ちます。コマンドラインから先頭の数文字を入力して、改行かタブを押せばそのコマンドは完全な形で処理されます。ただし、補完する候補が2つ以上存在する場合は、その候補を表示して知らせます。補完時の改行とタブの動作の違いは以下の通りです。改行を入力した場合は、補完したコマンドを実行します。タブを入力した場合は、補完したものを表示するだけで実行はしません。

3.1 Command syntax

GDB 内部コマンドは、コマンドラインから1行ずつ入力します。コマンドには、引数を取るものと取らないものがあります。例えば、引数を取らない `step` コマンドの場合、'`step 5`' という記述は、`step` コマンドを5回繰り返せという指示になります。

GDB は、いつでも補完を行ないますが、中には補完するものが固定のものもあります。例えば '`s`' は必ず '`step`' に補完されます。どのような略記が固定になっているかは、`help` コマンドの出力を参照して下さい。

コマンドラインで、改行を空打ちした場合、直前に実行したコマンドが繰り返し実行されます。しかし、`run` コマンドに関してのみ、この方法では繰り返せないように抑止してあります。これは、あなたが望まないプログラム実行の繰り返しを防ぐための措置です。`run` コマンドの他にも改行キーの空打ちで繰り返せないコマンドがいくつかあります。

`list` や `x` コマンドは、改行の空打ちで繰り返し実行できます。便利なことに、この繰り返しを行なうと、引数が自動的に更新されて見たい情報が次々と表示されます。

GDB において、改行の空打ちは、直前のコマンドを繰り返すという機能の他に、GDB のページャ機能の中では１行送りの意味になります。

行の先頭に ‘#’ を記述した場合、その行はコメント行として処理されます。これは、主にコマンドファイルを記述する上で役に立ちます。詳しくは「GDB Files」(63ページ)を参照して下さい。

3.2 Command completion

GDB では、入力途中の文字列を完全な GDB が認識できる文字列に補完する機能があります。認識できる文字列としては、コマンド、サブコマンド、プログラム中のシンボルがあります。

補完は、タブもしくは改行キーを入力した場合に起こります。タブでの補完の例を以下に示します：

```
(gdb) info bre[TAB]
          ↓
(gdb) info breakpoints
```

補完する候補が複数存在する場合は、以下のようになります：

```
(gdb) b make_[TAB]
 (ここで、ベルが鳴る！その時もう一度[TAB]を押すと)
make_a_section_from_file      make_environ
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_
```

3.3 Getting help

コマンドが分からなかったり、コマンドの使い方が分からなかったりした場合は、`help` コマンドで GDB に聞きましょう(^^)。ただし！GDB

は日本語が話せないので、英語で教えてくれます。

ヘルプはクラスとコマンドリストとコマンドとサブコマンドの４段階に分かれています。クラスを引数に指定することによって、GDB の機能分類と説明が見れます。引数に機能分類の一つを指定することによって、その機能分類に属するコマンドと説明が見れます。引数にコマンドを指定することによって、そのコマンドのサブコマンドリストと説明が見れます。引数にコマンドとサブコマンドを指定することによって、そのサブコマンドの説明が見れます[1]。

`help`

`h` クラスのヘルプを表示する。

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

running -- Running the program
stack -- Examining the stack
            :
obscure -- Obscure features
internals -- Maintenance commands
(gdb)
```

`help` *class*

そのクラスに属するコマンドリストのヘルプを表示する。

```
(gdb) help status
Status inquiries.

List of commands:

show -- Generic command for showing things about ...
info -- Generic command for showing things about ...
thread -- Use this command to switch between threads

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

---

[1] この時だけコマンドとサブコマンドの両方を指定する必要があります。

```
(gdb)
```

`help` *command*

そのコマンドの使い方を表示する。

```
(gdb) help info
Generic command for showing things about ...

List of info subcommands:

info set -- Show all GDB settings
info threads -- IDs of currently known threads
                 :
info watchpoints -- Synonym for "info breakpoints"
info breakpoints -- Status of user-settable ...

Type "help info" followed by info subcommand name ...
Command name abbreviations are allowed if ...
(gdb)
```

`help` *comannd subcommand*

サブコマンドの説明を表示する。

```
(gdb) help info set
Show all GDB settings.
(gdb)
```

Chapter 4

Running Programs Under GDB

GDB は、いろんな「しちゅえーしょん」でのデバッグが可能です。でもでも、その「しちゅえーしょん」に必要な情報がないと何もできない、おばかさんでもあります。情報の伝え方はいろいろ用意してありますので、好きな環境で思う存分 GDB を活用しましょう。

4.1 Compiling for debugging

いうまでもないことですが、シンボリックデバッグを GDB で行ないたいならば、デバッグ情報を実行形式ファイルに含めるようコンパイルすることが必要です。

通常、コンパイル時のオプションに '-g' を付けてコンパイルすれば、デバッグ情報が付加されます。

注意しなければいけないのは、'-O' と '-g' の両方のオプションを付けてコンパイルすると、多くのコンパイラでは、デバッグ情報を生成できません。このことは、あなたが、コードの最適化を指定した場合、デバッグ情報の生成を指示できないことを示します。ただし、GNU C コンパイラでは、両方のオプションが指定された場合、デバッグ情報を含んだコードの最適化を行ないます。

4.2 Starting your program

run r	GDB配下で、プログラムを起動する。起動されるプログラムは、GDB起動時に指定した実行形式ファイル。
引数	引数を `run` コマンドに与えた場合は、その引数をプログラムに渡す。
環境変数	環境変数は、シェルで設定されていたものを引き継ぐが、GDBの中でも設定、変更が可能である。
作業用ディレクトリ	作業用およびカレントディレクトリは `cd` コマンドで設定することができる。
標準入出力	通常GDBは標準入力と標準出力は同じデバイス装置を用いるが、リダイレクションを用いることで、これを変更することができる。

```
run > outfile
```

GDBでは、好きな場所でプログラムを停止させて、その位置での色々な情報を見たり、一時的に処理を変更することができます。こうすることにより、一つの障害を回避して別の障害を探すことができます。詳細は、「Stopping and Continuing」（35ページ）を参照して下さい。また、停止させた時に、変数の値を変更したり表示したりすることについての記述は「Examining Data」（61ページ）を参照して下さい。

デバッグ中に、再コンパイルなどをしてシンボル情報が更新された場合は、GDBに再度シンボル情報を読み込ませなくてはいけません：

```
symbol-file filename -readnow
```

これで、新しいシンボルテーブルが読み込まれます。詳細は、「GDB Files」（83ページ）を参照して下さい。

4.3 Your program's arguments

引数なしで、`run`コマンドを実行した場合は、直前に実行した`run`コマンドの引数を用います。また、`set args`コマンドを用いて引数を設定することもできます。

`set args`
　`run`コマンドがプログラムに渡す引数を設定する。

`show args`
　現在、設定されているプログラムに渡される引数を表示する。

4.4 Your program's environment

ここでは、環境変数とその値について解説します。GDBの中から環境変数を変更できることで、GDBに柔軟な動作をさせせることができるようになっています。

`path` *directory*
　実行形式ファイルのサーチパスを追加する。GDBの作業ディレクトリをサーチパスに入れたい時は、'`path $cwd`'と入力する。現在のディレクトリをサーチパスに追加したい時は'`path .`'と入力する。また、複数のディレクトリを同時に追加したい時は、'`:`'で区切って複数のディレクトリを指定する。
　注意 :通常のパスの設定との違いは、指定した値に置き換わるのではなく、パスリストに指定したディレクトリを追加することである。

`show path`
　現在、設定されているサーチパスを表示する。

`show environment` [*varname*]
　現在、設定されている環境変数とその値を表示する。環境変数名を指定した場合は、指定された環境変数の値を表示する。

```
(gdb) show environment TERM
TERM = kterm
(gdb)
```

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

環境変数を設定する '=' はあってもなくても良い。

```
(gdb) set env TERM = kterm
(gdb)
```

`unset envirnment` *varname*

環境変数を削除する。

4.5 Your program's working directory

GDB の中でプログラムを動作させる場合、カレントディレクトリを変更したい場合が起こります。そんな時、`cd` コマンドでカレントディレクトリを変更することができます。

`cd` *directory*

カレントディレクトリを *directory* に変更する。

`pwd` 現在のカレントディレクトリを表示する。

4.6 Your program's input and output

プログラムが使う標準入出力は、デフォルトでは、GDB が使用する端末 になっています。しかし、リダイレクションを用いて、デバッグプログラムが使う標準入出力を変更することができます。

`info terminal`

プログラムが使う、端末の情報を表示する。

リダイレクションの例 ：

```
run > outfile
```

こうすることで、プログラムの出力は、`outfile` に書き込まれます。また、別の方法でプログラムが使う標準入出力を切替えることができます。プログラムが使う端末を設定したい場合 ：

```
tty /dev/ttyb
```

この操作は、標準出力だけでなく標準入力も変更します。つまり、デバッグプログラムが使う端末を変更できるわけです(^^)。ただし、指定するデバイスにあなたのアクセス権が必要なことをお忘れなく。

4.7 Debugging an already-running process

`attach` *process-id*

実行中のプロセスをデバッグする。`attach` コマンドは、改行キーの空打ちで繰り返されない。実行中のプロセスは、UNIX コマンドの `ps` もしくは、`jobs -l` で確認できる。GDB の中から使う時は '`shell ps`' と入力する。

実行中のプロセスをデバッグする時は、`attach` コマンドを入力する前に `file` コマンドを用いてシンボルテーブルを読ませておくことを忘れないで下さいね(^^)。

`detach`

プロセスのデバッグを終えたい時は、`detach` コマンドを入力する。`detach` コマンドも、改行の空打ちで繰り返されれない。

`attach & detach` コマンドを使うことで、複数のプロセスのデバッグが可能になります。`run` コマンドを用いて、attach したプログラムをデバッグしている時に、あなたが GDB を抜けようとしたら、デバッグしているプロセスを殺してしまいます。デフォルトでは、GDB 終了時にデバッグしているプロセスを殺すかどうか、GDB があなたに問い合わせますが、それを煩わしいと感じるならば、`set confirm` コマンドで問い合わせなくなるようにできます。詳しくは「Optional warnings and messages」（95ページ）を参照して下さい。

4.8 Killing the child process

`kill`

GDB 配下の子プロセスを殺す。

このコマンドは、コアファイルを用いてプロセスをデバッグする時に役に立ちます。GDB は、プログラムが動作している最中はコアダンプするのを抑止してしまいます。そこで、一度プロセスを殺して、プログラムにコアを吐かせてから、コアファイルを読ませるわけですね(^^)。

その手順を以下に示します：

```
(gdb) kill
(gdb) shell program
Segmentation fault (core dumped)
```

```
(gdb) core-file core
(gdb)
```

いくつかの OS では、GDB 中でブレークポイントを設定していると、他にそのプログラムを起動できなくなります。そんな時、`kill` コマンドでプログラムを殺してしまえば、そのプログラムを起動することができるようになります。また、プログラムを再コンパイルした時 に、新しいプログラムをデバッグしたいのならば、`kill` コマンドで一度、現在デバッグしているプログラムを終了させる必要があります。

4.9 Additional process information

ある OS では、‘`/proc`’ に現在走っているプロセスの情報が置いてあります。そんな OS 上では、GDB がこの情報を用いるようにしてあれば `info proc` コマンドでデバッグプロセスに関する幾つかの情報を得ることができます。

`info proc`	デバッグしているプロセスに関する情報を表示する。
`info proc mappings`	デバッグしているプロセスがアクセス可能なアドレスや `rwx` に関する情報を表示する。
`info proc times`	デバッグしているプロセスの CPU 時間を表示する。
`info proc id`	デバッグしているプロセスの、プロセス ID、プロセスグループ ID、プロセスセッション ID を表示する。
`info proc status`	デバッグしているプロセスの状態を表示する。
`info proc all`	デバッグしているプロセスに関する全ての情報を表示する。

以下に‘`info proc`’コマンドの出力例を示します :

```
(gdb) info proc

Information for /proc/01008:

Process status flags:           PR_STOPPED PR_ISTOP PR_RLC PR_PCINVAL
Reason for stopping:            PR_SIGNALLED SIGTRAP (5)
Additional signal/fault info:   Signal 0 (0) sent by process 0, uid 0

(gdb)
```

4.10 Debugging programs with multiple threads

いくつかのオペレーティングシステムでは、一つのプログラムが一つ以上のスレッドを生成して動いても構いません。スレッドの精密な動きは、オペレーティングシステムにより異なりますが、大雑把に捉えるとスレッドは、マルチプロセスと大差ありません—それぞれのスレッドは、独自のアドレスを持ち一つのスタックになっていますし、たぶん、独自のメモリ空間を使用しているでしょう。

GDB が提供するマルチスレッドの機構は以下の通りです :

- 自動的に新しく生成されたスレッドを見つけます。
- ‘`thread` *threadno*’ コマンドでスレッドを切替えます。
- ‘`info threads` コマンドで、現在のスレッドの情報を表示します。
- スレッド特有のブレークポイントがあります。

注意 :スレッドが扱えるかどうかは、次のようにして確認することができます。スレッドが扱えない場合は、以下のような動作をするでしょう。

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

GDBのスレッドデバッグ機構は、プログラムが走っている間、あなたがスレッドを監視することを許します。—GDBがタスクをコントロールしていたとしても、一つのスレッドをデバッグできます。このスレッドのことをカレントスレッドと呼びます。そして、スレッド関連のデバッグコマンドは、このカレントスレッドに対して動作します。

GDBが新しいスレッドをプログラム動作中に検出したら、‘`[NEW` *systag*`]`’というメッセージを表示します。*systag*は、それぞれのシステムに依存するスレッド情報です。LynxOSでは、次のように表示されます。

```
[NEW process 35 thread 27]
```

GDBのスレッド機構は、それぞれのスレッドに独自の整数をスレッド番号として割り振ります。

`info thread`

現在、生成されている全てのスレッドの情報を表示する。カレントスレッドの先頭には‘`*`’を付加する。

```
(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=0x7ffffff8)
```

`thread` *threadno*

指定したスレッド番号のスレッドをデバッグする。スレッド番号は、`info threads`コマンドの出力の第一フィールドに表示される。

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

スレッド機構の動作の詳しいことは「Stopping and starting multi-thread programs」(48ページ)を参照して下さい。

マルチスレッドのwatchpointsに関しては、「Setting watchpoints」(38ページ)を参照して下さい。

Chapter 5

Stopping and Continuing

プログラムが全ての処理を終える前に、途中でプログラムを停止させて、その時点での状況を調査したい場合があります。ここでは、そんな状況の時に使う GDB のコマンドについて説明します。

`info program`
: デバッグ中のプログラム状態に関する情報を表示する。実行中かそうでないか、なぜ停止しているかなどの情報を表示する。

5.1 Breakpoints, watchpoints, and exceptions

GDB では、条件付きでブレークポイントを設定することができます。また、ソースの行番号、関数名、実行アドレスを指定しての設定も可能です。ブレークポイントは `break` コマンドで設定します。「Setting breakpoints」（36ページ）を参照して下さい。

ブレークポイントとは、プログラムの実行を途中で停止させる位置のことです。

watchpoint は、監視しているアドレスに変化が起こった時に処理を停止するブレークポイントです。詳細は「Setting watchpoints」（38ページ）を参照して下さい。

GDB は、ブレークポイントが設定されているとブレークポイントにシーケンシャルな番号を付けて管理しています。このように管理してい

るブレークポイントはブレークポイントリストと呼ばれるテーブルに登録されています。

これらのブレークポイントについて、enable（有効）, disable（無効）, delete（削除）という状態を設定することができます。有効なブレークポイントでは、処理は停止しますが、無効なブレークポイントは無視され処理は停止しません。削除されたブレークポイントは永久にブレークポイントリストから外されます。

また、プログラムがブレークポイントで停止した時に、自動的にある変数の値を表示させるように設定することもできます。詳細は「Automatic display」（65ページ）を参照して下さい。

5.2 Setting breakpoints

ブレークポイントは、`break`コマンドを使って設定します。また、GDBの内部変数として`$bpnum`という変数があります。この変数には、最後にコマンド引数に渡したブレークポイント番号が入っています。

`break` *function*

関数名を指定してブレークポイントを設定。

`break +`*offset*

`break -`*offset*

現在停止しているソースの行番号からみた行番号のオフセットで、ブレークポイントを設定する。

`break` *linenum*

ソースファイルの行番号を指定して、ブレークポイントを設定する。

`break` *filename:linenum*

ソースファイル名とそのファイルの行番号を指定してブレークポイントを設定する。

`break` *filename:function*

ソースファイル名とそのファイル内にある関数名を指定してブレークポイントを設定する。

`break` *address

実行アドレスを指定してブレークポイントを設定する。

`break`

引数なしの`break`コマンドは、次のフレーム（関数）にブレークポイントを設定する。設定される位置は、そのフレームから復帰するところである。`finish`コマンドは、そのフレームの処理から呼びだし元に復帰するところまで処理を続行して停止する。もし、現在のフレームが復帰値を持つならば終了時に復帰値を表示する。この機構は、ブレークポイントをわざわざ設定しなくても、処理を停止させることができるので便利である。

`break ...  if` *cond*

cond に条件を指定して処理を停止させる。

```
break function if a>0
```

という指定は、`function`にブレークポイントを設定するが、変数 a が 0 より大きい時にのみ、処理を停止させる。

`tbreak` *args*

一時的なブレークポイントを設定する。引数には`break`コマンドと同じものが使用できる。このコマンドで設定したブレークポイントは、一度、この位置で処理が停止したならば削除される。つまり、このコマンドを使えば、一度限り有効なブレークポイントを設定することができるのである。

`rbreak` regex

このコマンドは、指定したパターンにマッチする関数名全てにブレークポイントを設定する。

`info breakpoints [`*n*`]`

`info break [n]`

`info watchpoints [n]`

設定されたブレークポイントの情報を確認する。n を指定した場合は、n が情報を検索する最終アドレスの指定になる。GDB の内部変数 '`$_`' を用いた場合、コマンド `x` で設定したアドレスになる。

アドレスとは、メモリにロードされたプログラムのアドレスのことです。

$_には、最後にコマンドの引数に指定したアドレスが入っています。

ここで出力される各カラムの意味は次の通り ：

Breakpoint Numbers	
Type	`breakpoint` もしくは `watchpoint`。
Disposition	そのブレークポイントに止まった時に、削除するか無効にするか、そのまま保時されるかの情報。`del`，`dis`，`keep` のいずれかで表示される。
Enable or Disable	そのブレークポイントが有効か無効かを表示。
Address	そのブレークポイントのメモリ中でのアドレス。
What	そのブレークポイントのあるソースファイル名と行番号。

5.3 Setting watchpoints

watchpoint は、特定のアドレスの値が更新されるのを監視するブレークポイントです。*watchpoint* を設定すると、目に見えて処理速度が低下しますが、領域破壊などの障害を検出するために有効です。

`watch` *expr*

監視するアドレスを expr に指定する。ここには、直接メモリ中のアドレスを記述してもよいし、C 言語の記述形式でアドレスを指定してもよい（例 :`int i;` と宣言された変数のアドレスの場合`&c` と指定する。）。

`info watchpoints`

現在、設定されている `watchpoint` のリストを表示する。使い方は‘`info break`’コマンドと同じ。

注意 :マルチスレッドプログラムでは、watchpoint の機能に少しの制限があります。GDB は、一つのスレッドにしか watchpoint を設定することができません。watchpoint は、他のスレッドに処理を移さない場合は機能します。その例として、カレントスレッド以外のスレッドが、watchpoint を設定している領域を書き換えても、GDB は何も報告しません。

5.4 Delete breakpoints

ブレークポイントや *watchpoint* は、ある作業を終えた時に邪魔になる場合があります。そんな時、邪魔になったブレークポイントは削除しましょう。

`clear` や `delete` コマンドを用いると、ブレークポイントを削除することができます。`clear` コマンドは、現在、プログラムの何処で停止しているかによって、削除されるブレークポイントが左右されます。現在、プログラムが停止している位置で `clear` コマンドを入力すると、そのフレーム内にあるブレークポイントが全て削除されます。それに対して、現在停止している位置で `delete` コマンドを入力すると、設定されている全てのブレークポイントが削除されます。また、`delete` は、個々のブレークポイントが持つブレークポイント番号を指定してブレークポイントを削除することもできます。

`clear` 現在、デバッグ中のスタックフレーム内に存在するブレークポイントを削除する。

`clear` *function*

`clear` *filename:function*

指定した関数内にあるブレークポイントを全て削除する。

`clear` *linenum*

`clear` *filename:linenum*

指定されたソースファイルの行番号に設定されているブレークポイントを削除する。

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

指定されたブレークポイント番号を持つブレークポイントを削除する。

`set confirm` コマンドを入力すると、ブレークポイント削除時の確認メッセージを抑止することができます。

5.5 Disable breakpoints

`disable` コマンドを用いることで、ブレークポイントを削除するのではなく、ブレークポイントを一時的に無効にできます。また、無効にされたブレークポイントは、`enabele` コマンドを用いて、再び有効にすることができます。この機能は、一時的にだけ、ブレークポイントを外したい時に役立ちます。また、無効化されただけのブレークポイントは、ブレークポイントリストから削除されません。

ブレークポイントと *watchpoint* は、次のような４つの状態を持ちます：

- Enable（有効）。ブレークポイントに達した時にプログラムの処理を停止します。
- Disable（無効）。ブレークポイントに達してもブレークポイントは、プログラムの処理にたいして、なんの影響も与えません。
- Enable once。ブレークポイントは有効ですが、一度、プログラムの処理が停止した後は、そのブレークポイントを無効にします。
- Enable for delete。ブレークポイントは有効ですが、一度、プログラムの処理が停止した後は、そのブレークポイントを削除します。

ブレークポイントの状態を変化させるために、以下のコマンドが使用できます：

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

指定された番号のブレークポイントを無効にする。

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

指定された番号のブレークポイントを有効にする。

`enable [breakpoints] once` *bnums...*

指定された番号のブレークポイントを、次にそこでプログラムの処理が停止するまで有効にする。

`enable [breakpoints] delete` *bnums...*

指定された番号のブレークポイントを、次にそこでプログラムの処理が停止するまで有効にする。プログラムの処理が停止した後は、そのブレークポイントを削除する。

これらのコマンドを用いることで、`tbreak` ポイントで設定されたブレークポイントの状態も変化させることができます[1]。`enable once` コマンドで `tbreak` コマンドで設定したブレークポイントの状態を変化させると、無効になり‘Enable for delete’から抜けます。ただ単に `enable` コマンドを用いても‘Enable for delete’からは抜けませんので注意して下さい。

5.6 Break conditions

条件を指定したブレークポイントは、条件が満たされた時だけ処理を停止させます。条件を満たさない時に処理を停止させたい時は、条件式の前に‘`!`’を付けて条件を設定します。*watchpoint* は、ブレークポイントのような条件の記述ができませんが、`condition` コマンドによって、後から条件を設定することができます。

`watch if cond` という記述はできません。

`condition` *bnum expression*

指定した番号のブレークポイントに停止条件を設定。

---

[1] `until` コマンドによって自動的にに設定されたブレークポイントの状態は、これらのコマンドを用いても変更できません。

`condition` *bnum*

指定した番号のブレークポイントから停止条件を解除。

また、ブレークポイントで停止する条件に、このブレークポイントの何回目の通過時に停止しなさいという形の条件の設定もできます。この条件は以下のように与えます：

`igunore` *bnum count*

count で指定された回数目の通過時に処理を停止する。
この処理は、通過の度に *count* から1を減算していき、評価値が0になったら処理を停止する。ただし、指定されたブレークポイントの状態が無効状態のときは、処理を停止しない。つまり、評価値が0になった時、そのブレークポイントの状態や条件に従った処理を行なうのである。

この処理は以下の手順でも実現することができます：

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
3   breakpoint     keep y   0x000104ac in main at xx.c:6
6   breakpoint     dis  y   0x000104c0 in main at xx.c:7
(gdb) set $foo=5
(gdb) ignore 3 $foo
(gdb) condition 3 $foo--<=0
(gdb)
```

5.7 Breakpoint command lists

ブレークポイントに与える処理をシーケンシャルなコマンドのリストとして指定することができます。これは、起動時に読み込ませる`-batch`に与えるファイルに記述することもできますし、コマンドラインから入力することもできます。コマンドラインから`commands`コマンドを入力した時は、自動的にコマンドリストの編集モードに入り`end`を入力した時点で編集モードを抜けます。

`commands` [*bnum*]

...command-list...

`end` ブレークポイント番号で指定したブレークポイントの動作を指示する。

bnum を指定しない場合は、`$bnum` に入っているブレークポイント番号のブレークポイントの動作の指示になる。
`commands` を打ってすぐに `end` を入力した場合は、`$bnum` に入っているブレークポイント番号のコマンドリストをキャンセルする。
`commands` *bnum* を入力して、すぐに `end` を入力した場合は、指定されたブレークポイント番号のコマンドリストをキャンセルする。

コマンドリストの最初に `silent` を記述していると、通常のブレークポイントコマンドが出力するメッセージを抑止することができます。この機能を用いることで、そのブレークポイントに任意の出力メッセージを設定することができます。

では、ここで、独自のアクションをブレークポイントに与える例を一つ：

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

この設定では、`foo` という関数から復帰する位置に達したら 'x is 5' と表示して処理を停止させずに続行します。こんなかんじで GDB のコマンドを使うことで、ソース中にスナップ文を書かなくてもよくなりますね(^^)v。

また、１つの障害が見つかった時、その障害を仮修正して処理を続けるためには、次のように書いたりします：

```
break foo if x>0
commands
silent
set x = y + 4   ← ここで障害の修正をしている。
cont
end
```

コマンドリストの中では、C 言語のソースで使われている変数と同じ名前を使うことによって、C 言語の変数の中身を変更できるわけです(^^)。

昔のバージョンのGDB（4.9以前のバージョン）では、GDBが低レベルの端末入出力を行なうと、入力がペンディングするという障害がありました。この障害は、`commamds`コマンドの編集モードにとっては、致命的な問題になりかねませんでした。

5.8 “Cannot insert breakpoints”

あるオペレーティングシステムでは、プログラムにブレークポイントを設定すると、他から、そのプログラムが起動できなくなります。こういう状態に陥ったら、それがGDBの原因によるものかどうか試して下さい。

こういう状態を回避する方法として、以下の3つの方法があります：

1. ブレークポイントを全て削除するか、無効にする。
2. プログラムの名前を別の名前に変える。
3. プログラムをスタティックリンクでコンパイルしなおす。

こういう状態が起こる原因は、プログラムがダイナミックリンクライブラリを用いて処理を共有してしまうからなんですね。

5.9 Continuing and stepping

ブレークポイントで停止した処理を続行させるためには`continue`コマンド、もしくは、`step`コマンドを用います。`step`コマンドは、ソースコードの1行もしくは、アセンブラの1命令を実行するのに対して、`continue`コマンドは、次のブレークポイント（ブレークポイントが設定されていれば）もしくは、プログラムの最後まで処理を継続して実行します[2]。

`continue` [*count*]

`c` [*count*]

---

[2] もし、領域例外とかのシグナルのために処理が継続できない時は、‘`signal 0`’をコマンドラインから入力することで、処理を継続することができます。ただし、継続する前に、領域例外の原因を取り除きましょう。

`fg [`*count*`]` *count* を指定した場合は、ブレークポイントを *count* 回通過するまで、自動的に継続処理が行なわれる。つまり、*count* は、何回ブレークポイントを無視するかの指定回数である。また、`c`，`fg` は共に、`continue` コマンドの略記形式である。

フレーム処理中に、ソースコードの復帰箇所とは違う場所で、スタックフレームの上位のフレームに処理を戻したい時は、`return` コマンドを用います。また、スタックフレームと全く関係ない場所に、処理を移らせたい場合は `jump` コマンドを用います。これらのコマンドの詳細は「Altering Execution」(79ページ) を参照して下さい。

`step` コマンドを使うテクニックは、ここにバグがあるに違いないと思われる場所にブレークポイントを設定して、そこからソースを1行ずつ実行させて、プログラムの動きを観察することです。そうすることで、障害の検出を、素早く行なうことができます。

`step` 次のソース行まで処理を継続する。処理が終了したら、GDB に制御をもどす。
注意 :`step` コマンドは、デバッグ情報を含んでコンパイルされていないと、関数単位に処理を行なう。

`step [`*count*`]`
`step` コマンドを *count* 回数繰り返す。

`next [`*count*`]`
count 回数処理を繰り返す。`step` との違いは、処理中に関数があっても、その関数の処理を1ステップとして処理するところである。また、ループ中にある一連の処理1回も1ステップとして処理する。

ステップとは、`step` コマンドなどの1実行単位のことです。

`finish` 関数の処理が終了するまで、処理を続行する。その関数が復帰値をもつなら、その復帰値を表示してプログラムを停止する。

`until`

`u` ソースコードの現在行の処理に復帰するまで、処理を続行する。`next`との違いは、`next`はソース１行を常に１ステップとして扱うのに比べ、`until`は、ループ中の一連の処理を抜けるまで１ステップとして扱えるところである。`f(frame)`コマンドは、現在、処理中のソース行を知ることができる。`until`コマンドと`f`コマンドを用いた処理の例を以下に示す：

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

`until` *location*

`u` *location*

*location*で指定した場所に辿り着くまで処理を続行する。*location*には、`break`コマンドと同じ引数を指定することができる。

`stepi`

`si` アセンブラでの１命令を実行する。引数を与えた場合は、`step`コマンドと同じように繰り返し回数になる。

`nexti`

`ni` アセンブラでの１命令を実行する。実行を指定された命令が関数呼びだしならば、その関数から復帰するまで、処理を実行する。引数を与えた場合は、`step`コマンドと同じように繰り返し回数になる。

5.10 Signals

GDBでは、オペレーティングシステムで定義されているシグナルを処理することができます。

あるシグナル（例えば`SIGALRM`）は、プログラムが正常な処理を行なっていても発生するものですが、他のもの（例えば`SIGSEGV`）は、プログ

ラムにエラーがあった時に発生し、プログラムを直ちに終了させてしまいます。GDBでは、プログラムにこれらのシグナルをハンドリングさせる処理が記述されていなくても、これらのシグナルをハンドリングすることができます。

`info signals`

現在のシグナルに関する情報を出力する。

`handle` *signal keywords...*

シグナルを受けた時の動作を設定。
*keywords*に設定できる動作は以下の通り：

`nostop`	シグナルを受けたことを通知することが許可されていれば、シグナルを受けたことを通知するがプログラムの処理には影響を与えない。
`stop`	シグナルを受けたことを通知することが許可されていれば、シグナルを受けたことを通知してプログラムの処理を停止する。
`print`	シグナルを受けたことの通知を許可する。
`noprint`	シグナルを受けたことの通知を禁止する。
`pass`	デバッグ中のプログラムに、このシグナルを伝える。
`nopass`	デバッグ中のプログラムに、このシグナルを伝えない。

このように、`handle`コマンドの*keywords*に`pass`と`nopass`を使い分けて指定することで、プログラムに対して送られるシグナルをコントロールすることができるようになります。

また、`signal` コマンドを用いることで、好きなシグナルをプログラムに送ることもできます。`signal` コマンドの詳細は「Giving your program a signal」（81ページ）を参照して下さい。

5.11 Stopping and starting multi-thread programs

プログラムがマルチスレッドで動作している時は、スレッドを選んでブレークポイントを設定することができます。また、動作しているスレッドなら、どれでも選ぶことができます。

`break` *linespec* `thread` *threadno*

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

指定したスレッド番号のスレッドにブレークポイントを設定する。`thread` *threadno* を省略した場合は、全てのスレッドに対して、ブレークポイントが設定される。この使用例を以下に示す：

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

なんらかの理由で、プログラムが停止した場合、他のスレッドも、その時点で処理を停止します。これは、あなたがスレッドを切替えることができるようにするための処置です。

また、プログラムが再開されれば、停止した他のスレッドも動作を開始します。このことは、`step` コマンドなどを使って、ステップ実行をしていても変わりません。

GDB では、全てのスレッドを同時に1ステップづつ実行させることはできません。たとえカレントスレッドが1ステップづつ実行していたとしても、他のスレッドはその間に複数のステップを処理してしまいます。その上、ほかのスレッドは、処理が停止した時に奇麗にステートメントバウンダリで処理を停止するのではなく、適当な場所で処理を停止します。

どんな状況の時でも、ブレークポイントに入ったスレッドがあれば、プログラムはそこで処理を停止します。たとえ、あなたが、最初のスレッドの処理が終了したら停止するように要求していたとしてもです。

Chapter 6

Examinig the Stack

プログラムが処理の途中で止まってしまった時、まず始めに、そのプログラムがどこで止まっているか調べる必要があります。

プログラムが実行されて関数が呼ばれると、いつでもその関数の情報は、その関数が呼ばれた時に作成されたフレームにセーブされます。フレームには、関数に渡された引数や、ローカル変数などの情報が蓄えられています。GDB は、プログラムが使用しているスタックフレーム中のどんなフレームの情報でも選択して参照することができます。

また、GDB のいろんなコマンドを使えば、あなたが意識しなくても、いろんなフレームを使っています。

プログラムが停止した時、GDB は自動的にプログラムの停止しているフレームを処理の対象とします。そして、`frame` コマンドを入力する手間をなくしてくれます。

6.1 Stack frames

スタックとは、関数呼び出しによって積み上げられているフレームの状態のことをいいます。この状態にあるフレームの集合のことをスタックフレームと呼びます。それぞれのフレームには、関数に渡された引数や、ローカル変数[1]などの情報が蓄えられています。

---

[1] C において、その関数で参照できる変数のこと。GDB は、C において参照できる変数のみが、選択したフレームで参照できます。

プログラムが起動された時、フレームは main 関数のもの、ただ一つしかありません。このフレームのことを *initial frame* とか *outermost frame* とか呼びます。その後、関数が呼び出されて新しいフレームが作成されていきます。また、関数から復帰した時に、そのフレームはなくなってしまいます。呼び出された関数から、新たな関数呼び出しが続くと、フレームはどんどん積み上げられていきます。そして、最後に呼び出されて積まれた関数のフレームのことを *innermost frame* と呼びます。

フレームの大きさはマシンに依存します。

プログラムの中でスタックレームは、スタック領域と呼ばれる特別な場所に存在します。一つのフレームは多くのバイトから成り立ちそれぞれ独自のアドレスを持ちます。このフレームのアドレスは、プログラムが実行されている間は、フレームポインタレジスタに格納されています。

GDB では、最後に呼び出された関数（*innermorst frame*）から辿って、main 関数（*outermost frame*）までをトレースします。

あるCコンパイラでは、このスタックフレームを使用しないようにプログラムをコンパイルすることができます。そんなプログラムに対しても GDB はあたかもスタックフレームが用いられているように振舞います。

6.2 Backtraces

出力が長い場合は、複数行に分けて表示します。

バックトレースは、現在、プログラムがどの位置で停止しているかを、各フレームの簡易情報を一行ごとに表示することでして知らせてくれます。最初の行に表示されるのは *innermost frame* でフレーム番号は 0 です。その後は、*outermost frame* までのフレームを辿ってフレーム番号を振りながら各行に、そのフレームの情報を表示します。フレームが5つあるとすれば、*outermost frame* のフレーム番号は4番になります。

`backtrace`

`bt` 関数のトレースを表示する。この処理はいつでも `C-c` を入力することで中断できる。

`backtrace` *n*

`bt` *n* 　関数のトレースを *innermost frame* 方向に *n* 個表示する。

`backtrace -`*n*

`bt -`*n* 　関数のトレースを *outermost frame* 方向に *n* 個表示する。

`info stack`（略記は `info s`）コマンドは、`backtrace` の別名定義です。

‘`set print address off`’ というコマンドをコマンドラインから入力すれば、関数トレースを表示する時に関数のアドレスが表示されなくなります。

‘`backtrace 3`’ コマンドの出力例を以下に示します：

```
#0  m4_traceon (obs=0x24ebo, argc=1, argv=0x2b8c8)
    at builtin.c:933
#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...)
```

フレーム0番（*innermost frame*）の出力からプログラムは、builtin.c の933行目で停止していることが読みとれます。

6.3 Selecting a frame

GDBのほとんどのコマンドは、スタックフレームを調べるために用意されています。しかし、現在のフレームにあるローカル変数以外のローカル変数や他のフレームの情報を、フレームを変えずに表示させることはできません。そんな時、`frame` コマンドで別のフレームを選択すれば、その選択したフレームにある自動変数の値や、そのフレームに関する情報を参照することができるようになります。

グローバル変数はどこでも参照できます。

`frame` *n*

`f` *n* 　指定したフレーム番号を持つフレームを選択する。

`frame` *addr*	
`f` *addr*	指定したアドレスのフレームを選択する。
`up` *n*	*outermost frame*方向に *n* 個スタックを移動したフレームを選択する。
`down` *n*	*innermost frame*方向に *n* 個スタックを移動したフレームを選択する。
`up-silently` *n*	*outermost frame*方向に *n* 個スタックを移動したフレームを選択する。また、フレーム選択時のメッセージを表示しない。
`down-silentry` *n*	*innermost frame*方向に *n* 個スタックを移動したフレームを選択する。また、フレーム選択時のメッセージを表示しない。

`up` や `down` コマンドに引数を与えない時は、それぞれの方向に 1 スタック移動したフレームを選択したことになります。

スタックフレームに関する情報は、以下のコマンドで参照できます：

`frame`	
`f`	現在のフレームの概要を表示する。
`info frame`	
`info f`	現在のフレームに関する冗長な情報を表示する。
`info frame` *addr*	
`info f` *addr*	指定したアドレスのフレームの冗長な情報を表示する。

`info args` 現在のフレームの引数を表示する。

`info local`

現在のフレームのローカル変数に関する情報を表示する。

Chapter 7

Examining Source File

GDBでは、シンボル情報を付加してコンパイルしたプログラムならば、ソースファイルの参照をすることができます。表示されるソースコードは、特に指定しなくても自動的に、現在処理中のソース行、もしくは、現在処理しているフレームのソース中の関数の位置を中心とした所に、位置づけられます。

もし、あなたが'GNU Emacs'からGDBを使っているなら、Emacsのソースコード参照機能で、ソースコードの、現在処理している位置を、常に自動的に表示してくれます。詳細については「Using GDB under GNU Emacs」（103ページ）を参照して下さい。

7.1 Print source lines

ソースコードを出力させるためには、`list`（略記は`l`）コマンドを用います。表示位置は幾つかの方法で指定することができます。

`list` *linenum*

指定された行番号を中心とした位置のソースコードを表示する。

`list` *function*

指定された関数の先頭を中心とした位置のソースコードを表示する。

`list` 直前に表示したソースに続く部分を表示する。

`list -` 直前に表示したソースの上の部分を表示する。

GDB では、`list` コマンドで表示するソースコードの行数を '`set listsize`' コマンドを用いて変更することができます (デフォルトは 10 行)。

`set linesize` *count*
: `list` コマンドで出力するソースの行数を変更する。

`show linesize`
: 現在、設定されている `list` コマンドで出力するソースの行数を表示する。

`list` コマンドでソースコードを表示したあとは、改行キーの空打ちで、それに続くソースコードを表示させることができます。また、'`list -`' コマンドでソースを表示した後ならば、改行キーの空打ちでソースコードを遡って表示させていくことができます。

`list` コマンドには、もっと別な方法でソースコードの表示を指示することができます。以下に、その方法を示します：

`list` *linespec*
: *linespec* に従ってソースコードを表示する。

`list` *first,end*
: 指定された部分のソースコードを表示する。

`list +` 直前に表示したソースコードの次の部分を表示する。

`list -` 直前に表示したソースコードの前の部分を表示する。

`list` 直前に表示したソースコードを再度表示する。

以下に *linespec* に指定できるものを示します：

number 指定した行を中心とした位置のソースコードを表示する。

+offset 直前に表示したソースコードの最終行から *offset* 進めた行の位置を先頭行にしてソースコードを表示する。

-offset 直前に表示したソースコードの最終行から前に *offset* 行の位置を先頭行にしてソースコードを表示する。

filename:number
指定したファイルの行番号を中心としたソースコードを表示する。

function 指定した関数を中心としたソースコードを表示する。

filename:function
指定したファイルの関数を中心としたソースコードを表示する。

**address* 指定したアドレスに相当したソースコードの位置を中心としたソースコードを表示する。

7.2 Searching souce file

GDB では、ソース中をパターン検索するために 2 つのコマンドを用意しています。

`forward-search` *regexp*

`search` *regexp*
ソースコードの現在行から最終行に向かって、指定されたパターンに一致する行を検索して最初に一致した行を表示する。

`reverse-search` *regexp*
ソースコードの現在行から先頭行に向かって、指定されたパターンに一致する行を検索して最初に一致した行を表示する。

7.3 Specifying source directories

プログラムのデバッグをする時に、そのプログラムのソースコードが作業用ディレクトリに存在しない場合があります。そんな時でも、ソースコードを GDB が参照できるように、ソースサーチパスを設定することができます。GDB は、シンボリック情報を元に必要なソースコードが見つかるまでソースサーチパスに登録されているディレクトリの中を検索してくれます（こんなデバッガははじめてだ(;_;)）。

あなたが GDB を起動したばかりの時は、ソースサーチパスは空っぽです。`directory` コマンドを用いてソースサーチパスを追加してあげましょう(^^)。

`directory` *directoryname...*

ソースを検索するディレクトリをソースサーチパスに追加。もし、一度に複数のディレクトリを追加する場合は、各ディレクトリを ':' で区切って記述する。変数 '`$cdir`'[1] には、もし、全てのソースが 1 つのディレクトリでコンパイルされているとしたら、そのディレクトリが入っている。また、'`$cwd`' と '.' は GDB の中では別のディレクトリを意味する。'`$cwd`' は、GDB の作業用ディレクトリを意味し、'.' は、GDB を起動したディレクトリを意味する[2]

`directory` ソース GDB パスを初期化する。

`show directories`

設定されている、ソースサーチパスを表示する。

もし、ソースサーチパス中に違うバージョンの同じ名前のファイルが混在してしまった場合、GDB は混乱してしまいます。そんな時のために：

1. `directory` コマンドを引数なしで実行して、ソースサーチパスを初期化しましょう。

---

[1] `$cdir` 変数には、ソース参照ディレクトリが 1 つだけ設定できます。あなたが設定していないなら、GDB を起動時した時に自動的に GDB の実行ディレクトリが設定されています。

[2] まとめると、'`$cdir` は、ソースファイルのあるディレクトリで、'$cwd' がカレントディレクトリで、'.' が GDB の起動ディレクトリとなります。

2. `directory` コマンドを用いて、正しいソースの入っているディレクトリをソースサーチパスに追加しましょう。

おいおい、どこが対策やねん(^^;)。しかし、これに勝る手段はないかも知れない。。。

7.4 Source and machine code

‘`info line`’ コマンドを用いることで、そのソースコードがメモリ中のどこにマップされているか調べることができます。そして、その情報を元に、そのソースコードのアセンブラコードを出力させることができます。では、ソースコードの情報からアセンブラコードを引き出して見ましょう。

```
(gdb) info line sslalloc
Line 12 of ‘‘xx.c’’ starts at address 0x104e0 <sslalloc>
   and ends at 0x104f0 <sslalloc+16>.
  (ここで得た情報がこの関数のマップされている位置です。)
(gdb) disas 0x104e0 0x10504
Dump of assembler code from 0x104e0 to 0x10504:
0x104e0 <sslalloc>:     sethi  %hi(0xfffffc00), %g1
0x104e4 <sslalloc+4>:   add  0x3a0, %g1, %g1    ! 0xfffffa0 ...
0x104e8 <sslalloc+8>:   save  %sp, %g1, %sp
0x104ec <sslalloc+12>:  st  %i0, [ %fp + 0x44 ]
0x104f0 <sslalloc+16>:  ld  [ %fp + 0x44 ], %o0
0x104f4 <sslalloc+20>:  call  0x2058c <_lib_version+65648>
0x104f8 <sslalloc+24>:  nop
0x104fc <sslalloc+28>:  b  0x10504 <sslalloc+36>
0x10500 <sslalloc+32>:  nop
End of assembler dump.
  (以上で目的の関数のディスアセンブルが完了です。)
```

`disassemble`

この関数の引数には、アセンブラを出力するアドレスの範囲を指定する。第一引数が開始アドレスで、第二引数が終了アドレスである。指定するアドレスの情報はは、‘`info line` *linspec*’ コマンドで得ることができる。

Chapter 8

Examining Data

通常、変数に入っている値を参照するには`print`（略記は`p`）コマンドを用います。`print`コマンドは'`inspect`'と書くこともできます。また、変数を参照する時は、C言語で用いられる記述形式を、そのまま使うことができます。

`print` *exp*

`print` /*f* *exp*

変数の値を参照する。*exp*には、参照したい変数をC言語での記述形式で指定する。'/*f*'は、変数の表示形式を指定する。GDBの内部変数を参照する時は、`$foo`の形で`foo`を参照したい変数名に置き換える。

`print`

`print` /*f*

直前に`print`コマンドで参照した変数を再度参照する。

`x`コマンドは、よりローレベルな変数の参照方法を提供してくれます。`x`コマンドの使用例としては、メモリ中のアドレスを表示したりします。`x`コマンドは、また、独特な出力フォーマットで表示させることができます。`x`コマンドの詳細については『Examining memory』（64ページ）を参照して下さい。

変数の型を知るためには、`ptype`コマンドを使用します。このコマンドによって、その変数の構造体のメンバや型を表示させることができま

す。`ptype` コマンドの詳細については「Examining the Symbol Table」(75ページ)を参照して下さい。

8.1 Expressions

`print` コマンドは、C言語表記で記された変数の値を参照することができます。また、条件式の結果や関数呼びだし、キャストの結果、文字定数にいたるまで表示させることができますが、`#define` で定義されたマクロ定数の値だけは表示させることができません。

C言語の表記が色々使えるのは、C言語がそれほどに広く浸透しているからなのですが、ここでは、GDB独特の表記方法を用いた変数の参照方法を示しましょう：

`@`
: '`@`' は配列を参照する表記方法である。詳細については、「Artificial arrays」(63ページ)を参照のこと。

`::`
: '::' は、ファイルや関数を指定して、そこにある変数の値を参照する。詳細については、「Program variables」(62ページ)を参照のこと。

{*type*} *addr*
: *addr* で指定したメモリ領域にあるデータを *type* で指定した型のデータとして値を表示する。

8.2 Program variables

通常、変数は、変数の有効範囲を越えたフレームからは、参照することができません。また、あるファイルに閉じたスタティック変数についても、他のソースファイルにあるフレームを選択しているときは参照することができません。しかし、'::' を用いるこによって、そんな変数の値を参照することができます。参照時の引数の記述方法は以下のようになります：

file::variable
function::variable

8.3 Artificial Aarrays

この表示方法は、同じ構造をもって、メモリ中にならんでいる値を表示したり、ダイナミックにアロケートされた領域の中身を表示させたりするのに役立ちます。ここでは、オペレート文字に ‘`@`’ を用います。例えばソースコード中に以下のようなステートメントがあったとします：

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

ここで、`arry`（配列）の中身を見る場合は以下のように記述します：

```
print *arry@len
```

たまにこの方法で表示させると、表示が煩わしい場合があります。そんな時は、次のように記述すると良いでしょう：

```
set $i = 0
print dtab[$i++]->fv
RET
RET
...
```

こうすることで変数の中身を 1 つづつ確認していくことができますね。

8.4 Output formats

`print` コマンドのデフォルトの表示形式は、表示しようとする変数の型に依存します。GDB では、変数をいろんな形式で表示できるようにフォーマットを指定することができます。指定できるフォーマットは以下の通りです：

`x`	データを 16 進数で表示する。
`d`	データを符合付き 10 進数で表示する。
`u`	データを 10 進数で表示する。
`o`	データを 8 進数で表示する。
`t`	データを 2 進数で表示する。

a	データをアドレスとして 16 進数で表示する。また、近くにあるシンボルからのオフセット表記で表示する。
c	データをキャラクタ１文字として表示する。
f	データを浮動小数点フォーマットで表示する。

では、ここでプログラムカウンタレジスタを 16 進数で表示させてみましょう ：

```
（gdb） p/x $pc
$9 = 0x10490
（gdb）
```

`p` とスラッシュ‘/’ の間はスペースを入れないことに注意して下さいね。

最後に表示させた変数を繰り返し表示させるためには、改行キーを空打ちします。`print` コマンドを引数なしで入力しても、最後に指定したフォーマットは有効ではありません。別の変数を同じフォーマットで表示させる場合は、フォーマットを再度指定して下さいね。

8.5 Examining memory

データの参照をするために、`print` と違ったフォーマットが使いたい時は、`x` コマンドを使用します。`x` コマンドの使い方は次の通りです ：

`x/`*nfu addr*

`x` *addr*

x	*n,f,u* はメモリ内容を様々な形式で表示するための全てのオプションパラメータである。*addr* には、表示を開始するアドレスを指定する。もし、デフォルトのフォーマットを用いるのならば、スラッシュ‘/’ 以下の部分は指定する必要がない。

オプションパラメータの意味を以下に示します ：

n	表示の繰り返し回数を指定する。指定できるのは整数値。
f	表示フォーマットを指定する。‘`s`’ は文字列。‘`i`’ はアセンブラ１命令。‘`x`’ は 16 進数。デフォルトのフォーマットは ‘`x`’。

u データの長さを指定する。

- `b` 1バイト。
- `h` 2バイト。
- `w` 4バイト。
- `g` 8バイト。

addr 表示を開始するアドレスを指定する。アドレスを省略した場合は、他のコマンドで使用した最後のアドレスを使う。

$_に入っているアドレスが用いられます。

'`x/4xw $sp`' という記述は、スタックポインタレジスタに入っているアドレスを開始アドレスとして4バイト区切りに16進数で4つのデータを表示するという意味になります。ここで '`4xw`' の部分は、'`4wx`' と記述しても構いませんが '`wx4`' と記述した場合は動作しません。

また、GDBでは、組み込み変数として`$_`という最後に使用したアドレスを覚えている変数と`$__`というその変数の中身を覚えている変数が使用できます。つまり、`$_`は`$__`のポインタなんですね。

`x`コマンドを使用した後に、同じコマンドを使いたい時は、改行キーを空打ちします。この時は、直前に使用したオプションパラメータを覚えていますが、再度`x`コマンドだけを入力しても、直前に使用したオプションパラメータの指定にはなりません。これは、`print`コマンドの時と同じですね(^^)。

8.6 Automatic display

プログラムが停止する時のメッセージの表示形式や特定の変数の値を表示する形式を指定したい場合、GDBでは`display`コマンドを用いて、その形式を指定することができます。

`display` *exp*

プログラムが停止した時に表示するメッセージの形式を変更する。*exp*の指定は「Expressions」(62ページ)を参照のこと。

このフォーマットは`print`コマンドで指定するのと同じ形式が使用できる。

`display` */fmt exp*

auto-display list[1]に指定した変数用の新しい出力フォーマット

---

[1] *auto-display list*とは、`print`コマンドや`x`コマンドで表示する形式をユーザが場合に登録してあるテーブルのことです。

を追加する。フォーマットの詳細については、「Output formats」（63ページ）を参照のこと。

`display /`*fmt addr*

このフォーマットはxコマンドでしていするのと同じ形式が使用できる。

*auto-display list*に指定したアドレス用の新しい出力フォーマットを追加する。フォーマットの詳細については、「Examining memory」（63ページ）を参照のこと。

ここで設定した、*auto-display list*には、それぞれ固有の番号が振られます。その番号を指定して、追加したリストから、それを削除することができます。

`undisplay` *dnums...*

`delete display` *dnums...*

指定した番号の表示フォーマットを削除する。番号はブランクで区切って、一度に複数指定することが可能である。

`disable display` *dnums...*

指定した番号の表示フォーマットを無効にする。`display enable`コマンドを使用することで、再度、有効にすることができる。

`enable display` *dnums...*

指定した番号の表示フォーマットを有効にする。

`display` プログラムが停止した時に表示するメッセージの形式をデフォルトに戻す。

`info display`

*Automatic display*に関する情報を表示する。

8.7 Print setting

GDBでは、構造体、配列、シンボルなどの表示形式を選ぶことができます。以下に、その設定内容を示します：

`set print address on`

変数のアドレス表示を行なう。

on の時の出力 ：

```
(gdb) f
#0  wd_write (wp=0x2a148, data=0x27070, attr=0) at p_window.c:64
64              x = wp->x + data->x + 1;
```

`set print address off`

変数のアドレス表示を行なわない。

off の時の出力

```
(gdb) f
#0  wd_write (wp=, data=, attr=0) at p_window.c:64
64              x = wp->x + data->x + 1;
```

`show print address`

'print address' の on/off を表示する。

`set print symbol-filename on`

行番号や関数名の表示を行なう。

`set print symbol-filename off`

行番号や関数名の表示を行なわない。

`show print symbol-filename`

'print symbol-filename' の on/off を表示する。

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

max-offset で指定したアドレスより小さいアドレスについてのシンボル情報を参照する。指定したアドレスより大きいアドレスのシンボル情報は参照しない。*max-offset* に 0 が指定された場合は、全てのアドレスについてシンボル情報を参照する。

`show print max-symbolic-offset`

設定されている *max-offset* の値を表示する。

`set print fast-symbolic-addr off`

なんらかのシンボル情報が表示された時、同時に全てのシンボル情報を検索しておく。

`set print fast-symbolic-addr on`

なんらかのシンボル情報が表示された時、それに関する最低限のシンボル情報だけ検索する。デフォルトはこちら。

`show fast-symbolic-addr`

‘`fast-symbolic-addr`’ の `on/off` を表示する。

`set print array`

`set print arry on`

配列の表示行数を増やして表示を見やすくする。デフォルトは `off`。

`set print arry off`

配列の表示をコンパクトにする。

`show print array`

‘`print array`’ の `on/off` を表示する。

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

number-of-elements のところに表示する配列の最大要素数を指定する。

*number-of-elements*に 0 を指定した場合はすべての要素を表示する。

`show print elements`

設定されている、*number-of-elements*の値を表示する。

`set print pretty on`

構造体の表示行を増やして見やすくする。デフォルトは `off`。`on` の時の出力

```
(gdb) print *wp
$6 = {
  x = 2,
  y = 2,
  l = 32,
  h = 16,
  cx = 34,
  cy = 18
}
```

`set print pretty off`

構造体の表示をコンパクトにする。
`off` の時の出力

```
(gdb) print *wp
$7 = {x = 2, y = 2, l = 32, h = 16, cx = 34, cy = 18}
```

`show print pretty`

‘`print pretty`’ の `on/off` を表示する。

`set print seven-bit on`

7bit キャラクタの表示モード。

`set print seven-bit off`

8bit キャラクタの表示モード。デフォルトはこちら。

`show print seven-bit`

‘`print seven-bit`’ の `on/off` を表示する。

`set print union on`

union の表示をする。デフォルトはこちら。

`set print union off`

union の表示をしなくする。

`show print union`

‘`print union`’ の on/off を表示する。

`set print object`

`set print object on`

二重定義された関数の型を仮想関数テーブルを用いて表示する。

`set print object off`

二重定義された関数の型を仮想関数テーブルを用いずに表示する。デフォルトはこちら。

`show print object`

‘`print object`’ の on/off を表示する。

8.8 Value history

`print` コマンドは、別の値を表示しようとした場合、前回表示した値をヒストリ変数に保存します。この情報は、シンボルテーブルの内容が、シンボルテーブルの再読み込みによって変更されるまで保持されます。

`print` コマンドで値を表示した時、`print` コマンドは、その表示した変数の値を保持したヒストリ番号を表示します。こんな感じで、、、

```
(gdb) print p
$1 = 0x800ae674 "@\001\005G\222\020"
^^^
```

この `$` に続く整数がヒストリ番号だよ (^^)。
そいでもって、このヒストリ変数を `print` コマンドで表示すると、、、

```
(gdb) print $1
$2 = 0x800ae674 "@\001\005G\222\020"
^^^
```

このようにヒストリ変数の値も上昇してヒストリ変数は保持されていくわけです。

また、ヒストリ変数は、‘$’ だけを記述した場合は、‘$’ が最新のヒストリ変数に一致するという特徴を持っています。このことをうまく利用すると、リスト構造を持ったテーブルを簡単に辿ることができます。では、そのやり方を以下に示しましょう。

```
print *$.next
```

という１行をコマンドラインから入力します。その後は、改行を空打ちするだけであら不思議、どんどんリスト構造を辿っていきますよね(^^)。

ヒストリ変数に登録されている変数と値の一覧を見るには‘show values’と入力して下さい。また、ヒストリ変数が沢山たまっている時は、‘show values *n*’ コマンドを使って何番目のヒストリ変数を中心にして、一覧を表示するか指定します。指定したヒストリ番号を中心に 10 個のヒストリ変数のリストが表示される筈です。その後の部分を見たい場合は、‘show values +’ という具合に打ちます。そうすると、つづく部分のヒストリ変数が表示されます。後は、改行の空打ちで次々とヒストリ変数がなくなるまで一覧がでてきます。

8.9 Convenience variables

GDB を動作させていると、知らないまに幾つかの変数が作成されて使用されています。このような変数のことをコンビニエンス変数と呼びます。前節で紹介したヒストリ変数も、その一例です。この他には、ブレークポイントのところででてきた`$bpnum` や `print` コマンドででてきた`$_`, `$__`などが、その仲間になります。

また、GDB では、コンビニエンス変数を簡単に作成して使用することができます。その例を一つ ：

```
set $foo = 1
```

この１文で foo と言う変数を作成して、値１を代入するという処理になります。このように、GDB で使う変数は、頭に ‘$’ という文字を書いた英数字文字列を使うことで、簡単に利用できます。

コンビニエンス変数をどれくらい使っているかは、‘show convenience’コマンドを用いることで確認することができますよ(^^)。

8.10 Register

レジスタに入っている値を簡単に参照する方法は、GDB のレジスタ変数を用いる他に ‘`info register`’ というようなコマンドを用いる方法があります。レジスタ参照コマンドは以下の通り：

`info register`
: 全てのレジスタと現在選択しているフレームが使用しているフローティングレジスタの名前と値を表示する。

`info all-register`
: 全てのレジスタと全てのフローティングレジスタの名前と値を表示する。

`info register` *regname...*
: 指定した名前のレジスタの値を表示する。

また、GDB のレジスタ変数を用いると次のようになります。

`p/x $pc`	(プログラムカウンタの表示)
`p/x $sp`	(スタックポインタの表示)
`p/x $ps`	(プロセッサステータスの表示)

このレジスタ変数を用いた表示には、`print` コマンドだけでなく、もちろん `x` コマンドを用いることができます。

8.11 Floating point hardware

これは、マシン依存の機能ですが、メインフレームに関しては、‘`info float`’ コマンドを用いて、フローティングポインタレジスタ装置に関する情報を表示させることができます。

Chapter 9

Using GDB with Different Languages

　未稿

　この部分は、C言語以外のコンパイラ言語に関する記述であるため削除します。

Chapter 10

Examining the Symbol Table

ここでは、変数名や関数や型などのようなシンボルに対するコマンドについての解説を行ないます。GDBは起動時にプログラムのテキストセグメントからシンボルに関する情報を読み出しています。

このシンボルに関するコマンドには以下のようなものがあります ：

`info address` *symbol*

指定されたシンボルが何処にストアされているかの情報を表示する。レジスタ変数の場合はどのレジスタが保持しているかを表示する。また、ローカル変数の場合は、フレーム内のオフセット値で表示する。
`print` コマンドを用いて‘`print &symbol`’と入力する時と対比すると、‘`print &symbol`’はレジスタ変数についての情報を表示できない。また、表示するアドレスもオフセットではなく、変数の存在する実アドレスである。などのような、相違点がある。

`whatis` *exp*

指定した変数名の型を表示する。

`whatis` ‘`$`’（最新のヒストリ変数）の型を表示する。

`ptype` *typename*

指定した型の構造を表示する。型の記述はCの記述形式に習って、構造体なら'`struct` *struct-tag*'、共用体なら'`union` *union-tag*'、列挙型なら'`enum` *enum-tag*' の形式で指定する。

`ptype` *exp*

`ptype` 指定した変数名の型を表示する。`whatis` コマンドとの違いは、`whatis` コマンドは、構造体のメンバまで表示しないが、こちらは、構造体のメンバも合わせて表示するところにある。引数を省略した場合は、'`$`'（最新のヒストリ変数）の型を表示する。

`info types` *regexp*

`info types`

パターンにマッチする全ての変数名の型を表示する。動作は `whatis` コマンドに似ている。また、引数が指定されない場合、全てのソース中に定義されている全ての変数の型を表示する。

`info sorce`

現在、プログラムが実行しているソースコードのあるソースファイルに関する情報を表示する。ソースファイルの行数であるとか、記述言語の種類であるとか、ソースファイルのファイル名とかを表示する。

`info sources`

シンボルテーブルに含まれる全てのソースファイルの情報を表示する。既に、そのファイルのシンボルは読み込んでいるとか、処理がそのファイルに達した時に、そのファイルのシンボル情報を読むとかいった情報も表示する。

`info functions`

定義されている関数名や関数の型というような情報を表示する。

`info functions` *regexp*

パターンにマッチする関数の情報を表示する。ここで、'`info fun step`' という指定は、'`step`' という文字列を含む関数の情報を全て表示するのに対して '`info fun ^step`' という指定は、関数名の頭が '`step`' という文字列に一致する関数の情報のみを表示する。

`info variables`

全ての変数名および、その変数の型をローカル変数も含めて表示する。

`info variables` *regexp*

パターンにマッチする全ての変数名を表示する。ただし、ローカル変数は検索しない。

では、`ptype` コマンドで `typedef` によって定義されている、`CHARACTER` という構造体の構造を調べる例を示しましょう :

```
(gdb) ptype CHARACTER
type = struct s_character {
    struct s_defence defence;
    struct s_profile profile;
    struct s_batting batting;
    struct s_pitching *pitching;
}
(gdb) ptype struct s_defence
type = struct s_defence {
    struct s_position main;
    struct s_position sub;
}
(gdb) ptype struct s_position
type = struct s_position {
    char position;
    char level;
    char error;
    char throwing;
```

```
                }
```

Chapter 11

Altering Execution

以前、プログラム中で障害を見つけた時に、一時的に、その障害を修正して残りの処理を続けさせたいと思ったことがありませんか ?GDBに習熟すれば、そんなことも可能になります。

領域例外のエラーでさえも、正しい領域をプログラムに設定して、初期化シグナルを送ってやることで、プログラムは処理を続行することができます。また、関数呼び出しの復帰値を正しい値に修正することさえも可能です。

11.1 Assignment to variables

変数の値を変更したいならば、次のように記述します。

```
print x=4
```

このコマンドは、変数`x`に4を代入して、4という数字を表示します。また、この4という値はヒストリ変数に登録されます。

もし、あなたが4という数字が表示されることを嫌うなら、次のような書き方をすることもできます。この時、4という値は、ヒストリ変数に登録されません。

```
set var x=4
```

ここで注意しなくてはいけないのは、`set x=4`と`var`を抜かして記述してしまうと、構文エラーになってしまうことです。

また、あるメモリ中のアドレスに値を代入したい場合は、次のような記述をして下さい。

```
set {int}0x83040 = 4
```

こういう記述方法を取ることで、メモリ中に直かに値を書き込むことができます。

11.2 Continuing at a different address

もし、プログラムが停止している時に、全く別の場所からプログラムを継続させたい場合は `continue` コマンドではなく以下のコマンドを使用して下さい：

`jump` *linespec*

実行を *linespec* で指定した位置から再開する。*linespec* の指定方法は「Printing source lines」(55ページ) を参照のこと。`jump` コマンドは、フレーム、スタックポインタ、メモリロケーション、レジスタの値、プログラムカウンタを一切変更しない。`jump` コマンドを用いて、別のスタックフレームに処理を飛ばすことはできない。もし、現在のスタックフレームに存在しないフレームへ処理を飛ばそうとし場合は、GDB が処理の実行を確認してくる。これは、飛び先のフレームと現在のフレームが異なったパターンの引数やローカル変数を同じものであると混乱して、思いがけない結果を起こすのを警告するためである。ただし、アセンブラを熱心に勉強すれば、思いがけない動作も思いがけない動作ではなくなるであろう。

`jump` **address*

指定されたアドレスから処理を再開する。

また、`jump` コマンドを使わなくても、プログラムカウンタ `$pc` に新しい値を設定することで同じ処理をさせることができます。`jump` コマンドとの違いは、レジスタに値を入れた時に、処理が再開されないことです。処理を再開させるためには、レジスタの値を更新した後で `continue` コマンドを入力して下さい。プログラムカウンタに値を代入するには、以下のように記述します。

```
set $pc = 0x485
```

11.3 Giving your program a signal

`signal` *signalnum*

プログラムが停止した時に、シグナルをプログラムに送ることができる。送るシグナル番号は *signalnum* に指定する。もし、プログラムがシグナルの発生により何回も停止してしまう状況にあったとしても、'`signal 0`' と `continue` を繰り返し実行することで処理を継続させることができる。'`signal 0`' は、今まで送られてきたシグナルを無効にするのである。

11.4 Returning from a function

`return`

`return` *expression*

関数呼び出しから任意の位置で復帰させることができる。*expression* を指定した場合は、その値を復帰値とする。

`return` コマンドを用いることで、関数中の任意の位置から呼びだし元へ復帰させることができます。また、引数をあたえることで、その与えられた値を復帰値として呼びだし元へ復帰させることもできます。

`return` コマンドは `finish` コマンドとは違い自然なかたちで処理を継続しません。`return` コマンドは、復帰値用のレジスタに値をいれて、呼び出し元に処理を変更するような動きをするのです。

`finish` は、そのフレームの処理を終了させてから復帰します。

11.5 Calling program functions

`call` *expr*

expr に指定した関数を実行する。実行された関数は復帰値を表示しない。

もし、実行した関数の復帰値を見たかったら、`print` コマンドで、ヒストリ変数に格納された値を見て下さい。実行した関数の復帰値は、関数が値を返すのであればヒストリ変数に格納されています。実行された関数の復帰値を `call` コマンドは表示しません。

11.6 Patching programs

GDBのデフォルト状態では、実行形式ファイル（もしくは、コアファイル）は読み込み専用になっています。これは、GDB実行中にマシンコードが誤って書き換えられないための設定です。しかし、この防止策は、GDBの中からプログラムにバイナリパッチをあてることをできなくしてしまっています。

あなたが、もしパッチを当てることを好むのならば、`set write` コマンドを用いて、その操作を可能にすることができます。

`set write on`

`set write off`

`set write on` コマンドを実行すると、実行形式ファイルやコアファイルへの書き込みができるようになる。また、`set write off`（これがデフォルト）コマンドは、実行形式ファイルやコアファイルへの書き込みを禁止する。
もし、実行形式ファイルやコアファイルを変更した場合は、再度、そのファイルを読み込ませなくてはならない。

`show write`

`set write` の on/off を表示する。

Chapter 12

GDB Files

GDBは、シンボルテーブルを読み込んでプログラムの実行を開始するために、デバッグ対象のプログラム名を知る必要があります。以前のプログラム実行で発生したコアダンプをデバッグするためには、更にGDBにコアダンプしたファイル名を教えてあげなければなりません。

12.1 Commands to specify files

通常、あなたは、GDBが解析すべきファイルを、GDB起動時の引数として与えることでしょう。しかし、時々、GDBセッション中に異なったファイルへと変更する必要が発生することがあります。また、利用したいファイル名の記述を忘れたまま、GDBを起動してしまうこともあるでしょう。これらの状況に陥った時は、新しいファイルを記述するためのGDBのコマンドが便利です。

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

*filename*には、デバッグするプログラムを指定する。また、指定されたプログラムから、シンボル情報を読み込む。このプログラムをデバッグする時は、`run`コマンドで起動する。

作業用ディレクトリに、このプログラムのソースファイルが存在しない時は、`shell`と同じように`PATH`環境変数に登録されているディレクトリを検索する。この環境変数は、`path`コマンドを用いて変更することができる。

`-mapped`オプションを指定した場合、一部のシステムでは、GDBはメモリマップ機構を用いて動作する。メモリマップ機構を使用できるシステムでは、*filename*に対するGDBのシ

ンボルテーブルは、'*filename*`.sym`' というファイル名で存在する。'*filename*`.sym`' ファイルを用いた場合、GDB はシンボルの検索に時間がかからないため、素早く処理できるようになる。このファイルは、GDB のメモリ中のシンボルテーブルと同じ構造を持つ。

`-readnow` オプションは、シンボル情報を直ちに全て読み込むことを指定する。GDB は、すぐに全てのシンボルを読み込んでしまうわけではない。その代わり、GDB はどのソースファイルとどのシンボルが与えられたのかを調べるために、シンボルテーブルを高速にスキャンする。詳細は、必要が生じた時に、一つのソースファイルごとに後から読み込む。この 2 ステージ読み込み方式の目的は、GDB の起動を早くすることにある。よって、シンボル情報を参照した時に情報が足りない場合は、このオプションを用いて、シンボルテーブルを全て読み込ませるとよい。

`exec-file` *filename*

filename に指定された実行形式ファイルをデバッグ対象とする。もし、あなたがディレクトリを記述せず、GDB がファイルをカレントディレクトリで見つけることができなかった場合、GDB はサーチパスから、指定された名前の実行形式ファイルを探す。

`symbol-file it filename [-readnow] [-mapped]`

指定された *filename* からシンボル ·テーブル情報を読み取る。必要なら、環境変数 PATH に書かれているディレクトリを検索する。`-readnow -mapped` オプションは、`file` コマンドの時と同じ動きをする。

`exec-file` と `symbol-file` コマンドを同一のファイルに対して用いるのなら、`file` コマンドを使いなさい。

引数なしの `symbol-file` は、GDB のシンボルテーブルを初期化する。

`symbol-file` コマンドは、すぐに全てのシンボルを読み込んでしまうわけではない。その代わり、このコマンドはどのソースファイルとどのシンボルが与えられたのかを調べるために、シンボルテーブルを高速にスキャンする。詳細は、必要が生じた時に、一つのソースファイルごとに後から読み込む。こ

の2ステージ読み込み方式の目的は、GDBの起動を早くすることにある。あなたに対して、シンボルテーブルの詳細のための個々のソースファイルが読み込まれていることを報告する臨時メッセージを除いては、たいていの場合、これらのことは目につかない。

まだソースファイル内のデータが読み込まれていない関数に関するバックトレース結果を表示する場合は、シンボルテーブルの詳細がないと表示できないような引数の値といった情報ような情報は省略される。

‘`set verbose`’コマンドを使えば、`symbol-file`コマンドなどで表示されるメッセージをコントロールできます。

シンボルテーブルがCOFFフォーマットで格納されていた場合、`symbol-file`は全てのシンボルテーブルデータを即座に読み込む。残念ながら、私たち（GDBの作者）は、COFFに対して2ステージ読み込み方式をわざわざインプリメントするつもりはない。

`core-file` [*filename*]

メモリの内容として使われるコアダンプファイルがどこにあるのかを指定する。コアダンプには書き込み可能なメモリパートのみ含まれており; 読み出し専用のパートは実行可能ファイルにあることに注意すること。

引数なしの`core-file`は、コアファイルを使用しないよう指定する。

プログラムがGDBの下で実際に動作している時、コアファイルは無視される。プログラム実行中にコアファイルをデバッグしたくなった場合、動作中のプログラムのプロセスを殺す必要がある。プロセスを殺すには、`kill`コマンドを用いる。

`add-symbol-file` *filename address* `[-readnow] [-mapped]`

`add-symbol-file`コマンドは、*filename*というファイルから追加シンボルテーブル情報を読み込む。ファイルを実行中のプログラムに対して動的にロードする時に、このコマンドを用いる。*address*は、そのファイルが読み込まれる先のメモリアドレスである; GDBは、自動的にこれを検出することができない。

`-readnow -mapped`オプションは、`file`コマンドの時と同じ動きをする。

filename のシンボルテーブルは、`symbol-file` コマンドによって、読み込み済みのシンボルテーブルに追加される。

`add-symbol-file` コマンドは、コマンドラインからならいつでも利用できる。つまり、新しいシンボル情報は、古いものに追加して保持されるのである。`symbol-file` コマンドは GDB が既に読み込んだ全てのシンボル情報を初期化する。これは、GDB においてシンボル情報を初期化する唯一の方法である。

`info files`

GDB が、現在利用している、シンボルを読み込むための実行可能ファイル、コアファイルの名称を表示する。

これら 3 つのファイル記述用コマンドは、引数として絶対・相対パスのファイル名を引数として許していますが、GDB はファイル名を絶対パスに変換し、それを記憶しています。

`symbol-file` コマンドは、便宜上の変数、変数履歴、そして全てのブレークポイントと自動表示する式を忘れさせます。これは、これらの情報が、GDB の中から捨てられる古いシンボル情報の一部として、内部的に記憶されているシンボルとデータの型へのポインタを持っているからです。

12.2 Errors reading symbol files

シンボルファイルを読み込ませる時に、認められないシンボルが存在するとか、コンパイラの出力に障害があるとかという問題が起こる時があります。デフォルトでは、GDB は、そのよな問題を見つけてもあなたに通知しません。もし、あなたがシンボルテーブルを読み込んだ時に、どんな問題があったか知りたかったら、'`set verbose on`' コマンドを実行して下さい。このコマンドを実行した後でも、GDB は、1 つの問題に対してのメッセージしか出力しません。より多くのメッセージを出力させたかったら、'`set complaints`' コマンドを用いて出力させるメッセージの行数を設定して下さい。これらのコマンドについての詳細は、「Optional warnings and messages」(95ページ) を参照して下さい。この機能は、コンパイラのテストの時に役に立つでしょう。

Chapter 13

Specifying a Debugging Target

target とは、あなたのプログラムが使っている環境資源のことです。GDB は、多くの場合、プログラムの走っているマシンと同じマシンで使います。しかし、状況によっては、TCP で接続された先のマシンで走っているプログラムをデバッグしたい時もあるでしょう。GDB が走っているマシンと違ったアーキテクチャを持つプログラムをデバッグしたい時もあるでしょう。そんなときは `target` コマンドを用いてデバッグする *target* のタイプを指定します。`target` コマンドについては、「Commands for managing targets」（87ページ）を参照して下さい。

13.1 Active target

target は、プロセス、コアファイル、実行形式ファイル 3 つのクラスに分けることができます。GDB では、この 3 種類の *target* に対してデバッグすることができます。コアファイルが存在するとしたら、プログラムのクラッシュに関するより多くの情報を GDB は参照することができますコアファイルは、実行形式ファイルと合わせて使用されます。

13.2 Commands for managing targets

`target` *type parameters*

type には、プロトコルもしくはマシンタイプを指定する。*pa-*

ボーレートであるとかデバイス名であるとかを指定する。

rameters には、プロトコルに対するオプションを指定する。

`help target`

指定できる *target* の一覧と簡単な説明を表示する。

`help target` *name*

name で指定した *target* に関する情報を出力する。

`target exec` *program*

'`exec-file` *program*' と同じ。

`target core` *program*

'`core-file` *program*' と同じ。

`target remote` *dev*

シリアル回線上の *dev* を指定する。詳しくは、「Remote debugging」（88ページ）を参照のこと。

`target sim`

CPU シミュレータ。

13.3 Remote debugging

リモートデバッギングを開始するには、GDB を起動し、リモートマシンで動作している実行可能ファイルを指定します。これは、GDB に、プログラムのシンボルとピュアテキストを見つける方法を教えます。次に、プロセス ID の代わりにデバイス名を引数として持つ '`target remote`' コマンドを使って、コミュニケーションを確立します。例えば、

```
target remote /dev/ttyd
```

は、シリアル回線が '`/dev/ttyd`' という名称のデバイスに接続されている場合の指定です。これは、もしリモートマシンがまだ停止していない場合、停止させます。

TCP で接続されている場合は、

```
target remote manyfarms:2828
```

と指定します。これは、2828番ポートに接続されている`manyfarms`という名前のデバイスを指定したことになります。このとき、指定されたデバイスでは、`gdbserver`が動作していなくてはいけません。`gdbserver`については「Using the gdbserver program」(89ページ)を参照して下さい。

そこで、あなたは全ての通常コマンドを、リモートプログラムの調査やデータの変更、そしてステップ実行や継続のために使うことができます。

リモートプログラムから回復してそのデバッグをやめる場合は、'`detach`'コマンドを実行するか、'`C-c`'を入力します。

13.4 Using the gdbserver program

`gdbserver`は、Unix-likeなシステムを制御するプログラムです。このプログラムには、他のマシンで実行されているGDBから'`target remote`'コマンドを用いて接続することができます。

GDBと`gdbserver`の接続はTCPコネクションを用いて確立されます。

target 側の操作

`gdbserver`を用いたデバッグを行なう時は、実行形式ファイルを、ホスト側からコピーしなくてはいけません。また、`gdbserver`は、シンボル情報を必要としません。もし必要なら`strip`をかけて、プログラムからシンボル情報を削除してしまって構いません。全てのシンボル情報は、GDBがハンドリングします。

`strip`は、実行形式ファイルから、シンボル情報を削除するUNIXコマンドです。

`gdbserver`は、次のように入力して起動します。

```
gdbserver comm program [args... ]
```

*comm*には、シリアル回線のデバイス名、もしくは、TCPを利用する時は、ホスト名、ポート番号を指定します。

例として、シリアルポートが'`/dev/com1`'の時に、'`foo.txt`'というテキストを編集している`emacs`をデバッグするには以下のように`gdbserver`を起動します。

```
gdbserver /dev/com1 emacs foo.txt
```

また、TCP を利用する場合で、`silvia` というホストで 5963 というポート番号を持つマシンの場合は、

```
gdbserver silvia:5963 emacs foo.txt
```

という具合に起動します。

ホスト側の操作

いつものように GDB を起動します。では、*target* マシンに接続して見ましょう。
シリアル回線の場合は、

```
(gdb) exec-file emacs
(gdb) target remote /dev/com1
```

TCP を利用する場合は、

```
(gdb) exec-file emacs
(gdb) target remote silvia:5963
```

これで、*target* に接続できるはずです。もし、この時に '`Connection refused`' というメッセージが出力されたとしたら、ネットワークの設定に何か問題があるのでしょう。ネットワークの設定を確認してみて下さい。

Chapter 14

Contorolling GDB

GDB のユーザインターフェイスは、`set` コマンドを用いることによって、あなたの好みに近付くようカスタマイズできます。GDB コマンドが出力するデータをコントロールする方法は 「Print setting」(66ページ)を参照して下さい。

14.1 Prompt

GDB は、プロンプトという文字列を表示することによって、コマンドが入力可能であることを示します。この文字列は、通常 ‘`(gdb)`’ です。あなたは、この文字列を ‘`set prompt`’ コマンドによって変更することができます。例えば、GDB を使って GDB をデバッグするような場合、片方の GDB のプロンプトを変更することで、どちらの GDB があなたに話しかけているのかを識別することができ、とても便利です。

`set prompt` *newprompt*

GDB に、今後、プロンプト文字列を “*newprompt*” にすることを命じます。

`show prompt`

`prompt` を確認する。

14.2 Command editing

GDB は、コマンドラインからの入力を読みとります。この入力行は、`emacs`-like の編集方法も取れますし `csh`-like の編集方法も取れますし `vi`-

like[1]の編集方法も取れます。この機能は、無効にすることもできます。また、入力されたものは、GDB のヒストリ機能により保存されています。

コマンドラインでの編集方法は `set` コマンドで設定します：

`set editing`

`set editing on`
: コマンドラインでの編集を可能にする。(デフォルトは `on`)。

`set editing off`
: コマンドラインでの編集を不可能にする。

`show editing`
: 編集の `on/off` を表示する。

14.3 Command history

GDB はコマンドラインから入力されたものを、保存しています。これは、`csh` などのヒストリ機能と同じです。

`set history file` *filename*
: ヒストリを指定されたファイルに保存する。保存するファイル名は、環境変数 `GDBHISTFILE` で設定しておくことができる。また、デフォルトのファイル名は'`./.gdb_history`' である。このファイルは GDB 起動時に読み込まれる。

`set history save`

`set history save on`
: ヒストリをファイルに保存する。デフォルトはこちら。

`set history save off`
: ヒストリをファイルに保存しない。

---

[1] `vi-like` な編集を行ないたい時は付録「Command Line Editing」を参照して下さい。

`set history save` *size*

幾つのコマンドをヒストリファイルに残すか指定する。また、これは環境変数 `HISTSIZE` に設定しておくこともできる。デフォルト値は 256。

‘`history expansion`’ が有効なときには、‘`!`’ は特別な意味をもつキャラクタになります。

このキャラクタに意味を持たせるかどうかは、‘`set history expansion`’ コマンドで設定することができます。通常は、特別な意味を持っていません。‘`history expansion`’ が有効な時は、`csh` のヒストリ機能と似たような操作を、この文字を使うことによって行なえます。

‘`history expansion`’ は以下のコマンドで操作します ：

`set history expansion`

`set history expansion on`

‘`history expansion`’ を有効にする。

`set history save off`

‘`history expansion`’ を無効にする。

`show history`

`show history filename`

`show history save`

`show history size`

`show history expansion`

それぞれの機能の状態を表示する。

`show command`

ヒストリに登録されているコマンドを最後から 10 個表示する。

`show command` *n*

指定された番号を中心として、ヒストリに登録されているコマンドを１０個表示する。

`show command +`

最後に表示したヒストリの次の 10 個のコマンドを表示する。

14.4 Screen size

あるコマンドは、スクリーン上に膨大な量の情報を表示してしまいます。あなたがこれらの情報の全てを読むための手助けとして、出力した情報がページ単位（スクリーン一杯）になった時、GDB は出力をポーズしてあなたに問い合わせます。出力を続けたい場合は、改行を入力するだけでよいです。

この機能のことをページャ機能と呼びます。

通常、GDB は、環境変数 `TERM` とともに、`termcap` データベースを用いてスクリーンの形態を調べます。もし、これが正しくない場合は、コマンドによって値を変更することができます。

`set hight` *lpp*

`show hight`

`set width` *cpl*

`show width`

‘`set hight`’ を用いて、スクリーンの行数が *lpp* であることが設定できる。また、‘`set width`’ を用いて、カラム数が *cpl* であることが設定できる。

もし、行数に 0 を設定した場合、GDB は出力する情報がどんなに長くなろうとも、出力をページ分割しない。これは、GDB の出力をファイルにとったり、エディタのバッファに入力したりするときに便利である。

`show` は、現在の行数、カラム数を表示する。

14.5 Numbers

普段入力する数値が10進数であるとか、16進数であるとか、8進数であるとかを設定しておくことができます。こうしておくことで、16進数を入力するときに`0x`を付加したり、8進数を入力する時に0を付加したりしなくてもすみます。デフォルトは10進数になっています。

数学用語で、このように進数の基となる値のことを基数と呼びます。

`set radix` *base*

普段、入力する数値の基数を設定する。これには、8進数、10進数、16進数が選択できる。設定方法は以下の通り：

`set radix 012` (8進数の時)
`set radix 10` (10進数の時)
`set radix 0xa` (16進数の時)

`show radix`

基数を表示する。

14.6 Optional warnings and messages

GDBは、動作中に生じたある種の問題についてのメッセージを、デフォルト状態では出力しません。これらのうちのいくつかは、'`set verbose`'コマンドによって出力されるように切り替わります。

`set verbose off`

GDBのいくつかの情報メッセージ出力を抑止する。デフォルトはこちら。

`set verbose on`

GDBのいくつかの情報メッセージ出力を許可する。

普通、'`set verbose`'コマンドによってコントロールされるメッセージは、ソースファイル上のシンボル情報を読み込む時にアナウンスされる類のものです。詳しくは「GDB Files」(83ページ)を参照して下さい。

`set complaints` *limit*

GDBの表示する型やシンボルの出力行数の最大行数を設定する。
0を設定した場合は、制限なしの指定となる。

`show complaints`

現在、設定されている最大行数を表示する。

GDB は、本当にその操作をしても良いか判断できない時は、あなたに問い合わせます。例えば、あるプログラムを実行中に、再度そのプログラムを実行しようとした場合、次のように表示されます ：

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

こ問い合わせが煩わしいと感じるなら、以下のコマンドで抑止できます ：

`set confirm on`

処理の実行を問い合わせる。

`set confirm off`

処理の実行を問い合わせない。

`show confirm`

`confirm` の `on/off` を表示する。

あるシステムでは、デバッグプログラムの実行を止めないで、新しい実行形式ファイルを作成した場合、次の実行時に GDB に自動的に新しいシンボル情報を読み込ませることができます ：

VxWorks システムはこの機能をサポートしています。

`set symbol-reloading on`

実行形式ファイルが更新された場合、自動的にシンボルを再読み込みする。

`set symbol-reloading off`

実行形式ファイルが更新された場合、自動的にシンボルを再読み込みさせない。

`show symbol-reloading`

`symbol-reloading` の `on/off` を表示する。

Chapter 15

Canned Sequences of Commands

GDB は、実行するコマンドのシーケンスをユニットとして記憶するための 2 つの方法を提供しています。　その 2 つとは、ユーザ定義コマンドとコマンドファイルです。

15.1 User-defined commands

ユーザ定義コマンドとは、GDB のコマンドのシーケンスを、あなたが新しいコマンドとして定義したものです。これは、'`define`' コマンドで定義できます。

`define` *commandname*

commandname という名称のコマンドを定義する。もし、既に定義済のコマンドが存在する場合は、再定義して良いかとうか確認を取る。

`define` コマンドを入力すると編集モードに入りコマンドの定義を受け付ける。編集モードから抜けるには '`end`' を入力する。

`document` *commandname*

ユーザ定義した *commandname* の説明を作成する。*commandname* で指定されるコマンドは、このコマンドを使う前に定義

されている必要がある。このコマンドは、`define` コマンドと同様に、編集モードで説明を入力する。

`document` コマンドの定義終了後は、`help` コマンドで *commandname* を指定すれば、作成した説明文書が表示される。コマンドの説明を修正するためにも `document` コマンドを使用する。‘`define`’ コマンドを用いて、コマンドを再定義したとしても、説明を修正しない限り同じ説明が使用される。

ユーザ定義のコマンドは引数を受け付けません。また、これらが実行された時、コマンドの定義内容は表示されません。任意のエラーはユーザ定義コマンドの実行を停止させます。

対話的に処理されている時に確認を求めてくるコマンドは、ユーザ定義コマンドの中で使われた場合、確認を求めてきません。多くの GDB のコマンドは、通常、今何をやっているのかというメッセージを表示しますが、ユーザ定義コマンドの中で使用された場合、メッセージを省略します。

15.2 User-defined command hooks

GDB では、*hook* という特別なユーザ定義コマンドがあります。これは、‘`foo`’ というプログラムを動作させる前に、‘`hook-foo`’ というユーザ定義コマンドが存在しているとしたら、そのプログラムを常に引数なしで実行させます。

また、`stop` というコマンドが存在するなら、`hook-stop` というコマンドを定義すれば、`stop` というコマンドが実行される時に、それより前に `hook-stop` コマンドが実行されます。

それでは、`SIGALARM` に対して、この機構を用いた例を見て下さい：

```
define hook-stop
handle SIGALARM nopass
end

define hook-continue
handle SIGALARM pass
end
define hook-continue
```

```
handle SIGALARM pass
end
```

このように、*hook* 機能では、１行の処理しか指定できません。しかし、これはコマンドの別名定義のためのものではありません。この機能は、GDB コマンドのために使用するべきでしょう。もし、GDB コマンドに対して *hook* を用いた時に、その記述に誤りがあったとしたら、GDB コマンドは実行されずプロンプトしか表示されなくなります。

もし、あなたが *hook* を存在しないコマンドに定義しようとしたら、`define` コマンドから警告メッセージを受けとることになります。

15.3 Command files

GDB のコマンドファイルとは、GDB のコマンドを行として持つファイルのことです。コメント (‘#’ ではじまる行) もまた、含まれます。コマンドファイル内の空行は何も行いません; これは、端末で行った場合のように、最後のコマンドを繰り返したりはしません。

GDB を起動した時、GDB は、その初期化ファイルである ‘`.gdbinit`’ という名称のコマンドファイルを自動的に実行します。GDB は初期化ファイルを (もしあれば) あなたのホームディレクトリから読み込み、(もしあれば) カレントディレクトリからも読み込みます。 あなたは、また、`source` コマンドによって、コマンドファイルの実行をリクエストすることができます:

もし、‘`-nx`’ オプションが指定されていたならば、初期化ファイルは実行されません。

`source` *filename*

filename というコマンドファイルを実行する。

コマンドファイル上の行はシーケンシャルに実行されます。実行されるコマンドの名称は、実行中に表示されません。どこかのコマンドでエラーが発生した場合、コマンドファイルの実行は中止されます。

対話的に処理されている時に確認を求めてくるコマンドは、コマンドファイルの中で使われた場合、確認を求めてきません。多くの GDB のコマンドは、通常、今何をやっているのかというメッセージを表示しますが、コマンドファイルの中で使用された場合、メッセージを省略します。

15.4 Commands for controlled output

コマンドファイルやユーザ定義コマンドの実行中は、その中で出力コマンドを用いて表示しているもの以外は出力されません。ここでは、定義したコマンドのメッセージ出力に便利な 3 つのコマンドを説明します。

`echo` *text*

text を表示する。表示できない文字は、C 言語で`\n` と書くと改行するように、`echo` コマンドでも C 言語と同じエスケープシーケンスを用いて記述する。改行マークは表示しようとしない限り絶対に表示されることはない。また、C 言語の標準エスケープシーケンスに加えてバックスラッシュの後に続くスペースはスペースを意味する。このスペースの表現は、文字列の最初や最後にスペースを出力する時に便利である。従って、‘`and foo = `’ と表示するには、‘`echo \and foo =\`’ というコマンドを使う。

引数の文字列は通常いらない空白を削除されてしまうため空白が入らない。

text の最後にあるバックスラッシュは、C 言語と同様、その後に続く行の継続を行いう。例えば、

```
echo This is some text\n\
which is continued\n\
onto several lines.\n
```

は、次の場合と同様の出力を行う。

```
echo This is some text\n
echo which is continued\n
echo onto several lines.\n
```

`output` *expression*

expression の値だけを表示する: 改行も ‘`$nn = `’ も表示しない。また、値はヒストリ変数に記録されない。*expression* についての詳しい説明は、「Expressions」（62ページ）を参照のこと。

`output /`*fmt expression*

fmt に従って、*expression* の値を表示する。詳しくは、「Output formats」（63ページ）を参照のこと。

`printf` *string, expressions...*

string の制御に従って、*string* の値を表示する。*expression* はカンマで区切られた、数字かポインタでなければならない。これらの値は、プログラムが

`printf(`*expression, ...*`);`

を実行する場合と同様に、指定された *string* に従って値を表示する。例えば、2 つの値を次のようにして 16 進数で表示することができる:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

この文字列の中で使うことができるバックスラッシュによるエスケープシーケンスは、バックスラッシュの後に 1 文字続くという単純な形式のものだけである。

Chapter 16

Using GDB under GNU Emacs

GDBはGNU Emacsの配下で動作させせることによって、ソースファイルを参照したり、編集したりというような使い方に対して特別なインターフェイスを提供します。

このインターフェイスを用いるためには、Emacsの中で‘`M-x gdb`’と打ちます。このコマンドを実行すると、GDBはEmacsの子プロセスとして入力や出力を新しくEmacs中に作成されたバッファで行なうようになります。

GDB単独で動作させるよりEmacs配下で動作させる方がよいことが２点あります。

- 全ての入出力がEmacsバッファで行なえます。

これは、GDBコマンドの入出力だけでなく、デバッグプログラムの入出力もバッファ内で可能になります。

このことが役に立つという理由は、同じ入力が繰り返しあったり、出力の一部を入力に使いたいという局面が、Emacsのカット＆ペーストを用いることで繰り返し行なえるようになるからです。

Emacsのシェルの機構は、デバッグプログラムに対しても働きます。特別なのは、シグナルの送り方が少し変わることですね — たとえば、`C-c`

`C-c`がインターラプトシグナルを生成します。また、`C-c C-z`がストップシグナルを生成します。

- ソースコードがEmacsを通して表示されるようになります。

Emacsは自動的にカレント行を検出して矢印（‘=>’）をソースの左はしに位置付けます。また、Emacsはウインドウを2分割して、ソースとGDBセッションをいつも表示してくれます。

これで、GDBの`list`コマンドでソースリストを出力したりとか、`search`コマンドでソース中のあるパターンを検索したりとかいう、煩わしい労働からあなたは解放されます。

注意 ただし、ソースが色々な場所に散在するという状況下では、Emacsはソースを簡単に見つけることができません。Emacsが自動的にソースを表示できるのは、ソースファイルが作業用ディレクトリにあるときだけです（おい、おい`(^^;)`なんとかならんもんかいな）。

`M-x gdb`で起動されるGDBプログラムは、デフォルト名が`gdb`になっています。もし、あなたの使っているGDBプログラム名が、これと違う場合は、Emacsの変数の`gdb-command-name`に、その名前を設定してやる必要があります。設定方法は以下の通りです：

```
(setq gdb-command-name “mygdb”)
```

この1行を、あなたの‘`.emacs`’ファイルに追加するか、`*scratch*`バッファで評価させるか、この式を入力してから`ESC ESC`を打って評価させて下さい。こうすることで、起動するGDBプログラムの名前が“*mygdb*”に変更されます。

GDB入出力バッファで使える、標準のShellモードに追加された特別なEmacsコマンドのリストを以下に示します：

`C-h m`	GDBモードの特徴を表示する。
`M-s`	`step`コマンドを実行する。
`M-n`	`next`コマンドを実行する。

`M-i`	`stepi` コマンドを実行する。
`M-x gdb-nexti`	`nexti` コマンドを実行する。
`C-c C-f`	`finish` コマンドを実行する。
`M-c`	`continue` コマンドを実行する。 注意 :Emacs v19 では `C-c C-p`。
`M-u`	`up` コマンドを実行する。 注意 :Emacs v19 では `C-c C-u`。
`M-d`	`down` コマンドを実行する。 注意 :Emacs v19 では `C-c C-d`。
`C-x &`	`disassemble` コマンドを実行する。

`M-u` と `M-d` は引数を与えることができます。詳しくは「The GNU Emacs Manual」のセクション「Numeric Arguments」を参照して下さい。

ちなみに、`C-u` 引数で数字の引数を与えることができます。

ソースバッファに対して使える Emacs コマンドに `C-x SPC` があります。このコマンドを使うことによって、カーソルのある行にブレークポイントを設定することができます。

何かのアクシデントが起きて、ソースバッファが消えてしまった時は、GDB のコマンドラインで `f` コマンドを入力して下さい。これは、フレームの表示を要求するコマンドですが、あなたが Emacs 配下で作業しているならば、直ちにソースバッファを作成してソースを表示します。

もし、ソースバッファを見ていて、編集したいという欲求にかられても、GDB と Emacs が通信しているんだということを思いだして、思い留まってくださいね。もし、ソースコードを編集したり、行番号を削除したりしてしまったら、GDB は、そのバッファとソースコードが正確に一致しているということを確認する術をを失ってしまいます (つまり、動作は保証せんてっか(^^;))。

Part III

付録

Appendix A

Command Line Editing

GNU で規定されている、コマンドラインのインターフェイスについて記述します。GDB では、ここに示す通りにインプリメントされていないかも知れませんが、概ね同じように動作するはずです。

A.1 Introduction to Line Editing

以下の段落にキーストロークの表現について記述します。

'`C-K`' という表記は、'Control-K' と読みます。これは、コントロールキーと '`K`' キーを同時に押すという意味です。

'`M-K`' という表記は、'Meta-K' と読みます。これは、メタキーと '`K`' キーを同時に押すという意味です。メタキーを持たないキーボードの場合は、'`ESC`' キーを押した後に '`K`' キーを押して下さい。

'`M-C-K`' という表記は、'Meta-Control-K' と読みます。これは、メタキーとコントロールキーと '`K`' キーを同時に押すという意味です。メタキーを持たないキーボードの場合は、'`[ESC]`' キーを押した後に '`C-K`' キーを押して下さい。

この他に、いくつかのキーは、独特の名前で呼んでいます（例えば、`DEL`，`ESC`，`LFD`，`SPC`，`RET`，`TAB`）。この表記は、あるファイルでキーを定義する時にも用いることができます。

A.2 Readline Interaction

よく、コマンドラインから、長いテキストを入力する時があります。こんな時、最初の単語に誤りがあることに気づいたとしたら、`Readline` ライブラリが、この単語の修正のための機能を提供してくれます。カーソルを修正したいところに移動して、修正して、テキストが確定したら `RET` キーを入力して下さい。わざわざ、行末までカーソルを移動させて `RET` キーを入力する必要はありません。

A.3 Readline Bare Essentials

文字の入力は、ただ単にキーボード上の入力したいキーを押せばよいです。そうすれば、画面にその文字が表示されます。入力を間違えた場合は、`DEL` キーを押して間違えた文字を削除して下さい。

時々、ある程度文字を入力した後に、ミスタイプしたことに気づく場合があります。そんな時は、’`C-F`’ でカーソルを行頭の方へ移動させて修正したい文字に位置付けて下さい。‘`C-B`’ キーで行末の方へカーソルを移動させることもできます。

また、コマンドラインで行なった修正を取り消したい時は、‘`C-_`’ を入力すれば、一つ前の修正が取り消されます。このキーを繰り返し押せば、どんどん前に行なった修正を取り消していくことができます。

`C-B`	カーソルを行末方向へ動かす。
`C-F`	カーソルを行頭方向へ動かす。
`DEL`	カーソルの左にある文字を削除する。
`C-D`	カーソルの位置にある文字を削除する。
`C-_`	一つ前の修正を取り消す（`Undo`）。

A.4 Readline Movement Commands

コマンドラインでの入力を編集するために用意されたカーソル移動キーの一般的なものを以下に示します：

`C-A`	行頭にカーソルを移動する。
`C-E`	行末にカーソルを移動する。

`C-F`	行頭方向に一文字分カーソルを移動する。
`C-B`	行末方向に一文字分カーソルを移動する。
`C-L`	画面をクリアして、現在のコマンドラインを再描画する。

また、‘`C-F`’ は、文字単位にカーソルを移動させますが、`M-F` を使うことによって、単語単位にカーソルを移動させることができます。

A.5 Readline Killing Commands

現在、コマンドラインに表示されているテキストを消してしまいたい時や、ある単語を別の位置に移動させたい時は、*Killing* `text` コマンドを用います。以下に、そのコマンドを示します ：

`C-K`	カーソルのある位置から後ろの部分を削除する。
`M-D`	カーソルのある位置から単語ひとつを削除する。
`M-DEL`	カーソルのある位置の前の単語をひとつ削除する。
`C-W`	カーソルのある位置から直前のスペースまでの文字列を削除する。

では、次に、消した文字を *yank* するコマンドを示します ：

`C-Y`	消したテキストを戻す。
`M-Y`	消したテキストを戻す。繰り返しこのキーを入力すれば、どんどん前のテキストに戻っていく。

テキストを削除した場合、そのテキストは *killing-ring* に保存されます。また、保存されたテキストは、そのコマンドライン中だけでなく別の行でも利用することができます。

A.6 Readline Arguments

コマンドラインで使える編集キーには、数字の引数を与えることができます。例えば、`M-5 DEL` と入力すれば、カーソル位置の左側の 5 文字が削除されます。

```
(gdb) This is a japan         (ここで M-5 と入力すると)
(arg: 5) This is a japan      (そこで DEL を入力すると)
(gdb) This is a               (こうなります。)
```

A.7 Readline Init File

Readline ライブラリは、Emacs-like に設定されていますが、あなたの好みに合わせてキーバインドを変更することができます。このカスタマイズを行なう時は、‘ ~/.inputrc’ に設定を記述します。‘.inputrc’ は起動時に読み込まれます。

また、`C-X C-R` を入力すれば、‘.inputrc’ を再読み込みします。

A.7.1 Readline Init Syntax

ここでの記述は‘~/.inputrc’ ファイルにだけ適用されます ：

Variable Settings

編集モードを Vi の用にしたい時は、次のように記述する。

```
set editing-mode vi
```

ここで、設定できるモードは以下の通りである ：

`editing-mode`

> `emacs` か `vi` を設定する。これは、どちらのキー操作に似せるかの設定である。デフォルトは `emacs`。

`horizontal-scroll-mode`

> `On` か `Off` を設定する。水平方向の入力最大幅に達した時、改行するかしないかの設定である。`On` の時は改行しない。デフォルトは `Off`。

`mark-modified-lines`

> `On` の時は、この行が編集されていたら*を行頭に表示する。デフォルトは `Off`。

`prefer-visible-bell`

> `On` の時は、ビジュアルベルを使用する。デフォルトは `Off`。

Key Bindings

キーバインドは、以下のように記述して変更する ：

keyname: function-name or macro

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: ">&output"
```

この設定は、‘`C-U`’ に `universal-argument`’ をバインドする。また、‘`C-o`’ を押すと “`>&output`” という文字列がカーソル位置に挿入される。

“keyseq”: function-name or macro

keyseq には、Emacs のような表記が使える。

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

キーマクロコマンド一覧

`beginning-of-line (C-A)`	行頭へカーソルを移動する。
`end-of-line (C-E)`	行末へカーソルを移動する。
`forward-char (C-F)`	カーソルを１文字前へ移動する。
`backward-char (C-B)`	カーソルを１文字後ろへ移動する。
`forward-word (M-F)`	カーソルを１単語前へ移動する。
`backward-word (M-B)`	カーソルを１単語後ろへ移動する。
`clear-screen (C-L)`	画面をクリアして、現在のコマンドラインを再描画する。
`accept-line (Newline, Return)`	改行する。行が空行でなければ、入力をヒストリに登録する。

previous-history（C-P）	一つ前のヒストリを表示する。
next-history（C-N）	一つ後のヒストリを表示する。
beginning-of-history（M-<）	先頭のヒストリを表示する。
end-of-history（M->）	最後のヒストリを表示する。
reverse-search-history（C-R）	ヒストリを先頭に向かって検索する。
forward-search-history（C-S）	ヒストリを最後に向かって検索する。
delete-char（C-D）	カーソルのある位置の１文字を削除する。行頭では EOF の入力になる。
backward-delete-char（Rubout）	カーソルの右にある１文字を削除する。
quoted-insert（C-Q, C-V）	制御文字をエスケープする。この文字の次に入力された文字を、そのまま特別な意味を持たせずに入力する。
tab-insert（M-TAB）	タブ文字を入力する。
self-insert（a, b, A, 1, !, ...）	任意の１文字を入力する。
transpose-char（C-T）	カーソル位置の文字と直前の１文字を入れ換える。
transpose-word（M-T）	カーソル位置の単語と直前の単語を入れ換える。
upcase-word（M-U）	カーソル位置から最後までの文字を大文字にする。

downcase-word（M-L）	カーソル位置から最後までの文字を小文字にする。
capitalize-word（M-C）	カーソル位置から最後までの単語の先頭の１文字を大文字にする。
kill-line（C-K）	カーソルから後ろの部分を削除する。
backword-kill-line ()	カーソル位置から前の部分を削除する（通常はバインドされていない）。
kill-word（M-D）	カーソル位置の後にある単語を削除する。
backword-kill-word（M-DEL）	カーソル位置の前にある単語を削除する。
yank（C-Y）	*kill-ring* から、最後の１つをヤンクする。
yank-pop（M-Y）	*kill-ring* を繰り返しヤンクする。
digit-argument（M-0, M-1...）	キーコマンドに数字の引数を与える。
universal-argument ()	任意の引数をキーコマンドに与える。通常はバインドされていない。
complete（TAB）	補完キー。bash の TAB と同じ働きをする。
possible-completions（M-?）	補完後のテキストを表示する。
re-read-init-file（C-X C-R）	初期化ファイルを再読み込みする。
abort（C-G）	編集コマンドの実行を途中で止める。
prefix-meta（ESC）	メタキャラクタを指定する。
undo（C-_）	Undo する。
revert-line（M-R）	現在行に対する全ての編集を取り消す。

Appendix B

Using History Interactively

GNU ヒストリ拡張ライブラリについて説明します。GDB では、ここに記述した全ての機能をサポートしているわけではありません。中には、使えない機能があるかも知れません。

B.1 History Interraction

ヒストリ拡張ライブラリのヒストリ機能は、Csh に似せて作成しています。以下の章では、その使い方について説明します。

ヒストリ拡張は 2 種類の部分に分けることができます。2 種類の部分とは、コマンドラインに対するヒストリ拡張と単語に対するヒストリ拡張です。コマンドラインに対するヒストリ拡張のことをイベントと呼びます。また、単語に対するヒストリ拡張をワード (Word) と呼びます。これらの処理は、Bash と同じような動きをします。

B.2 Event Designators

コマンドラインに対するヒストリ拡張を以下に示します :

`!`	ヒストリ機能を示す文字です。コマンドラインの先頭に記述します。
`!!`	‘`!-1`’ と同じで、一つ前のコマンドを実行します。

`!n`	ヒストリ番号 n 番のコマンドを実行します。
`!-n`	現在のヒストリ番号から n を減算した番号のコマンドを実行します。
`!string`	string という文字列で始まる直前に実行したコマンドを実行します。
`?string[?]`	string という文字列を含む直前に実行したコマンドを実行します。

B.3 Word Designators

単語に対するヒストリ拡張を以下に示します：

`0(zero)`	先頭の単語。多くの場合、コマンド名。
`n`	n 番目の単語。
`^`	単語の先頭の 1 文字。
`$`	最後の単語。
`%`	?string?に一致した単語。
`x-y`	文字の範囲を指定する。この場合 x から y になる。
`*`	0 個以上の文字と一致する。

B.4 Modifiers

単語に対するヒストリ拡張のオプション記述子を以下に示す：

`#`	入力された単語の数。
`h`	パス名の先頭部分。
`r`	パス名の基本部分（拡張部分を含まない）。
`e`	パス名の拡張部分。
`t`	パス名の末尾部分。
`p`	表示する（実行しない）。

索 引

版数	作成日付	著者	備考
第 1 版	1994/04/23	藤原　詩郎	新規作成
第 2 版	1994/04/30	藤原　詩郎	「Remote debugging」他の章追加
第 2.1 版	1994/05/05	藤原　詩郎	構成変更。一部加筆。
第 2.2 版	1994/05/08	藤原　詩郎	索引追加。
第 2.3 版	1994/05/12	藤原　詩郎	索引修正。脱字修正。
第 3.0 版	1994/05/18	藤原　詩郎	GDB Ver4.12 版対応。

おわりに

本書の査読に協力してくださった。石垣くん＆郷間くんに感謝します。また、私のプロジェクトでいつも、ばかなリーダを支えてくれている “ばくちゃん” & “とりちゃん” に感謝をこめてこのマニュアルを捧げます（おら、おら、働かんか（^^;））。

それでは、このマニュアルが少しでもみなさまの役に立つことを祈って、、、

平成 6 年 4 月末日
株式会社富士通ソーシアルサイエンスラボラトリ
藤原　詩郎