

大規模ソフトウェアを手探る

田浦

Contents

1	前口上	3
1.1	本課題の目標	3
1.2	この課題を作った理由	3
1.3	ロードマップ	3
2	アホとはさみ—configure, make, gdb—を学ぶ	5
2.1	ソースを入手する	5
2.2	通常のビルド手順	6
2.3	configure の最重要オプション: <code>--prefix</code>	6
2.4	デバッガで追跡しやすいコンパイル方法	8
2.5	デバッガで追跡する	9
2.5.1	Emacs で <code>gud-gdb</code> コマンド	10
2.5.2	<code>break</code> コマンドで停止位置を設定	11
2.5.3	<code>run</code> コマンドで実行開始	11
2.5.4	<code>next</code> コマンドで一行実行	11
3	実録! <code>gnuplot</code> を変更する	12
3.1	目標の再確認	12
3.2	基本作戦— <code>next</code> と <code>step</code>	12
3.3	実行開始	13
3.4	<code>next</code> か? <code>step</code> か?	13
3.5	行き過ぎてから、やり直す方法	15
3.6	<code>step</code> で奥へ進む	16
3.7	<code>finish</code> コマンドで関数から抜けるまで実行	16
3.8	再び <code>next</code> で関係ないところを通過〜 <code>do_line</code> , <code>command</code> までの旅	17
3.9	<code>until</code> コマンドでループの終わりまで通過	18
3.10	ついにポスカラ (<code>lookup_ftable</code>) に遭遇!?	18
3.11	配列の中身を一挙に表示する方法 (@)	19
3.12	ポスカラその 2: (<code>plot_command</code>)	20
3.13	変更の概要	21
3.14	変更 1: <code>plot_command</code> の変更	21
3.15	ソースコード変更時の基本—簡単に元に戻せるように	22
3.16	変更 2: <code>lookup_ftable</code> の変更	23
3.17	<code>up</code> コマンドで関数呼び出し元を表示する	23
3.18	<code>etags</code> コマンド + <code>find-tag</code> で定義へ飛ぶ	24
3.19	<code>command_ftbl</code> 表の変更	25
3.20	変更したコードをいざコンパイル	25
3.21	この続きは...	26
3.22	より模範的なソースコード変更の仕方	27
4	これからどこへ	27

A	ビルドに関連する基本知識およびコマンド	29
A.1	分割コンパイルの基本	29
A.1.1	ビルドの手順	29
A.1.2	変数, 配列	30
A.1.3	関数の呼び出し	31
A.1.4	ヘッダファイル	32
A.1.5	型	33
A.1.6	マクロ定義	34
A.2	よく使われるコンパイラのオプション	34
A.3	make	35
A.3.1	make の基本	35
A.3.2	make のカスタマイズ	36
A.3.3	自分で定義した変数の利用	36
A.3.4	最小限の再コンパイル	37
A.3.5	一般の Makefile	38
A.3.6	clean	38
A.3.7	コマンドラインを隠す人	38
A.3.8	make と configure	39
A.3.9	それでも Makefile に修正を加えなくてはならない場合	39
A.4	configure	40
A.4.1	configure とは	40
A.4.2	configure でエラーが出た時の対処	41
B	ソースツリーを探索するツールあれこれ	43
B.1	grep	44
B.2	GNU global (gtags, htags, global)	44
B.2.1	インストール	44
B.2.2	htags : ブラウザでソースを閲覧	45
B.2.3	gtags と global	45
B.2.4	Emacs と gtags	46

1 前口上

1.1 本課題の目標

この課題では、

- 世の中で実際に使われている、
- 大きな

ソフトウェアを改良，機能拡張できるようになることを目標にする．本課題のテーマは，実際のソフトウェア作りの現場—製品づくりでも研究でも—でほぼ必ず遭遇する場面：

- 自分で書いたわけではないので全容を把握していない，または、
- 全容を把握できない程度に大きい

ソフトウェアを改良したり，機能拡張する場面を経験して，大きなソフトウェアを作るのに必要なスキルや考え方を身につけることである．

1.2 この課題を作った理由

ほとんどの大学のプログラミング課題は，あるトピック—アルゴリズム，ネットワーク，音声，グラフィクス，...—に関連するプログラムを「一から」作るという課題である．限られた時間内に，一から作るという制約上，作れるソフトウェアの規模は自ずと小さくなる．そのような課題は，プログラミング言語の基本を身につけ，対象となるトピックの基本を理解するためには必須・有用なものであるが，それを繰り返していても本格的な大規模プログラムを構築するために当たりまえのように使われている道具，スキル，知識，概念，etc. は身につかない．それらは，体系的に整理することが難しい，ないし，整理するほどのことでもないと思われるということもあり，あまり明示的に教えられないこともない．

この課題では，それを学ぶ—教えるというよりも自ら学んでもらう—ことを目的とする．

1.3 ロードマップ

そのためにこの課題では、

既存のオープンソースのソフトウェアの改良・機能拡張

を目標とする．具体的にどのソフトウェアを対象とするか，それにどのような改良・拡張を施すかも自分達で決める．

課題期間中の活動内容は、

1. デバッガを使ってソフトウェアの動きを追跡する練習
2. チーム作り
3. チームで，どのようなオープンソースソフトウェアを対象に，どのような改良・拡張を行うかを議論する
4. その計画を早いうちに発表し，他のチームや教員・TAからのフィードバックを得る
5. 定期的に進捗を発表し，他のチーム，教員・TAからのフィードバックを得る
6. 活動内容，調査内容のドキュメントを残す(レポート)

最終的に達成した課題の有用さや難易度も大事だが，それをしっかりとドキュメント化(あとに引き続く人が有用だと感じるドキュメントを残す)することも重要視する．

この教科書の残りは主に上記の1.を補足するために書かれているが，それは課題の内容の，最も表面的な部分に過ぎない．実際に身につけて欲しいのは，以下のことである．それは研究や製品作りの場面できつと有用になることである．

1. よくわからないソフトを解析して，なんとかする根性
2. 背景理論もよく知らないソフトの背景理論を独習して，なんとかする根性
3. ツールの高度な使い方を，人から教わるのではなく，自分で一次ソース (と言ってもソフトの使い方の一次ソースはマニュアルであり，歴史的文書と異なり，簡単に手に入る) を調べて，自らスキルを高めていく根性

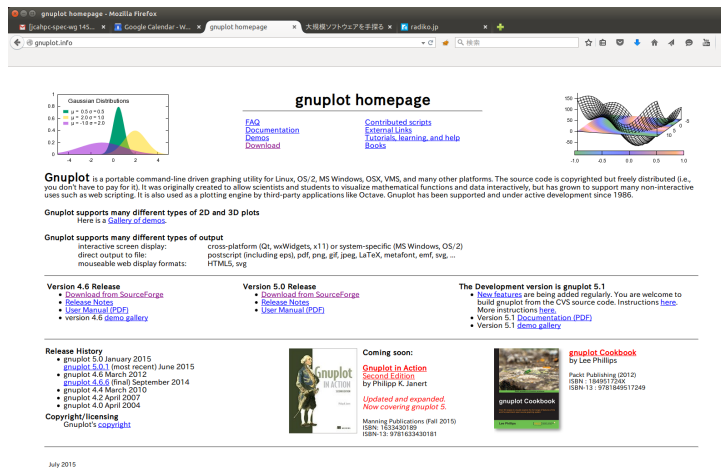


Figure 1: Gnuplot ホームページ

2 アホとはさみ—configure, make, gdb—を学ぶ

この節では例題として、ポピュラーで、さほど大きくない¹ オープンソース・ソフトウェアとして、gnuplot をとりあげ、

デバッガ (gdb) でその動作を追跡できるように なる

ために必要なノウハウを説明する。第3節で、gnuplot に簡単な変更を試みる。

デバッガでプログラムの動作を追うことは、およそどんなプログラムが対象であっても、追跡の基本手法である。デバッガ無しでソースコードを眺めても、詳細な動作の理解はおぼつかない。当てずっぽうでソースをいじることになるし、エラー (例えば segmentation fault) に遭遇しても、その原因を追うことはとてもできない。

デバッガでプログラムの動作を追ひ、変更するためには、

1. プログラムをソースからコンパイル (ビルド)、
2. デバッガ情報を付けてコンパイル、
3. デバッガで走らせる

というステップを踏む必要がある。本節の残りでは gnuplot を対象に上記のステップを説明する。

2.1 ソースを入手する

今日、Linux でも Mac でも、プログラムのインストールはパッケージ管理ソフトで簡単に行えるようになっている。だが、プログラムを変更しようと思ったら必要なのは、ソースからビルドすることである。

ほとんどの有名なオープンソースソフトウェアのソースコードは、プログラム名でそのホームページを検索すれば、入手することができる。Gnuplot も例外ではなく、そのホームページ (gnuplot.info; 図1) はググれば簡単に見つかる。ソースコードはこの中の、“Download from SourceForge” というリンクをたどれば入手できる。以降の説明ではここから、gnuplot-5.0.1.tar.gz を入手したことを前提としている。これはソースコード一式が収められたファイルを、一つのファイルに収めたもので、アーカイブとか、ターボール (tar ball) などと呼ばれる。

¹かつての学科のほとんどの人は既に利用経験済みであろう

準備課題 2.1 gnuplot 5.0.1 のソースコードを入手せよ。

2.2 通常のビルド手順

Linux, Mac, FreeBSD など, Unix 系の OS 用のオープンソース・ソフトウェアでは, ソースを入手したあとのビルド手順は, ほとんどのソフトウェアで共通化されており, 以下の手順で行える。

1. tar コマンドでソースアーカイブ (.tar.gz, .tar.bz2 など) を解凍
2. 行儀の良いソフトであれば, ひとつだけディレクトリができるので, そこへ移動²
3. そのフォルダにある, ./configure コマンド (シェルスクリプト) を走らせる
4. make でコンパイル
5. make install でインストール

ソフトウェアをインストールする「正しい手順」は, まずきちんとインストール方法に関するドキュメントを読むことであるが, 実際のところ, ソースアーカイブを解凍し, 中に, configure という名前のファイル (中身シェルスクリプト) があることが確認されたら, 「あ, これは普通の慣例に従ったソースだ」と安心して, あとは上で述べた手順を反射的に踏む, というのが多くの人がやっていることである。³

2.3 configure の最重要オプション: --prefix

configure というコマンドは, コンパイルにまつわる色々な設定を行ったり, これからコンパイルを行う環境の情報を入手して適切な対処を行うためにある。configure で行われることの一つに, 最終的にプログラムがどこへインストールされるかを定める, という処理がある。

通常は, make install を行うと, /usr/local というディレクトリの下にインストールされるよう設定される。よって, 通常 make install には, /usr/local ディレクトリへの書き込み権限—ほとんどのシステムではすなわち, 管理者権限—が必要になる。もし自分が, 複数の人に使われているシステムの管理者で, 彼らを代表してそのプログラムをインストールしているのなら, make install を管理者権限で実行 (sudo make install) すれば良い。⁴ だが今回のような実験目的にはそれは面倒だし, 後でインストールしたプログラムを消去したりするのも不確実な操作になる。そこでそうする代わりに, 専用のインストール先フォルダを作り, そこにインストールするのがよい。configure スクリプトに, --prefix というオプションを与えると, インストール先フォルダを指定することができる。

例えば, インストール先を/home/denjo/gnuplot_install にする場合の手順は以下の通りとなる。

```

1 $ tar xvf gnuplot-5.0.1.tar.gz
2 $ cd gnuplot-5.0.1
3 $ ./configure --prefix=/home/denjo/gnuplot_install
4 $ make
5 $ make install # sudo は不要

```

なお, このテキストでは, 自分で入力すべき部分に下線を付けて表示する。\$ はシェルのプロンプトである。これは, コマンドを誰 (シェルなのか, gdb なのか, gnuplot なのか) に向かって打ち込むのかをはっきりさせるために表示している。\$ はシェルに向かって打ち込むことを示している。

この手順を経ると/home/denjo/gnuplot_install ディレクトリの下に, bin, libexec, share の3つのディレクトリが作られ, 目当てである gnuplot コマンドは bin ディレクトリの下にインストールされている。

² 全く必然性はないのだが, Windows 向けのアーカイブ (zip ファイルなど) では, 解凍するとカレントディレクトリに多数のファイルや, ディレクトリをぶちまけるものが多い。用心して, 空のディレクトリをひとつ作って作業するに越したことはない。

³ 多分

⁴ その場合でも, それ以前の手順 (configure と make) は通常のユーザ権限で行うのが普通。

```

1 $ cd /home/denjo/gnuplot.install
2 $ ls
3 bin/ libexec/ share/
4 $ ls bin
5 gnuplot*

```

なお, gnuplot-5.0.1 のディレクトリ構成を見てみると, トップレベルに数十のファイルとディレクトリがある. ソースは/home/denjo/gnuplot-5.0.1 に展開されているものとする.

```

1 $ cd /home/denjo/gnuplot-5.0.1
2 $ ls
3 BUGS          Makefile.in      aclocal.m4       demo/            src/
4 ChangeLog     Makefile.maint   compile          depcomp*         stamp-h
5 CodeStyle     NEWS            config/          docs/            stamp-h1
6 Copyright     PATCHLEVEL       config.h          install-sh*      term/
7 FAQ.pdf       PGPKEYS          config.hin        m4/              tutorial/
8 GNUmakefile   PORTING          config.log        man/             win/
9 INSTALL       README           config.status*    missing*
10 INSTALL.gnu   RELEASE_NOTES    configure*         mkinstalldirs*
11 Makefile      TODO             configure.in*      pm3d/
12 Makefile.am   VERSION          configure.vms*     share/

```

ディレクトリ名から察するに, src の下にソースファイルが入っているのではないかと思われ, そこを見てみるとやはり多数の .c ファイルや .h ファイルが並んでいる.

```

1 $ cd src
2 $ ls
3 GNUmakefile  datafile.c      graphics.h       parse.c          stdfn.c
4 Makefile.am  datafile.h      help.c           parse.h          stdfn.h
5 Makefile.in  dynarray.c      help.h           plot.c           strftime.c
6 Makefile.maint dynarray.h      hidden3d.c       plot.h           syscfg.h
7 NeXT/        eval.c          hidden3d.h       plot2d.c         tables.c
8 OpenStep/    eval.h          history.c        plot2d.h         tables.h
9 alloc.c      external.c      internal.c       plot3d.c         tabulate.c
10 alloc.h      external.h      internal.h       plot3d.h         tabulate.h
11 axis.c       fit.c           interpol.c       pm3d.c           template.h
12 axis.h       fit.h           interpol.h       pm3d.h           term.c
13 beos/        gadgets.c       libcerf.c        qtterminal/      term.h
14 bf_test.c    gadgets.h       libcerf.h        readline.c       term_api.h
15 bitmap.c     genopt.com      linkopt.vms      readline.h       time.c
16 bitmap.h     getcolor.c      makefile.all     save.c           unset.c
17 boundary.c   getcolor.h      makefile.awc     save.h           util.c
18 boundary.h   gnuplot.opt     matrix.c         scanner.c        util.h
19 breaders.c   gp_hist.h       matrix.h         scanner.h        util3d.c
20 breaders.h   gp_time.h       misc.c           set.c            util3d.h
21 color.c      gp_types.h      misc.h           setshow.h        variable.c
22 color.h      gpexecute.c     mouse.c          show.c           variable.h
23 command.c    gpexecute.h     mouse.h          specfun.c        version.c
24 command.h    gplt_x11.c      mousecmn.h       specfun.h        version.h
25 contour.c    gplt_x11.h      multiplot.c      standard.c       vms.c
26 contour.h    graph3d.c       multiplot.h      standard.h       win/
27 datablock.c  graph3d.h       national.h       stats.c          wxterminal/
28 datablock.h  graphics.c      os2/             stats.h          x11.opt

```

ざっと行数を数えてみると, サブディレクトリを含めずに, 99473 行ある.

```

1 $ wc *.c *.h

```

```

2      128      578      3719 alloc.c
3      2186     9863     70457 axis.c
4      185      732      4367 bf_test.c
5
6      ... <中略> ...
7
8      96       587      4048 util3d.h
9      131      465      3930 variable.h
10     52       300      1958 version.h
11     99473    375340    2945118 合計

```

10 万行はこの手の多機能なソフトウェアとしては決して大規模ではないが、それでもある変更をしようと思った時に、「まずはソースコードをじっくり眺める」という作戦で挑みたくはない規模である。したがってデバッガで、プログラムの実行の様子を追跡することが必須になる。

2.4 デバッガで追跡しやすいコンパイル方法

デバッガでプログラムの動きを追跡しやすくするためには、以下の二つのオプションを、コンパイラ (gcc や g++) に渡す。

-g: 実行可能ファイルに、「デバッグシンボル」を含める

-O0: 最適化を最低レベルにする

両方共、デバッガを使うために必須というわけではないが、この二つのオプションは事実上、つけるものと思っておくのが良い。

「デバッグシンボル」とは、大雑把に言えば、コンパイル後の命令列とソースコード上の位置 (ファイル名と行番号) の対応関係のことで、これがあるとデバッガが、プログラム実行位置を、ソースコード上の位置 (ファイル名と行番号) で表示してくれる。-g を付けないと、デバッグシンボルが含まれないため、プログラムをデバッガで実行することはできるものの、現在実行中の位置などをソースコードと対応付けて表示させることが出来ない。

-O0 を付けないと (正確には、最適化レベルを落とさないで)、プログラムの実行順序が最適化によって、ソースコード上の見た目と大きく食い違ってしまい、デバッガで追跡する際に混乱させられることが多い。またデバッガには、変数や式の値を表示させる機能があるが、最適化すると、多くの変数や式の値が表示できなくなるなど、不都合が大きい。そのため何とかして最適化レベルを最低にしてコンパイルするのが良い。

ではどうやって-O0, -g を C コンパイラのオプションに渡すかだが、多くの configure スクリプトは、実行時に環境変数 CFLAGS が定義されていると、それを C コンパイラに渡してくれる。ただし、それではうまく行かないソフトも存在するし、その指定の仕方も異なる場合があり、あまり必ずこうだと決めつけてはいけない。configure スクリプトとの付き合い方については第 A.4 節で簡単に説明する。

実際にどのようなオプションでプログラムがコンパイルされているかは、make によって表示されるコマンドラインを観察するとわかる。多くの場合、自分では決してやらないような長いコマンドラインで、gcc やら g++ やらが実行されている様子が表示される。具合が悪くなりそうなのをこらえつつ、それらに-O0 や-g がきちんとわたっているかを観察する。なお、ときどきこれを隠して表示しないよう書かれたソフトウェアも存在する。対処方法を含め、make との付き合い方は第 A.3 節で説明する。

なお、多くのオープンソースのソフトウェアでは、何も指定しなければ、-O2 と-g を付けてコンパイルされるようである。ここまですとまとめると、推奨手順は図 2 の通りとなる

準備課題 2.2 gnuplot 5.0.1 を、-O0 -g をつけてビルドして、インストールせよ (-O0 = マイナス オー ゼロ)。なお以降では、/home/denjo/gnuplot-5.0.1 にソースが展開されており、/home/denjo/gnuplot_install にインストールされているという前提でコマンドなどを表示する。


```

1 $ tar xvf gnuplot-5.0.1.tar.gz
2 $ cd gnuplot-5.0.1/
3 $ CFLAGS="-O0 -g" ./configure --prefix=/home/denjo/gnuplot_install
4 $ make
5 $ make install

```

Figure 2: (推奨) デバッガで追跡するためのビルド手順

知って得する情報: 必須ではないが知っておくと良い情報として, `make` に `-j` 数というオプションを与えると, 指定された数だけのコマンドを並列に実行してくれ, コンパイルが短時間で終わる。「数」としては, 自分が使っているマシンのプロセッサ数を指定すると良い. プロセッサ数は, `top` や, `cat /proc/cpuinfo` などで調べられる. 例えば 8 であれば,

```

1 make -j 8

```

とする.

なお, 一度 `-O0`, `-g` を付けずにコンパイルしたあと, もう一度 `./configure` と `make` を実行しても, `make` は, すでにコンパイルはすんでいるものとしてやり直してくれない. あくまでコンパイルをやり直す場合, `make` を実行する前に, `make clean` を実行する. これは, 以前にコンパイルした結果, 生成されたファイルを消すコマンドである. つまり手順としては以下になる.

```

1 $ CFLAGS="-O0 -g" ./configure --prefix=/home/denjo/gnuplot_install
2 $ make clean
3 $ make
4 $ make install

```

2.5 デバッガで追跡する

ひとたび実行したいファイルができれば, まずは動作を確かめてみよう. 通常通りコマンドとして実行すればお馴染みの動きをするはずである.

```

1 $ cd /home/denjo/gnuplot_install/bin
2 $ ./gnuplot
3
4      G N U P L O T
5      Version 5.0 patchlevel 1      last modified 2015-06-07
6
7      Copyright (C) 1986-1993, 1998, 2004, 2007-2015
8      Thomas Williams, Colin Kelley and many others
9
10     gnuplot home:      http://www.gnuplot.info
11     faq, bugs, etc:    type "help FAQ"
12     immediate help:    type "help" (plot window: hit 'h')
13
14 Terminal type set to 'x11'
15 gnuplot> plot sin(x)
16 plot sin(x)
17 gnuplot>

```

これが確認できたら次はデバッガ (GNU Debugger; `gdb` コマンド) で実行を追う. `gdb` は通常のコマンド同様コマンドラインから実行することもできるが, 不便なので Emacs から実行する.

2.5.1 Emacs で gud-gdb コマンド

まず, Emacs で gud-gdb コマンドを実行 (M-x gud-gdb と入力) する.⁵

```
U:--- doss textbook.tex [
M-x gud-gdb
```

実行可能ファイルを指定すると言われるので, 入力する.

```
U:**- 0 All L25 (Shell:run)
Run gud-gdb (like this): gdb --fullname ファイル名
```

実行可能ファイルの入力の仕方は 2 種類ある.

入力方法 1: 絶対パス名 (/から始まる) で入力する.

入力方法 2: 相対パス名 (/以外から始まる) で入力する. この場合, gud-gdb コマンドを実行したバッファのカレントディレクトリ (current directory) からの相対パスになる.

一番便利なのは, 入力方法 2 において, 目当ての実行可能ファイルが存在するディレクトリをカレントディレクトリとするようなバッファ上で, gud-gdb コマンドを実行する方法. そのためには, 目当てのコマンドが存在するディレクトリ (/home/denjo/gnuplot_install/bin) に, 何でも良いからファイルを作るとか, Emacs の shell コマンド (M-x shell) を実行して, そのシェルの中で目当てのディレクトリに移動すれば良い.⁶ こうすると入力するパス名が短くなるのみならず, 最初から候補として gnuplot コマンドを表示しておいてくれる.⁷

```
U:**- 0 All L26 (Shell:run)
Run gud-gdb (like this): gdb --fullname gnuplot
```

リターンキーを押して, gdb のプロンプトが現れることを確認する. またプロンプトの直前の, Reading symbols from ... というメッセージを確認する.

```
1 Current directory is /home/denjo/gnuplot_install/bin/
2 GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
3
4 ... <中略> ...
5
6 For help, type "help".
7 Type "apropos word" to search for commands related to "word"...
8 Reading symbols from gnuplot...done.
9 (gdb)
```

最後の行が, Reading symbols from gnuplot...done. のようになっている場合, デバッグシンボルが無事読み込まれたということである.

読み込めなかった場合以下のようなメッセージが現れる.

```
1 ... <前略> ...
2
3 Reading symbols from gnuplot...(no debugging symbols found)...done.
```

これが出たらコンパイル時に -g が付けられていない可能性が高い.

⁵M-x の入力方法: Alt キーを押しながら x を押す, または, C-[を押してから x を押す.

⁶これは, Emacs の中で何もかもをやろうとする, Emacs 内引きこもり症な人であれば, 最初からやっていることである.

⁷おそらく, そのディレクトリ中の最新の実行可能ファイルを, 候補にしている

2.5.2 break コマンドで停止位置を設定

実行を開始するに先立って、プログラムを停止させる場所—ブレークポイント—を設定する。今は実行を最初から追跡したいので、main 関数にブレークポイントをセットする。そのために、break コマンド (b だけでも良い) を用いる。以降の説明では、正式なコマンド名 (break) とその省略形 (b) を合わせて、b(reak) のように書くことにする。

```
1 (gdb) b main
2 Breakpoint 1 at 0x486d4c: file plot.c, line 282.
```

こうすることで次にプログラムを実行した際、main 関数の実行に差し掛かったところで実行が停止する。なおここでもデバッグシンボル付きでコンパイルされていることの確認として、

```
1 Breakpoint 1 at 0x486d4c: file plot.c, line 282.
```

のように、ファイル名と行番号が表示されていることを確認せよ。これが出れば、正しくデバッグシンボル付きでコンパイルされているということである。デバッグシンボルがないと、

```
1 (gdb) b main
2 Breakpoint 1 at 0x486d4c
```

というだけの表示になる。

2.5.3 run コマンドで実行開始

実行する。そのためには、r(un) コマンドを用いる。

```
1 (gdb) r
```

すると、main 関数の先頭で実行が止まり、かつ、Emacs 内にソースファイルが自動的に読み込まれる。

```
emacs@nanamomo
File Edit Options Buffers Tools C Gud Help

#ifdef LINUXVGA
    LINUX_setup();                /* setup VGA before dropping privilege DBT 4/5/9 */
#endif
/* make sure that we really have revoked root access, this might happen if
   gnuplot is compiled without vga support but is installed suid by mistake */
#ifdef _linux_
    setuid(getuid());
#endif

#if defined(MSDOS) && !defined(_Windows) && !defined(_GNUC_)
    plot.c 24% L282 (C/l Abbrev)
Starting program: /home/tau/public_html/lecture/dive_to_oss/projects/gnuplot_ins
tall/bin/gnuplot
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffff428) at plot.c:282
(gdb)

U:*** *gud-gnuplot* Bot L25 (Debugger:run)
```

2.5.4 next コマンドで一行実行

ここからプログラムを一行ずつ実行するには、n(ext) コマンドを実行する。

```
1 (gdb) n
```

するとプログラムが現在行の次の行まで実行される。

プログラムを少しずつ実行するにも様々なやり方があり、目当ての部分にたどり着くにはそれらを正しく駆使しなくてはならない。次の節ではそれらを含め、ソースコード上で変更が必要な部分を突き止めるために必要なコマンドを説明する。

準備課題 2.3 -O0, -g をつけてビルドされた `gnuplot` コマンドを、`gdb` で実行せよ。main にブレークポイントを設定し、そこから `n(ext)` コマンドで何行か追跡してみよ。

3 実録! gnuplot を変更する

3.1 目標の再確認

前節までで、デバッガで実行を追うための準備が整った。そこで実際に、目的を設定してプログラムの実行を追跡していくことにしよう。ここで設定する目標は、

グラフをプロットする際に、“plot” というキーワードを省略できることにする

ということにする。つまり、`gnuplot` のプロンプトに、

```
gnuplot> sin(x)
```

とだけ打ち込めば、それが、

```
gnuplot> plot sin(x)
```

と打ち込まれたのと同様に動作する、ということである。

`plot` はよく使うコマンドだからいちいち入力したくない、というのがその変更の正当化だが、果たしてこれが本当に良い変更なのか、むしろコマンド名を入力し間違えた時に、おかしい動作をして混乱を招く、ダメな変更なのではないか、などということを、ここで真剣に問うて教員を困らせてはいけない！これはあくまで練習のため、である。

準備課題 3.1 この機能を実現せよ。ただし、答えは以降を読めば書いてある。自分で試行錯誤しつつできそうな人は以降を読まずにやってもよいし、以下で基本知識を習得しながらやりたい人は以下に沿ってやってもよい。

3.2 基本作戦—next と step

この変更をするにあたって再び、「ソースコードをなんとなく眺めて、関係がありそうなところを探す」という作戦はいかにもやりたくない。そうではなくやりたいのは、ファイル中に書かれていた文字列がどこで読み取られ、どこで “plot” というコマンド名が認識され、やがて実際のプロット処理に移っていくか、その過程を追うことである。そしてどの部分に介入すれば所望の機能が実現できるかを突き止めることである。そのためにデバッガが力を発揮する。何を目的とする場合でも、基本的な方法は以下のとおりである。

1. 基本的には `n(ext)` コマンドで、一行一行実行していく
 2. 中身を追跡したい関数呼び出しを含む行に達したら、`s(tep)` コマンドでその関数の中に入っていく
- `n(ext)` コマンドと `s(tep)` コマンドの違いは以下のとおりである。今実行が、

```
1 => f(x);
2   ...
```

という行に達したとする ($f(x)$ がある行をこれから実行する。以降、「これから実行する行」の脇に \Rightarrow をおいて示すことにする)。ここで $n(\text{ext})$ コマンドを実行すると、上記の... があるところで実行が止まる。言い換えれば、実行は $f(x)$ の中では止まらずに実行される。一方ここで $s(\text{tep})$ コマンドを実行すると、実行は $f(x)$ の最初の行で停止する。つまり $s(\text{tep})$ コマンドは問答無用で「現在いるのと異なる行」で止まる。 $n(\text{ext})$ コマンドは「現在いる次の行」で止まる。例えて言うならば、 $s(\text{tep})$ は各駅停車、 $n(\text{ext})$ は(関数の中でいちいち止まらない)快速である。この二つを駆使して、「関係ない関数は $n(\text{ext})$ で通過、関係ありそうな関数は $s(\text{tep})$ で各駅停車」で、関係する場所に到達する。

3.3 実行開始

まずはデバッガで `gnuplot` を立ち上げる。

```
1 (emacs で) M-x gud-gdb
```

```
1 Run gud-gdb (like this): gdb --fullname gnuplot
```

次に `main` 関数にブレークポイントを設定する。

```
1 (gdb) b main
2 Breakpoint 1 at 0x486dd8: file plot.c, line 282.
```

次に、引数として中身が、

```
1 plot sin(x)
```

であるようなファイル `a.gnuplot` を作り、それを引数に渡して実行を開始する。それには `r(un)` コマンドの引数に、`a.gnuplot` を与えれば良い。

```
1 (gdb) r a.gnuplot
```

なお、引数でファイルを与える代わりに `gnuplot` のプロンプトに向かって、`plot sin(x)` と入力しても良いが、毎回打ち込む手間を無くしたいということと、`gdb` の中にさらに、`gnuplot` のプロンプトが出てくることで生ずる混乱を避けるため、`gnuplot` の入力、ファイルから読み込ませる。

無事 `main` 関数の先頭らしきところで実行が停止したら、

```
1 /* make sure that we really have revoked root access, this might happen if
2    gnuplot is compiled without vga support but is installed suid by mistake */
3 #ifdef __linux__
4 =>   setuid(getuid());
5 #endif
```

あとはここから `next` コマンドを何度も実行して、「あまり関係なさそうなところ」をどんどん実行していく。なお、`gdb` のプロンプトに空行(つまり、リターンキーのみ)を入力すると、直前のコマンドを再実行するので、一度 $n(\text{ext})$ コマンドを実行したらあとはリターンキーを連打すれば良い。

関係ありそうな行—つまり、その行で行われている関数呼び出しの中で、関係する処理が行われているような行—to到達したら、 $n(\text{ext})$ ではなく、 $s(\text{tep})$ で、中に分け入っていく。

3.4 next か? step か?

「あまり関係なさそうなところ」を $n(\text{ext})$ で、関係ありそうなら $s(\text{tep})$ で、と書いたが、ここである行が「関係なさそうか、ありそうか」を判断するにはどうしたらよいか。まず、あまり心配しなくても、関数名、周辺の処理、コメントなどから想像できることが多い。⁸ また、とりあえず関係なさそうと思って先へ

⁸つまりここは人工知能ではなく、神もしくは偉大なる親御様から授かった天然の知能を使って判断する

進め、あきらかに「通り過ぎた」と思ったら、もう一度最初からやり直す、という方法も有用である。例えば我々の現在の目的からは、窓が現れて `sin` 関数のグラフが表示されてしまったら、「明らかに通り過ぎた」と言える。そうになったら `r(un)` コマンドをもう一度発行してスタートからやり直す。「もう一度最初からやり直す」作業を効率的に行う方法は、第 3.5 節で説明する。

実際にはプログラムを書いた人間をある程度信頼すれば、「これは明らかに無関係」という関数はある程度想像はつくものである。例えば最初にプログラムが停止した位置で実行されるのは、

```
1 /* make sure that we really have revoked root access, this might happen if
2    gnuplot is compiled without vga support but is installed suid by mistake */
3 #ifdef __linux__
4 =>    setuid(getuid());
5 #endif
```

というところである。setuid, getuid という関数が何をするかを知っていれば (または man ページで調べれば) ここで入力を処理しているわけではないと断言できる。その上のコメントを見てもおそらくここで入力を処理しているわけではないだろうと 99% 想像できる。また、`#ifdef __linux__` となっている時点で、この行は linux 以外なら実行されないであろう、したがって (どんな OS でも実行されるべき) 入力の処理がここで行われているとは考えにくい。そこでこの行は 99% の自信を持って、`next` で通過する。⁹

この行を飛ばしたあとで遭遇するのは以下のループの先頭で、

```
1 =>    for (i = 1; i < argc; i++) {
```

これは見るからに、コマンドライン引数の処理をしていると思われるループである。

`n(ext)` を何度か実行してこのループを抜け、さらにしばらく `n(ext)` を実行していった時に遭遇する行を、適当につまみ出したのが以下である。いずれもその関数名や、そのまわりの雰囲気から、「まだ本題 (入力の処理) ではない」と想像できる。例えば `init_constants()` のようないかにも何かを「初期化 (initialize)」していると思しき関数の中で、実際にファイルが読み込まれているとは考え難い。これらもどんどん `n(ext)` で通過していく。

```
1 setbuf(stderr, (char *) NULL);
2 if (setvbuf(stdout, (char *) NULL, _IOLBF, (size_t) 1024) != 0)
3 setvbuf(stdin, (char *) NULL, _IONBF, 0);
4 (void) add_udv_by_name("GNUTERM");
5 init_constants();
6 init_memory();
7 init_terminal(); /* can set term type if it likes */
8 show_version(NULL); /* Only load GPVAL_COMPILE_OPTIONS */
9 interrupt_setup();
10 get_user_env();
11 init_loadpath();
12 init_locale();
13 init_fit(); /* Initialization of fitting module */
14 init_session();
```

それらの処理を抜け、次に到達するのは以下の `while` 文で、

```
1 /* load filenames given as arguments */
2 while (--argc > 0) {
3     ++argv;
```

その上のコメント (「引数で与えられたファイルをロードする」) を一瞥すれば、ついに本題に近づいたか、という匂いを感じる。

ループの中では `*argv` という式に対して順々に、比較が行われている。

```
1 } else if (strcmp(*argv, "-e") == 0) {
```

⁹なお、仮に `step` で実行しても、`setuid` や `getuid` はのソースコードが見つからないので、結局その中に入っていくことはできない。

```
1 } else if (!strcmp(*argv, "-d", 2) || !strcmp(*argv, "--default-settings")) {
```

```
1 } else if (strcmp(*argv, "-c") == 0) {
```

デバッガでは式の値を `p(rint)` コマンドで表示することができる。そこで `*argv` の値を表示させると、

```
1 (gdb) p *argv
```

```
2 $1 = 0x7fffffff81d "a.gnuplot"
```

今見られているのがまさしく引数で与えた `a.gnuplot` であることがわかり、ますます目的地が近づいている匂いを感じる。上の条件式が全て偽となったあと、実行は `else` 節の以下の文に達する。

```
1     interactive = FALSE;
```

```
2     noinputfiles = FALSE;
```

```
3 => load_file(loadpath_fopen(*argv, "r"), gp_strdup(*argv), 4);
```

様々な周辺状況から、この行はクサイ (= この中でファイルの中身が読み込まれている) と判断できる。

- `load_file` という関数名がクサイ
- ファイル名 (`*argv = a.gnuplot`) を元に引数を渡しているところがクサイ
- この行が終わるとループの次の行へ進む、したがってこの行以外、このループ内で他にクサイ行が見当たらない。

そこで、ここでの次の一手は `step` ということになる。

3.5 行き過ぎてから、やり直す方法

なお、この行の関数呼び出し (`load_file`) がクサイということを見逃してここで `n(ext)` を実行すると、窓が出てきて $\sin(x)$ のグラフが表示される。そこへ来て、やはりこの関数がそれだったのかと思い直す、ということもあり得る。というよりも、実際の追跡の現場では、必ず何度かはそのような「行き過ぎ」をやってしまうものである。その場合もう一度最初からやり直す必要がある。その方法には主に、

方法 1: `main` から `n(ext)` を連発して同じ所までたどりつく

方法 2: 目当ての行:

```
1 load_file(loadpath_fopen(*argv, "r"), gp_strdup(*argv), 4);
```

にブレークポイントを設定して実行する、

という二つがある。

方法 1 は、なるべくすぐに卒業したほうが良い。以下、方法 2 について。ある行にブレークポイントを設定するには、Emacs 上で、その行にカーソルをおいて、

```
1 (emacs に) C-x SPACE
```

を入力する。¹⁰ または `gdb` のプロンプトに向かって、「ファイル名:行番号」の書式で、ブレークポイントの位置を指定する。

```
1 (gdb) b plot.c:654
```

この状態で `r(un)` コマンドをもう一度実行する。今回の `run` コマンドには、引数 `a.gnuplot` を与えなくて良い。前回与えたものが今回も自動的に与えられる。すると、実行が最初からやり直され、(先ほど設定した) `main` 先頭のブレークポイントで停止する。そこから、`c(ontinue)` コマンドを実行すると、次のブレークポイントまで実行してくれる。

¹⁰Ctrl を押しながらか x; その後 (ctrl は離して)SPACE. もしここで SPACE も ctrl を押しながらか押せという場合は、C-x C-SPACE と表記される

```
1 (gdb) r
2 (gdb) c
```

c(ontinue) コマンドは、「次のブレークポイントまで実行する」というもので、最も柔軟かつ汎用的な実行の仕方である。

一度必要な場所にブレークポイントを設定しておけば、気軽に実行を頭からやり直すことができる。
なお、頭から実行する代わりに、

方法 3: gdb の逆実行機能 (reverse-step, reverse-next, など) を利用する

という方法もある。これは、プログラムの実行を少し巻き戻すという、不思議な機能である。場合によっては有用だが、原理的に巻き戻しなどできるはずがない (例えば、一度画面に出力してしまった文字列を引っ込めることは出来ない) 場合もある。ここではこの謎な機能には頼らないことにするが、興味のある人は探求してみると良い。

3.6 step で奥へ進む

さて、いよいよ

```
1 load_file(loadpath_fopen(*argv, "r"), gp_strdup(*argv), 4);
```

の行で実行される関数呼び出しの中身に入っていくことにする。そのために s(tep) コマンドを実行する。

```
1 (gdb) s
```

すると実行は目当ての load_file ではなく、gp_strdup(const char *s) という関数の先頭で停止する。

```
1 char *
2 gp_strdup(const char *s)
3 {
4     char *d;
5
6 => if (!s)
7     return NULL;
8
9 #ifndef HAVE_STRDUP
10     d = gp_alloc(strlen(s) + 1, "gp_strdup");
```

よく考えればそれもそのはずで、もとい行:

```
1 load_file(loadpath_fopen(*argv, "r"), gp_strdup(*argv), 4);
```

では load_file 以外にも二つの関数呼び出し loadpath_fopen(*argv, "r"), gp_strdup(*argv) が実行されており、当然のことながらこれらのほうが load_file よりも先に実行される。

loadpath_fopen(*argv, "r") と gp_strdup(*argv) はどちらが先に実行されてもおかしくないが、今はたまたま後者が先に実行されたわけである。

3.7 finish コマンドで関数から抜けるまで実行

今我々が主に興味があるのは load_file の方で gp_strdup の中身には余り興味がない。名前からも想像され、関数の中身を見てもわかるとおり、これは文字列をコピーするだけの関数である。

そのような関数の中に入ってしまったら、n(ext) コマンドを使って抜けることもできるが、一番手っ取り早いのは、fin(ish) コマンドを使うことである。fin(ish) を使うと、その関数の実行が終了したところで止まる。

```
1 (gdb) fin # gp_strdup 終了まで
```


すると実行は再び元の行に戻る。もう一度 `s` を実行すると今度は、`loadpath_fopen` の中に入るから再び `fin` で元に戻る。ここでもう一度 `s` を実行すると、めでたく `load_file` の先頭に到着する。

```
1 (gdb) s      # loadpath_fopen に突入
2 (gdb) fin     # loadpath_fopen 終了まで
3 (gdb) s      # load_file に突入
```

```
1 load_file(FILE *fp, char *name, int calltype)
2 {
3     int len;
4
5     int start, left;
6     int more;
7 => int stop = FALSE;
8
9     /* Provide a user-visible copy of the current line number in the input file */
10    udvt_entry *gpval_lineno = add_udv_by_name("GPVAL_LINENO");
```

3.8 再び `next` で関係ないところを通過～`do_line`, `command` までの旅

ここから再び `n(ext)` 連発で関係ないところを通過する旅を続ける。

```
1 (gdb) n
```

ここは実際に我々のファイル `a.gnuplot` が読み出されているところだから、どのへんでそれが実際に行われているかに注意しつつ、実行を追っていく。すると、以下のような行に遭遇する。

```
1 if (fgets(&(gp_input_line[start]), left, fp) == (char *) NULL) {
```

`fgets` はファイルから一行読み込む関数で、第一引数が読みこむ先のアドレスである。したがってこの行を通り過ぎると、`&gp_input_line[start]` というアドレスに、ファイルの中身が一行分、読み込まれていると期待できる。もう一行実行したあとで、実際に内容を表示してみると、期待通りの結果を観測できる。

```
1 (gdb) n
2 (gdb) p &(gp_input_line[start])
3 $4 = 0x7bf8d0 "plot sin(x)\n"
```

このように自分たちの理解に自信を深めながら旅を続けていく。途中で以下のような、「少し意味有りげな」行たち

```
1 => string_expand_macros();
```

```
1 => num_tokens = scanner(&gp_input_line, &gp_input_line_len);
```

にも遭遇するが、構わず先へ進めると、より本命臭い行に到達する。

```
1 => if (do_line())
2     stop = TRUE;
```

`do_line()` という名前からしていかにもクサイ。ちなみにここで `n(ext)` を実行するとやはり窓が現れ、「通り過ぎた」ことが確認できる。

そこで `do_line()` の中に `s(tep)` で分け入っていく。するとほどなくしてやはり見るからに怪しい行

```
1 while (c_token < num_tokens) {
2 =>     command();
3     if (command_exit_status) {
```

に到達する。そこでこの `command()` 関数内に `s(tep)` で分け入る。`command()` 関数内で最初に遭遇するのは次のループである。

```
1 =>   for (i = 0; i < MAX_NUM_VAR; i++)
2       c_dummy_var[i][0] = NUL;          /* no dummy variables */
```

これを `n(ext)` 連発で実行していくと、ループの回数 $\times 2$ 分連打が必要になるので辛い。

3.9 until コマンドでループの終わりまで通過

このようなループから手っ取り早く脱出するには、`u(ntil)` コマンドを使えば良い。`u(ntil)` コマンドは、`n(ext)`、`s(tep)` の変種で、手っ取り早く言うと、「ループの繰り返しを 2 度以上実行しない」。したがって `n(ext)` の代わりに `u(ntil)` を連打すれば、同じループの中身はただかき一回実行するだけでループを脱出することができる。

なお、`u(ntil)` コマンドを用いる代わりに、ループを抜けたところにブレイクポイントを設定した上で、`c(ontinue)` コマンドを実行しても良い。

3.10 ついにボスキャラ (`lookup_ftable`) に遭遇!?

どちらにしてもしばらく実行すると、益々クサイ以下の行にたどり着く。

```
1 (*lookup_ftable(&command_ftbl[0],c_token))();
```

ここで C にあまり慣れていない人のためにこの行の読み方を説明しておく。全体は、引数のない関数呼び出しである。ただし関数のところが単なる関数名ではなく、ややこしい式になっている。

```
1 (*lookup_ftable(&command_ftbl[0],c_token))      ();
2 .....
3          関数                                引数 (なし)
```

呼び出される関数自身が、

```
1 lookup_ftable(&command_ftbl[0],c_token)
```

という関数呼び出しによって計算 (取得) されている。`lookup_ftable` という関数名からは、何か表を検索しているものと想像される。そうして表を検索して返ってくるものが関数で、その関数を呼んでいる、というのが元々の処理である。ばらして書けば以下のようなことをやっているわけである。

```
1 f = lookup_ftable(&command_ftbl[0],c_token);
2 f();
```

`lookup_ftable` が、行の内容に応じて、「その行を処理する適切な関数」を返し、それが呼ばれている。つまり、コマンドに応じて行処理を振り分ける処理がここで行われていると想像される。

ではまず、`s(tep)` によって、`lookup_ftable` 関数の中身に入っていく。すると中身は

```
1 parsefuncp_t
2 lookup_ftable(const struct gen_ftable *ftbl, int find_token)
3 {
4 =>   while (ftbl->key) {
5       if (almost_equals(find_token, ftbl->key))
6           return ftbl->value;
7       ftbl++;
8   }
9   return ftbl->value;
10 }
```

という短いもので、なおかつ行っていることもいかにも、表 (ftbl) を検索しているような処理になっている。ついでながら、返される値の型 `parsefunc_t` も、いかにも関数の型っぽい名前 (`...func...`) であることも確認しておこう。

我々がやりたかったこと—plot を省略した時に、plot があつたかのように動作をする—を思い起こすと、目当ての場所に限りなく近づいた感がある。要はこの表内に、適切な処理が見つからなかった場合の動作を変更すればよいのである。

今、実際に検索されている表 ftbl の中身を、`p(rint)` コマンドで覗いてみよう。

```
1 (gdb) p ftbl
2 $5 = (const struct gen_ftable *) 0x563540 <command_ftbl>
```

アドレスだけ表示されても意味はないので、じっさいに中身を表示させてみる。例えば `ftbl[0]` は先頭要素、`ftbl[1]` はその次の要素である。

```
1 (gdb) p ftbl[0]
2 $6 = {key = 0x56341a "raise", value = 0x415b3f <raise_command>}
3 (gdb) p ftbl[1]
4 $7 = {key = 0x563421 "lower", value = 0x415b4f <lower_command>}
```

何やら、コマンド名 (key) と、それを処理すべき関数名が現れた。この中に、plot コマンドを処理する行があるに違いないと期待される。

3.11 配列の中身を一挙に表示する方法 (@)

配列の中身を一挙に表示したければ、gdb の特別な記法として、「式@要素数」という記法がある。例えば以下で、先頭から 10 要素を一挙に表示できる。

```
1 (gdb) p ftbl[0]@10
```

と入力すれば良い。

```
1 (gdb) p ftbl[0]@10
2 $8 = {{key = 0x56341a "raise", value = 0x415b3f <raise_command>}, {key = 0x563421 "lower",
3       value = 0x415b4f <lower_command>}, {key = 0x563428 "bind",
4       value = 0x415b5f <bind_command>}, {key = 0x56342e "call",
5       value = 0x41613f <call_command>}, {key = 0x563434 "cd",
6       value = 0x4161c0 <changedir_command>}, {key = 0x563437 "clear",
7       value = 0x416253 <clear_command>}, {key = 0x56343e "do", value = 0x416e02 <do_command>},
8       {key = 0x563441 "eval", value = 0x4163cb <eval_command>}, {key = 0x56344b "exit",
9       value = 0x416417 <exit_command>}, {key = 0x563451 "fit",
10      value = 0x431588 <fit_command>}}
```

10 を 20, 40, 80, ... と倍々に大きくしていくと、そのうち、

```
1 (gdb) p ftbl[0]@80
2 ...
3 key = 0x7932780031793178 <error: Cannot access memory at address 0x7932780031793178>,
4 ...
```

のようなエラーが出現する。これは、表のサイズを越えてしまったから出ているものと想像できる。このエラーが出ないギリギリのところを二分探索¹¹で突き止めると、どうやら 45 で、ぴったり全てが表示されるようである。

```
1 (gdb) p ftbl[0]@45
2 ...
3 ...
```

¹¹具体的には、40, 80(+), 60(+), 50(+), 45, 47(+), 46(+) で 45 が見つかる。(+) は、そこでエラーが出るという意味。

```

4 {key = 0x56352a "$",
5   value = 0x41dcdd <datablock_command>}, {key = 0x0, value = 0x419575 <invalid_command>}}

```

そして、この最後のエントリの key が 0x0 となっている。これは、表の終わりを示すエントリであろうと想像でき、表を検索するループが、

```

1 while (ftbl->key) {

```

となっていることとも符合する。

そして、表示された配列をよく見ると中に、plot コマンドに該当すると思しき行も見つかる。

```

1 {key = 0x563483 "p$plot", value = 0x4179f4 <plot_command>}

```

なおあまり本題とは関係ないが、key のところが、‘plot’ではなく、‘p\$plot’という妙な文字列になっているのは、コマンドを検索する際のルールによる。実は、plot をまるごとコマンド名として与えなくても、\$の手前までを与えても、一致と見なすようになっている。すなわち p\$plot は、p とも一致する。その仕組みは、lookup_ftable の中で呼ばれている、

```

1 if (almost_equals(find_token, ftbl->key))

```

という関数の中にある。興味がある人はこの中に分け入って調べてみると良い。ということでこれから gnuplot を使う時は、plot コマンドを実行するのに、p だけで済ませられる...こうした隠れキャラのような機能が見つかるのも、ソースコードを追跡する楽しみの一つである。

さて、lookup_ftable 関数の実行終了まで、fin(ish) コマンドで実行を進める。

```

1 (gdb) fin
2 Run till exit from #0  0x00000000004de4b0 in lookup_ftable (ftbl=0x563540 <command_ftbl>,
3   find_token=0) at tables.c:738
4 0x00000000004159ec in command () at command.c:625
5 Value returned is $16 = (void (*)(void)) 0x4179f4 <plot_command>

```

なお、fin(ish) コマンドを実行すると関数の返り値も表示される。予想通り、plot コマンドを処理すると思しき関数 plot_command が返される。いよいよこれが実際に plot コマンドを処理するわけである。

3.12 ボスキャラその 2: (plot_command)

lookup_ftable が終了してから s(tep) を実行すると、plot_command の中に入って行ける。plot_command は gnuplot の中心とでも言うべきコマンドだが、表面的には意外と短い。

```

1  /* process the 'plot' command */
2  void
3  plot_command()
4  {
5  =>   plot_token = c_token++;
6       plotted_data_from_stdin = FALSE;
7       SET_CURSOR_WAIT;
8
9       #ifdef USE_MOUSE
10        plot_mode(MODE_PLOT);
11        add_udv_by_name("MOUSE_X")->udv_undef = TRUE;
12        add_udv_by_name("MOUSE_Y")->udv_undef = TRUE;
13        add_udv_by_name("MOUSE_X2")->udv_undef = TRUE;
14        add_udv_by_name("MOUSE_Y2")->udv_undef = TRUE;
15        add_udv_by_name("MOUSE_BUTTON")->udv_undef = TRUE;
16        add_udv_by_name("MOUSE_SHIFT")->udv_undef = TRUE;
17        add_udv_by_name("MOUSE_ALT")->udv_undef = TRUE;
18        add_udv_by_name("MOUSE_CTRL")->udv_undef = TRUE;
19    #endif
20    plotrequest();

```

```

20     SET_CURSOR_ARROW;
21 }

```

というよりも、実質的な中身は全て、`plotrequest()` 関数内で行われているものと想像できる。

3.13 変更の概要

ここまで追跡した範囲で、今回の目的のために変更すべき範囲は、ほぼ明らかになったと言える。

1. `plot` を省略した際に、あたかも `plot` があったかのような動作をさせるには、`lookup_fhtable` でコマンドが見つからなかった場合に、`plot_command` もしくはそれをすこし変更したようなものを返せば良い。現状では、表を最後まで探索して見つからなければ、表の最後のエントリが返され、それは `invalid_command` という関数である。その中身は、`invalid command` と表示するだけの関数である。したがってその表の最後のエントリを変更すれば良い。
2. そこで単純に `plot_command` を返すだけではきつとうまく行かない。というのも、現存する `plot_command` は、読み込んだ行の先頭に、`‘plot’` (ないし `‘p’`) という文字列が存在している場合に、正しく動作するように書かれているに違いないからである。つまり、実際に表示すべき式やデータ (ここでは `‘sin(x)’`) は、その後に来るものと仮定して処理をしているに違いないからである。今回の我々の動作を実現するには、`‘plot’` という文字列が、行頭に無いような行を処理する関数を作らなくてはならない。おそらくそれは `plot_command` 関数と良く似たものになるだろう。

3.14 変更 1: `plot_command` の変更

まず後者を実現するために、`plot_command` の中身を今一度見てみよう。

```

1  /* process the 'plot' command */
2  void
3  plot_command()
4  {
5      plot_token = c_token++;
6      plotted_data_from_stdin = FALSE;
7      SET_CURSOR_WAIT;
8  #ifdef USE_MOUSE
9      ... < 省略 > ...
10 #endif
11     plotrequest();
12     SET_CURSOR_ARROW;
13 }

```

突き止めるべきは、そもそも `plot_command` へ渡すべき入力文字列、(`‘plot sin(x)’`) が、どこから渡ってきていて、それがやがてどこで読み取られるのかということである。もう少し模範的なソースコードであればそれは `plot_command` の引数としてわたって来るべきところだが、`plot_command` が何も引数を受け取らないところを見ると、どうやらそれは大域変数として暗黙的に渡されるようである。¹²

その大域変数がなんであれ、その先頭には `‘plot’` というコマンド名が入っているはずなので、`plot_command` は、おそらくそれを読み飛ばしてそれ以降を処理するはずである。今やりたいのは、その「読み飛ばし」をしない、そして、それ以外の点においては `plot_command` と同じ動作をするような関数である。

そこで本来であればこのあと、`plot_command` 関数—実質的には `plotrequest` 関数—が、どこから入力を読み、どうやって、`‘plot’` という文字列をやり過ぎて、プロットすべき `‘sin(x)’` という文字列を発見するのか、を見ていくことになるのだが、幸いなことにここでは、`plot_command` の先頭で、まさしくその読み飛ばしをやっているのではないかと、思いき行が見つかる。

```

1  plot_token = c_token++;

```

¹²その大域変数が何であるかはこれまでソースコードを追跡してきた人であれば、`gp_input_line` ではないかと想像できる。が、今はそれはあまり重要ではない。

これが「読み飛ばし」に思えるためには多少の勘の良さと経験値が必要かもしれない。そもそも `token` というのは単語という意味である。 `plot sin(x)` という入力を読み込んだ際、それが二つの単語からなっており、最初の単語が `plot` で、次の単語が `sin(x)` で、という処理が施されている。そのことは、デバッガで追跡している際に、入力を読み込んでいるところを、少し注意深く見ていればわかるのだが、実際のところは、`token` という単語に反応して、そう想像しているという色彩が強い。また、およそあらゆる文字列をコマンドとして受け取るプログラムでは、文字列を文字列のまま扱うことは稀で、最初にそれを単語の列に切り分け (`tokenize`)、単語と単語の切れ目を認識する、ということをする。それなしには後々の処理全てが面倒になるからである。

... 等々の予備知識があるところの行が、「一単語読み飛ばしている」ように、おそらく見えるようになる。なおもちろん何事も決めつけてはいけなし、では本当にこの行を削除するだけで良いのか、は別問題である。自信を持って事を進めるためには、`c_token` という変数がその後どう使われ、実際にそれを元に、入力の `'sin(x)'` が拾い出されていること、を目撃する必要がある。 `plotrequest` 関数の中を追っていけばその現場に遭遇することができるが、今は省略する。

今はあまり深く考えずに、`plot_command` からこの行を削除した関数—`default_plot_command` と命名する—を作ってしまう。

その関数を書くファイルとしては、`plot_command` 関数が書かれているファイル—`command.c`—が適当だろう。

3.15 ソースコード変更時の基本—簡単に元に戻せるように

実際にソースコードをいじる際、後で変更点が簡単に見つかるようにすること、およびすぐに元に戻せるようにしておくのが良い。

そのための「とりあえず」のやり方として、以下のように、変更した部分を `#if 1 ...#endif` で囲っておくというやり方がある。

```
1 #if 1
2   ... 追加した部分 ...
3 #endif
```

`#if ... #end` という指示は、`if` 文ではなく、コンパイル時に、その中身をソースコードの一部として含める・含めない、を決めるものである。条件式が 0 でなければ含め、0 なら含めない。

よってこの `#if 1` の部分をあとから `#if 0` に変更すれば、追加部分を消さなくても、元のソースになる。¹³

ソースへの追加でなく、ある部分を別のものに置き換える場合も、

```
1 #if 1
2   ... 追加した部分 ...
3 #else
4   ... 既存 (削除したい) 部分 ...
5 #endif
```

とすることで、同様に対処できる。

ちなみにこの `#if 1` というやり方はかなり付け焼刃的なもので、変更が何箇所にも及ぶと、それを元に戻すときに、何箇所も変更する必要がある。もう少しマシなやり方は、第 3.22 節で説明する。

とりあえずこの付け焼き刃なやり方で、追加したコードを以下に示す。

```
1 #if 1
2 /* process the 'plot' command */
3 void
4 default_plot_command()
5 {
6     //plot_token = c_token++;
7     plotted_data_from_stdin = FALSE;
```

¹³ なお、ソースコードの一部を無効化するのに、コメント `/* */` で挟む人が多いが、`#if 0 ...#endif` で囲むほうがはるかによい。コメントで挟む方法は、中にコメントが含まれていた場合におかしなことになる。

```

8      SET_CURSOR_WAIT;
9  #ifdef USE_MOUSE
10     plot_mode(MODE_PLOT);
11     add_udv_by_name("MOUSE_X")->udv_undef = TRUE;
12     add_udv_by_name("MOUSE_Y")->udv_undef = TRUE;
13     add_udv_by_name("MOUSE_X2")->udv_undef = TRUE;
14     add_udv_by_name("MOUSE_Y2")->udv_undef = TRUE;
15     add_udv_by_name("MOUSE_BUTTON")->udv_undef = TRUE;
16     add_udv_by_name("MOUSE_SHIFT")->udv_undef = TRUE;
17     add_udv_by_name("MOUSE_ALT")->udv_undef = TRUE;
18     add_udv_by_name("MOUSE_CTRL")->udv_undef = TRUE;
19 #endif
20     plotrequest();
21     SET_CURSOR_ARROW;
22 }
23 #endif

```

3.16 変更 2: lookup_ftable の変更

次に、plot キーワードがない場合に、今定義した default_plot_command 関数が (invalid_command の代わりに) 呼ばれるようにする。そのために、lookup_ftable でコマンドが見つからなかった時に返される関数—つまり、以下の ftbl の最後のエントリに格納されている value—に、invalid_command ではなく、default_plot_command を入れておく。

```

1  parsefuncp_t
2  lookup_ftable(const struct gen_ftable *ftbl, int find_token)
3  {
4      while (ftbl->key) {
5          if (almost_equals(find_token, ftbl->key))
6              return ftbl->value;
7          ftbl++;
8      }
9      return ftbl->value;
10 }

```

そこで、そもそもこの ftbl はどこでどう作られているのか、ということ突き止める。まずそのための基本として、ftbl が lookup_ftable の引数として渡されていることに注意しよう。したがって lookup_ftable を呼び出しているところで、何を渡しているかをさかのぼって見つけ出す必要がある。それには、lookup_ftable に s(tep) コマンドで入って行く前の行:

```

1  (*lookup_ftable(&command_ftbl[0], c_token))();

```

を自分で思い出せばよいのだが、もちろん一般にはそんなことを思い出せるものではない。

3.17 up コマンドで関数呼び出し元を表示する

そのような場合、up コマンドで、現在実行が停止している関数の呼び出し元を表示させることができる。

つまり、lookup_ftable 関数内や、それが返した plot_command 関数内で停止中に、up コマンドを実行すると、Emacs の画面が再び、

```

1  => (*lookup_ftable(&command_ftbl[0], c_token))();

```

を表示してくれる。up コマンドは、現在注目している関数に、引数として渡された値が、元はといえばどうやって計算されたかを逆に追跡して行くのに有用である。

`lookup_fhtable` の第一引数で使われている `command_ftbl` という、配列と思しき変数は、`command()` 関数の引数でも、局所変数でもないので、おそらく大域変数であろうと思われる。¹⁴

3.18 etags コマンド + find-tag で定義へ飛ぶ

そこでソースコード中で、大域変数 (配列) `command_ftbl` を見つけたい。そのための方法:

方法 1: 原始的に、`src/`ディレクトリ内で

```
1 $ grep command_ftbl *.c
```

などとする。定義も使用も同じように見つかってしまうので、推奨しない。

方法 2: `etags` というコマンドを使う (注: `gdb` のコマンドではなく、通常のシェルから実行するコマンド)。これは、関数名や大域変数の定義がどこで行われているかを記録して、Emacs でそこへ簡単に飛べるようにしてくれるコマンドである。

```
1 $ cd /home/denjo/gnuplot-5.0.1/src
2 $ etags *.c *.h
```

としておくと、同ディレクトリに、`TAGS` というファイルができる。その上で Emacs 内の、適当なソースファイルを開いているバッファで、`find-tag` コマンドを実行する (`M-x find-tag` とするか、ショートカット `M-.`)。

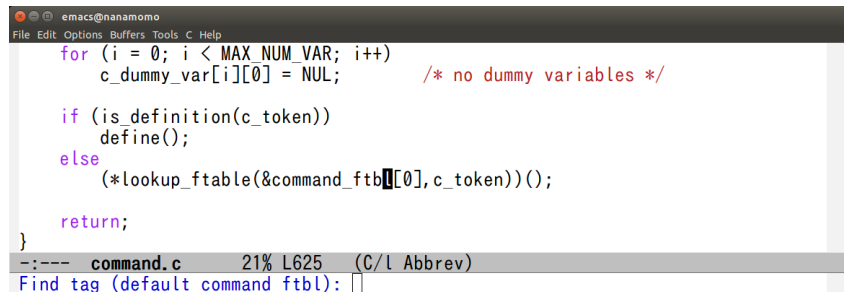
```
--:--- command.c      21% L620  (C/l
Find tag (default c_dummy_var):
```

すると、検索したい名前を聞かれるので、`command_ftbl` と入力する。

```
--:--- command.c      21% L620  (C/l Abbrev)
Find tag (default c_dummy_var): command_ftbl
```

すると Emacs が該当のファイルを開いて、その定義の付近にカーソルを移動してくれる。

なお、カーソルが `command_ftbl` という文字列の上にあるときに、実行すると、デフォルトでそれを検索してくれる。通常ソース上に、定義を探したいものが現れた場合にこの機能を使うので、多くの場合、入力の手間を省略できる。



```
emacs@nanamomo
File Edit Options Buffers Tools C Help
for (i = 0; i < MAX_NUM_VAR; i++)
  c_dummy_var[i][0] = NUL; /* no dummy variables */

if (is_definition(c_token))
  define();
else
  (*lookup_fhtable(&command_ftbl[0], c_token))();

return;
}
--:--- command.c      21% L625  (C/l Abbrev)
Find tag (default command_ftbl):
```

方法 3: Eclipse のような統合環境を使う。Emacs にどうも慣れないという人は、使ってみても良いかもしれない。¹⁵

どの方法でも、`tables.c` というファイルの中で、以下のように定義されているのが見つかる。

¹⁴ときとしてこれが `#define` で定義されたマクロで、... というケースもある

¹⁵こちらに余力があれば、適宜授業 HP 上などで補足する予定


```

1 const struct gen_ftable command_ftbl[] =
2 {
3     { "raise", raise_command },
4     { "lower", lower_command },
5     ...
6
7     { "$", datablock_command },
8     /* last key must be NULL */
9     { NULL, invalid_command }
10 }

```

3.19 command_ftbl 表の変更

明らかにこれが目当てのものでありそうなので、この最後のエントリを変更する。ここでも変更した箇所がわかるように、かつあとで簡単に戻せるように、`#if 1` で囲む。

```

1     { "$", datablock_command },
2     /* last key must be NULL */
3 #if 1
4     { NULL, default_plot_command }
5 #else
6     { NULL, invalid_command }
7 #endif

```

3.20 変更したコードをいざコンパイル

ソースコードを変更したあと、再ビルドするには、最初にビルドした手順の、`make`以降を実行すれば良い。`configure`は、コンパイルのためのオプションを変更するなどしないかぎり必要ない。

`make`を実行すると、ソースコードの変更した部分によって、再コンパイルが必要なファイルだけを再コンパイルしてくれる。さて、いざ実行すると、以下のようなエラーが出る。

```

1 $ make
2
3 ... <省略> ...
4
5 gcc -DHAVE_CONFIG_H -I. -I.. -I../term -I../term -DBINDIR=\"/home/tau/public_html/
  lecture/dive_to_oss/projects/gnuplot_install/bin\" -DX11_DRIVER_DIR=\"/home/tau/
  public_html/lecture/dive_to_oss/projects/gnuplot_install/libexec/gnuplot/5.0\" -
  DQT_DRIVER_DIR=\"/home/tau/public_html/lecture/dive_to_oss/projects/gnuplot_install/
  libexec/gnuplot/5.0\" -DGNUPLOT_SHARE_DIR=\"/home/tau/public_html/lecture/dive_to_oss/
  projects/gnuplot_install/share/gnuplot/5.0\" -DGNUPLOT_PS_DIR=\"/home/tau/public_html/
  lecture/dive_to_oss/projects/gnuplot_install/share/gnuplot/5.0/PostScript\" -
  DGNUPLOT_JS_DIR=\"/home/tau/public_html/lecture/dive_to_oss/projects/gnuplot_install/
  share/gnuplot/5.0/js\" -DGNUPLOT_LUA_DIR=\"/home/tau/public_html/lecture/dive_to_oss/
  projects/gnuplot_install/share/gnuplot/5.0/lua\" -DCONTACT=\"gnuplot-bugs@lists.
  sourceforge.net\" -DHELPPFILE=\"/home/tau/public_html/lecture/dive_to_oss/projects/
  gnuplot_install/share/gnuplot/5.0/gnuplot.gih\" -DGNUPLOT_X11=\"echo gnuplot_x11 | sed
  's,x,x,'\" -DXAPPLRESDIR=\"/etc/X11/app-defaults/\" -pthread -I/usr/include/cairo -I
  /usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/
  pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng12 -I/usr/include/pango-1.0 -
  O0 -g -MT tables.o -MD -MP -MF $depbase.Tpo -c -o tables.o tables.c &&\\
6 mv -f $depbase.Tpo $depbase.Po
7 tables.c:102:13: error: 'default_plot_command' undeclared here (not in a function)
8     { NULL, default_plot_command }

```

実行されている長いコマンドラインの中にエラーメッセージが埋没してしまっているが、

```
1 tables.c:102:13: error: 'default_plot_command' undeclared here (not in a function)
2     { NULL, default_plot_command }
3     ^
```

がエラーメッセージである。なお、コンパイル自体を Emacs の M-x compile を用いて行えば、エラー行へ飛ぶ手間が省けるのも言うまでもない。¹⁶

このエラーは、

- default_plot_command という関数は、command.c に定義されている
- 今コンパイルしているのは tables.c で、tables.c 内には、default_plot_command の定義も宣言もない、

という理由で生じている (この事の基本については、第 A.1 節参照)。

これを防ぐためには、command.ftbl の定義に先立ち、default_plot_command 関数の宣言:

```
1 void default_plot_command();
```

を書き加えれば良い。¹⁷

直接 tables.c にそれを書き足してもよいのだが、他の関数がどこで宣言されているかを突き止め、それに習うのが良い。tables.c の中の前の方を見ても、関数が宣言されている様子はない。その代わりに多くの、#include 文がある。この中の、どこに目当ての宣言がなされているかは、例えば grep で関数名を検索するなどしても良いが、ファイル名からも

```
1 #include "command.h"
```

がそれであろうと想像できる。実際に開けてみると、

```
1 ... <前略> ...
2 void refresh_request __PROTO((void));
3 void refresh_command __PROTO((void));
4 void call_command __PROTO((void));
5 void changedir_command __PROTO((void));
6 ... <後略> ...
```

のように、多数の関数が宣言されている。ここに以下の一行を書き足せば良い。

```
1 #if 1
2 void default_plot_command __PROTO((void));
3 #endif
```

こうして再び make すると今度は無事成功し、make install して実行する。

...と、なんとあっさり、所望の動作が実現できた!

3.21 この続きは...

今回の変更はこれだけで実現できてしまったが、本当にこれだけの変更で良いのかは、もう少し、色々な入力を与えて検証すべきである。特に今回行った変更は、実質的には、

```
1 plot_token = c_token++;
```

¹⁶C-x ‘でエラー行へ飛んでくれる

¹⁷意味がわからなければ第 A.1 節参照。

という行を削除したということだけだから、その後 `plot_token` という変数を使う関数がないか、そのような関数がおかしな動作をしないかは、特に注意深く見る必要がある。

`plot_token` が局所変数であれば、その関数の中だけを見れば良い。しかしどうやら、`plot_token` は大域変数なので、これがここでセットされているということは、どこか別の関数で必ずや使われている、ということだろう。なので本来は、`plot_token` という変数が、セットされたあとどこで使われて (読まれて) いるかを見極めて、適切な変更を施さなくてはならない。

一連の説明の目的は、プログラムの動作を追跡するためのイロハを説明することであり、`gnuplot` を知り尽くすことではないので、これ以上は深入りしない。

一応捕捉として、実はこの代入分を省略することによって、`replot` というコマンド (最近の `plot` コマンドをもう一度行う) の動作がおかしくなるということを述べておく。

3.22 より模範的なソースコード変更の仕方

第 3.15 節では、追加する部分を、`#if 1 ...#endif` で、削除する部分を、`#if 0 ...#endif` で囲むことを推奨した。こうしておく、少なくとも変更点をあとから `grep` で検索したりするのはやりやすくなる。

ただしこれでも変更を元に戻したり、またそれを有効にしたりするたびに、多数の箇所の 1 と 0 を書き換えなくてはいけなくなる。不具合が生じた時に、それが変更によって生じたものなのかどうかを突き止めるために、変更箇所の有効・無効を切り替えたいことはよくあり、コードを変更する場合はそれに備えた変更方法が望ましい。

これを行うために以下のようにする。

- 何か変更名称—たとえば `DEFAULT_PLOT`—を付ける。
- 追加部分を、`#if DEFAULT_PLOT ...#endif` で囲む
- 削除部分を、`#if !DEFAULT_PLOT ...#endif` (もしくは `#if DEFAULT_PLOT ...#else ...#endif` の `else` 節) で囲む
- 変更箇所を有効にしたい場合、コンパイルオプションに、`-DDEFAULT_PLOT=1` を、無効にしたい場合、`-DDEFAULT_PLOT=0` をつけてコンパイルする。これを実行するには、`-O0`、`-g` をコンパイルオプションに指定した時と同様、`configure` を実行する際に `CFLAGS` にオプションを追加する。

つまり手順としては、以下で変更箇所を有効にしてコンパイルする。

```
1 $ CFLAGS="-O0 -g -DDEFAULT_PLOT=1" ./configure --prefix=/home/denjo/gnuplot_install
2 $ make clean
3 $ make
4 $ make install
```

4 これからどこへ

最初に述べたとおりこの教科書で述べたことは、「道具の使い方」で、これをさっとなしてからが本番である。あえて形式的に書けば、チーム分けをしたあと、

本課題 4.1 (ディスカッションフェーズ): チームで変更したいオープンソース・ソフトウェア、および施したい改良・拡張について議論せよ

本課題 4.2 (ディスカッションフェーズ): 変更したいオープンソース・ソフトウェア、および施したい改

良や拡張について、ミニ発表せよ

本課題 4.3 (探求フェーズ): 対象となるオープンソースソフトウェアの実行を追跡してみよ。効率的に追跡できるように、各ソフトごとに問題があればそれを乗り越えて共有せよ

本課題 4.4 (探求フェーズ): 提案機能を実現するための背景理論などについて必要な独学をし、実現までこぎつけよ

とでもなろうか。

また、適宜、これまでにわかったことや、作業内容を記録に残すようにし、こまめに短い発表と議論を挟む。

課題をやりながら自分で学ぶ姿勢を重要視しているので、最初の時点で実現可能性があまり見えていなくてもまずは大きめの目標を設定し、後で徐々に現実的な課題に落としていくことは、気にせずにやって良い。その過程もチーム間で共有・議論することを目指して、実験時間を使う。

オープンソース・ソフトウェアと言われても何があるのかもよくわからないという人は、普段自分が何を使っているかを振り返ってみれば良い。実に様々なものを使っているはずである。

- 言語 (gcc, llvm, python, ruby, perl, etc.)
- シェル (bash, zsh, etc.)
- オフィス系 (libreoffice, gnumeric, etc.)
- お絵かき・画像 (inkscape, imagemagick)
- データベース (sqlite3, mariadb, postgresql, etc.)
- インターネット (firefox, chromium, etc.)
- OS (Linux, FreeBSD, etc.)
- ...

自分(達)が興味のわくもの、改良・拡張できたら嬉しいと思われるものを選び、まずはトリビアルな変更でいいから施してみよ。本格的な変更はそれをやりながら考えても良い。また、よい機能拡張が中々思いつかない場合、性能改善、そのための並列化、などというオプションは常に使える。大きなデータを与えたり、時間の掛かりそうな処理をやらせてみて、それを改善できないか考えてみると良い。

なお、OSの実行をデバッガで追跡するのは一苦勞である。kgdbという機能を使うとできるが、Linuxのカーネルを再コンパイルするするだけでも一晩は覚悟しておいたほうが良い。演習の時間中、ただただコンパイルを待っているだけ、とかいうことにはならないように、時間の使い方を考えること。

ビルドや追跡が難しいソフトウェアに関する tips は適宜、授業 HP などでもフォローするかもしれない。

A ビルドに関連する基本知識およびコマンド

実用的なほとんどすべてのプログラムは、複数のソースファイルからなっている。そのようなプログラムをビルドするための、定型化された手順として、多くのオープンソースソフトウェアは、Windows 用のものを除き、

```
1 ./configure --prefix ...
2 make
3 make install
```

の「3つのおまじないで」インストールできるようになっている。

この節ではこのおまじないの背後で何が起きているか、それを理解するための基本となる事項を説明する。

説明する理由は、

- 複数ファイルからなるプログラムを無事にコンパイルするときに、知っておかなくてはならない C 言語の規則がある。例えば新しい関数を追加して、それを別のファイルから参照するときに何をしなくてはいけないかを第 3.20 節で見た。
- configure はともかく、make は複数ファイルからなるプログラムをビルドするための基本的なツールであり、自分用の小さなプログラムでも、make を単独で使うことは十分にあり得る。
- いざエラーに直面すると、それぞれが何をやっているのかを理解する必要がある。

A.1 分割コンパイルの基本

C 言語で複数のファイルに別れたプログラムをビルドする場合の基本を説明する。実際には、あるファイルで定義されている関数、変数、型、etc. を、他のファイルから参照できるようにするための規則が、身に付けるべき項目である。

以降の説明に共通の事項として、プログラムが、

- abc.c
- def.c
- main.c

の 3 つからなっているものとする。main.c 内に main 関数の定義—さしあたり以下—が書かれているとする。

A.1.1 ビルドの手順

まず、色々守らなくてはならない規則を述べる前に、それらが全て問題なく満たされた場合のビルド手順を述べる。それは、各ソースファイル (.c ファイル) を `-c` オプションを付けてコンパイルし、オブジェクトファイル (.o ファイル) に変換し、それらをリンクして実行可能ファイルを得る、という手順である。今の例であれば、以下が手順となる。

```
1 $ gcc -c abc.c # abc.o ができる
2 $ gcc -c def.c # def.o ができる
3 $ gcc -c main.c # main.o ができる
4 $ gcc abc.o def.o main.o # a.out ができる
```

実行可能ファイルを作るのは最後の行で、この操作をリンクと呼ぶ。これに対し最初の 3 行をコンパイルと呼ぶこともある。ただし、コンパイルという言葉はもっと広い意味で、リンクまでも含めて用いられることもある。

もちろんこれまで学んだ通り `-O0`、`-g` オプションをつけ、さらにもう少し気の利いた名前—`my_prog` とする—で実行可能ファイルを作るには以下のようにすることになる。

```

1 $ gcc -O0 -g -c abc.c # abc.o ができる
2 $ gcc -O0 -g -c def.c # def.o ができる
3 $ gcc -O0 -g -c main.c # main.o ができる
4 $ gcc -o my_prog abc.o def.o main.o # my_prog ができる

```

以下は、ファイルにまたがって参照関係がある際に、上記がうまく行くために知っておくべき規則についての説明である。

あるファイルで定義されたもので、他のファイルから参照可能なものとしては、以下がある。

- 変数, 配列
- 関数
- 型 (typedef)
- マクロ定義 (#define)

以下で順に説明する。

A.1.2 変数, 配列

あるファイルで定義されている大域変数や配列を、他のファイルで参照する際の規則を述べる。今、abc.c に大域変数と配列

```

1 int v_abc = 10; /* 変数 */
2 int a_abc[10] = { 100 }; /* 配列 */

```

が定義されており、それを def.c 内で以下のように使用しているとする。

```

1 int def() { return v_abc + a_abc[0] + 1; }

```

def.c には上記以外の内容は一切ないものとする。

ここでそのまま第 A.1.1 節で述べたビルドの手順を実行しようとする、

```

1 $ gcc -O0 -g -c def.c

```

が以下のように失敗する。

```

1 $ gcc -c def.c
2 def.c: In function 'def':
3 def.c:1:20: error: 'v_abc' undeclared (first use in this function)
4   int def() { return v_abc + a_abc[0] + 1; }
5                       ^
6 def.c:1:20: note: each undeclared identifier is reported only once for each function it appears
7   in
8 def.c:1:28: error: 'a_abc' undeclared (first use in this function)
9   int def() { return v_abc + a_abc[0] + 1; }
                        ^

```

いずれも、def.c 内に、v_abc, a_abc の定義も、宣言もないことがエラーの原因である。同じディレクトリ中に abc.c があり、その中に v_abc や a_abc があるから、それを見してくれる、というほど、C コンパイラは気が効いていない。参照を解決するために見るのはあくまで def.c の中だけである。

ではということで、def.c 内にも、

```

1 int v_abc = 10; /* 変数 */
2 int a_abc[10] = { 100 }; /* 配列 */

```

を書くことにしたらどうなるか? 都合、def.c の内容は以下のようになる。

```

1 int v_abc = 10;      /* 変数 */
2 int a_abc[10] = { 100 }; /* 配列 */
3 int def() { return v_abc + a_abc[0] + 1; }

```

(注: abc.c 内にも同じ内容が書かれている) すると, def.c を def.o にするステップ自体は成功するが, 最後実行可能ファイルを得るときに, 次のエラーが起きる.

```

1 $ gcc -c abc.c
2 $ gcc -c def.c
3 $ gcc -c main.c
4 $ gcc abc.o def.o main.o
5 def.o(.data+0x0): multiple definition of 'v_abc'
6 abc.o(.data+0x0): first defined here
7 def.o(.data+0x20): multiple definition of 'a_abc'
8 abc.o(.data+0x20): first defined here
9 collect2: error: ld returned 1 exit status

```

見ての通り, v_abc, a_abc が複数定義されているというものである. abc.c, def.c の両者がてんでにそれらを定義しているのだから, 当然とも言える. 必要なのは, def.c が, 自分では定義せずに, abc.c 内で定義されているそれらの変数・配列を, 参照する方法である.

それが, 変数・配列の「宣言」(extern 宣言) というものである. 文法的には, 変数や配列の定義の前に, extern というキーワードをつければ良い. つまり, def.c を以下のようにすれば良い.

```

1 extern int v_abc;      /* 変数 */
2 extern int a_abc[10]; /* 配列 */
3 int def() { return v_abc + a_abc[0] + 1; }

```

これで無事に全てのステップが成功する.

```

1 $ gcc -c abc.c
2 $ gcc -c def.c
3 $ gcc -c main.c
4 $ gcc abc.o def.o main.o

```

大域変数や配列の参照に関する C 言語の規則は,

あるファイル中で大域変数や配列を参照する場合, 参照する位置よりも文面上で前の位置に, その「定義」ないし「(extern) 宣言」がなくてはならない,

というものである.

A.1.3 関数の呼び出し

今, abc.c に関数:

```

1 double abc(double x, double y) { return x * x + y * y; }

```

という関数が定義されており, それを def.c 内で以下のように呼び出しているとする.

```

1 double def() { return abc(1.2, 3.4); }

```

この関係は先の, 大域変数・配列の参照関係と似ているから, 先と同様に, def.c をコンパイルする際, abc という関数に関してのエラーが出るものと推測される.

しかしやってみると, エラーは発生しない.

```

1 $ gcc -c abc.c
2 $ gcc -c def.c
3 $ gcc -c main.c
4 $ gcc abc.o def.o main.o

```

ではこれを放っておいて良いのかというと実はそうではない、というところがややこしい。
これにまつわる C 言語の仕様は以下のようなものである。

あるファイル中で関数を呼び出すとする。呼び出す位置よりも文面上で前の位置に、その「定義」ないし「(extern) 宣言」があるかないかで以下のように定める。

場合 1 ある場合: その関数は、その定義ないし宣言された型の引数および戻り値を持つものと仮定される。

場合 2 ない場合: その関数は `int` を返すものと仮定される。また、引数は、その呼び出しにおいて渡された型の引数を持つものと仮定される。

今の例に則して言い換えれば、`def.c` で

```
1 double def() { return abc(1.2, 3.4); }
```

`abc` という関数を呼び出しているが、それより前に `abc` の定義も宣言も見当たらないので、`abc` は、

- `double` 型の引数二つを受け取り、
- `int` 型の戻り値を返す

ものと仮定される、ということである。

実際の `abc` 関数は、`double` 型を返すので、この型の不一致は、「戻り値が正しく受け渡されない」という問題をもたらす。¹⁸

コンパイルが成功してしまうことがかえって問題を複雑にしているが、「正しい」解決方法は変数の場合と同じで、関数の宣言を、使用の前に書いてやることである。今の場合、`def.c` に、

```
1 double abc(double, double);
2 double def() { return abc(1.2, 3.4); }
```

のように、`abc` 関数の引数と戻り値を書いてやる。

なお、関数を「呼び出す」のではなく、あたかも変数のように参照する場合は、変数と同様、宣言がなければ直ちにエラーになる。例えば以下はエラーになる。

```
1 int def() { printf("%p\n", abc); }
```

これの対処の仕方も同様で、関数 `abc` の宣言を書いてやれば良い。

```
1 double abc(double, double);
2 int def() { printf("%p\n", abc); }
```

A.1.4 ヘッドファイル

結局、変数の場合でも関数の場合でも、使う前に「宣言」を書いてやるというのが解決策である。それをきちんとやり遂げれば分割コンパイルは成功する。しかし、ある変数や関数を使う全てのファイルに、該当する宣言を書き加える必要があり、面倒だけでなく、

- 後に引数や戻り値の型が変更された場合、すべての宣言を変更する必要がある、
- ひとつでもそれを怠ると再び、型の不一致という危険性が生ずる

という問題がある。要するに、同じことを何度も書くのは面倒であるという以外に、不一致による間違いが生ずる元である。

そこで、あるファイルで定義された変数や関数で、他のファイルから参照されるものの宣言をまとめたファイル—ヘッドファイル—を別に作り、各ファイルからそれを `#include` によって取り込む、という方法が慣例として用いられる。ヘッドファイルは通常、`.h` という拡張子を持っている。今の状況であれば、

¹⁸実際のところ、何が返ってくるのか、というのは C 言語の仕組みの詳細を説明しないと、説明しづらい。

- `abc.c` で定義され、他のファイルから使用され得る変数や関数の宣言を、`abc.h` にまとめる
- それらの変数や関数を一つでも使うプログラムは、

```
1 #include "abc.h"
```

という一行を入れる

- `abc.c` 自身にも、

```
1 #include "abc.h"
```

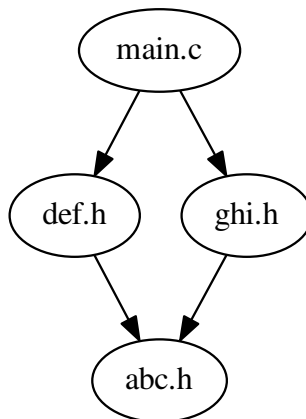
を入れる。

`#include` 句自身の動作は単純で、単に、指定されたファイルの中身をそこに展開する、というだけのものである。中身はなんでもよい。が、C プログラムにおいてはもっぱら、変数や関数の宣言を集めたヘッダファイルを、複数の C ソースから取り込むために用いられる。

なお、ヘッダファイルには先頭に、

```
1 #pragma once
```

の一言を入れるのが通常である。これは何かというと、通常通り `#include` を処理していくと同じファイルを 2 度取り込む羽目になってしまう時、2 度目以降の取り込みを抑止するためにある。典型的には、`include` で必要とされる関係が下図のように、合流してしまう時に必要になる。図で、 $A \rightarrow B$ は、ファイル A に、`#include "B"` が書かれている、ということを示している。この場合であれば、`abc.h` に、`#pragma once` を書いておくことが多くの場合必須になる。



A.1.5 型

あるファイルで、`typedef` を用いて定義した型を、他のファイルで参照する場合について。例えば、`abc.c` に、

```
1 typedef int my_int;
2 typedef struct { double x; double y } point2d;
```

のように型が定義され、それを、`def.c` で

```

1 int def() {
2     point2d p = { 3.3, 4.4 };
3     my_int x = p.x + p.y;
4 }

```

のように使用する場合. 今度は想像通り, `def.c` をコンパイルする際にエラーが出る. 対処方法も想像通りで, 型の定義を `abc.h` に書き, `def.c` (および `abc.c`) からそれらを `#include` することである.

- `abc.h`

```

1 typedef int my_int;
2 typedef struct { double x; double y } point2d;

```

- `def.c`

```

1 #include "abc.h"
2 int def() {
3     point2d p = { 3.3, 4.4 };
4     my_int x = p.x + p.y;
5 }

```

A.1.6 マクロ定義

`#define` で定義されているシンボル (マクロ) も全く同様. ヘッダファイルに書き, それを使いたいファイルはそれを `#include` すればよい.

A.2 よく使われるコンパイラのオプション

-O0, -O1, -O2, -O3 最適化レベルを指定する. デバッガで追跡する場合, なるべく `-O0` をつける.

-g, -gdwarf-4 デバッグシンボルをつけてコンパイルする. 通常は `-g` を指定するが, `-gdwarf-4` をサポートしている場合, こちらをつけると, `gdb` 内の `macro expand` というコマンドで, マクロを展開することができる.

-I 変数=値 ソースファイル内で `#define 変数 値` としたのと同じ効果をもたらす. 主に, 条件付きコンパイルの (この定義によって, ソースコードのある部分を含めたり含めなかったりする) ために用いる.

-I ディレクトリ `#include` で指定したファイル名を探索する起点を追加指定する. 例えば, `-I/home/denjo/include` とすると,

```

1 #include <abc.h>

```

は, `/home/denjo/include/abc.h` (があれば) を発見してくれるようになる. なお,

```

1 #include "abc.h"

```

のように, 2 重クオートで囲った場合, `-I` で指定された場所の他に, ソースファイルと同じディレクトリも検索してくれるようになる.

-l ライブラリ名 リンク時のオプション. リンク時に, 指定されたライブラリ名に対応するファイルを, リンクすることを指定する. 例えば, `-lfoo` と指定すると, `libfoo.so` または `libfoo.a` (先に見つかった方) がリンクされる. 「リンクされる」とは, そのファイルの中にある関数や変数などが使えるようになるということである.

-L ディレクトリ リンク時のオプション。-l と共に用いられる。リンク時に、-l で指定されたライブラリ名に対応するファイルを、探索するディレクトリを追加指定する。例えば、`-L/home/denjo/lib -lfoo` とすると、リンク時 `/home/denjo/lib/libfoo.so`, `/home/denjo/lib/libfoo.a` というファイルを (あれば) 発見してくれるようになる。

-Wl,-R ディレクトリ リンク時のオプション。-l と共に用いられる。プログラムが起動された時、リンクされた動的リンクライブラリ (`.so` ファイル) を探索するディレクトリを追加指定する。ほとんどの場合、`-L` で指定したディレクトリを、重ねて指定する。例えば、`-L/home/denjo/lib -Wl,-R/home/denjo/lib -lfoo` とする。リンク時に動的リンクライブラリ `/home/denjo/lib/libfoo.so` が見つかって、使われたとする。実際にこの動的リンクライブラリがリンクされるのはプログラムが開始された時で、`-Wl,-R/home/denjo/lib` はそれを `/home/denjo/lib/libfoo.so` から探すという意味である。`-L` で指定したディレクトリを、起動時にも自動的に探索してくれればいいのと思うのだが、そうはなっていない、ということである。

A.3 make

A.3.1 make の基本

多数のファイルを分割コンパイルするようになると、ソースファイルを修正後にもう一度コンパイルするだけでも大変な作業になる。また、コンパイラに渡すオプションを変えてコンパイルし直すのも大変である。

`make` は、そのようなビルド作業を自動化するために作られたツールである。ここではその基本について説明する。引き続き、ソースが `abc.c`, `def.c`, `main.c` の 3 つからなり、以下の手順で構築できる場合を考える。

```
1 $ gcc -c abc.c      # abc.o ができる
2 $ gcc -c def.c      # def.o ができる
3 $ gcc -c main.c     # main.o ができる
4 $ gcc abc.o def.o main.o # a.out ができる
```

`make` コマンドは無引数で実行されると、そのディレクトリで、`Makefile` という名前のファイル¹⁹を探し、そこに書かれた指示に従ってコマンドを実行する。

ではその「指示」をどう書くか。上記を達成するための、最低限の `Makefile` は以下くらい単純である。

```
1 a.out : abc.o def.o main.o
2      gcc abc.o def.o main.o
```

この二行を書くだけで、以下が実行される。

```
1 $ make
2 cc      -c -o abc.o abc.c
3 cc      -c -o def.o def.c
4 cc      -c -o main.o main.c
5 gcc abc.o def.o main.o
```

注目すべきことは、

- 最後に実行されている、`gcc abc.o def.o main.o` は、`Makefile` にかかっている事そのままである
- 一方それ以前に実行されている `cc -c ...` たちは `Makefile` にかかれておらず、これは `make` がもとと知っているということである。

より正確に、`make` は `Makefile` をどう読み、もともと何を知っているが故に、上の 3 行を実行したのか?

まず第一点目: `Makefile` の

```
1 a.out : abc.o def.o main.o
2      gcc abc.o def.o main.o
```

¹⁹`makefile` (先頭が小文字) など、他にも探されるファイル名はあるのだが、伝統的にほとんどの場合、`Makefile` が用いられる

の1行目

```
1 a.out : abc.o def.o main.o
```

において、「a.out を作るためには、abc.o def.o main.o が必要である」ことが指示されている。これを「依存関係」という。言い換えれば、a.out は、abc.o def.o main.o に依存している。そこで make はそれら3つのファイルもつくろうとするのだが、xxx.o というファイルを作るのに、xxx.c というファイルがあれば、それを元に、

```
1 cc -c -o xxx.o xxx.c
```

というコマンドでできることを make は言われなくても知っている。

A.3.2 make のカスタマイズ

上記で実行されるコマンドラインは、もともと期待していたものと微妙に違い、コマンド名が gcc ではなく cc である。また、我々が常に付けたがる、-O0, -g も勝手には付いてくれない。このへんの動作は Makefile 内で特定の変数を定義すれば制御できる。具体的には、

- CC : C コンパイラ名
- CFLAGS : .c を .o にコンパイルする際のオプション
- LDFLAGS : リンク時のオプション
- LIBS : リンク時に指定するライブラリ

などを定義することでコマンドラインを制御できる。例えば Makefile を以下のようにすると、

```
1 CC:=gcc
2 CFLAGS:=-O0 -g -Wall -Wextra
3 LDFLAGS:=-g
4 LIBS:=-lm
5
6 a.out : abc.o def.o main.o
7         gcc $(LDFLAGS) abc.o def.o main.o $(LIBS)
```

以下の動作をする。事前に、a.out と .o ファイルたちを消去しておくこと。理由は後で述べる。

```
1 $ rm a.out *.o
2 $ make
3 gcc -O0 -g -Wall -Wextra -c -o abc.o abc.c
4 gcc -O0 -g -Wall -Wextra -c -o def.o def.c
5 gcc -O0 -g -Wall -Wextra -c -o main.o main.c
6 gcc -g abc.o def.o main.o -lm
```

A.3.3 自分で定義した変数の利用

Makefile 内には自分で自由に変数を定義することもできる。通常のプログラムであっても、Makefile であっても、同じことを繰り返し書く代わりに変数を用いよ、という原則は変わらない。我々の Makefile で冗長な点は、abc.o def.o main.o という文字列が2度現れていることである。もし今後ファイルが増えていったら、毎回2箇所を更新しないといけなくなる。そうする代わりに以下のように変数を用いることが推奨される。

```
1 CC:=gcc
2 CFLAGS:=-O0 -g -Wall -Wextra
3 LDFLAGS:=-O0 -g
4 LIBS:=-lm
```

```

5
6 OBJS:=abc.o def.o main.o
7
8 a.out : $(OBJS)
9      gcc $(LDFLAGS) $(OBJS) $(LIBS)

```

A.3.4 最小限の再コンパイル

一旦 make を実行したあとで、一つのファイル—例えば main.c—を修正してから、もう一度 make を実行すると興味深いことが起きる。

```

1 $ make
2 gcc -O0 -g -Wall -Wextra -c -o abc.o abc.c
3 gcc -O0 -g -Wall -Wextra -c -o def.o def.c
4 gcc -O0 -g -Wall -Wextra -c -o main.o main.c
5 gcc -O0 -g abc.o def.o main.o -lm
6 $ emacs main.c # main.c を修正
7 $ make        # またmake
8 gcc -O0 -g -Wall -Wextra -c -o main.o main.c
9 gcc -O0 -g abc.o def.o main.o -lm

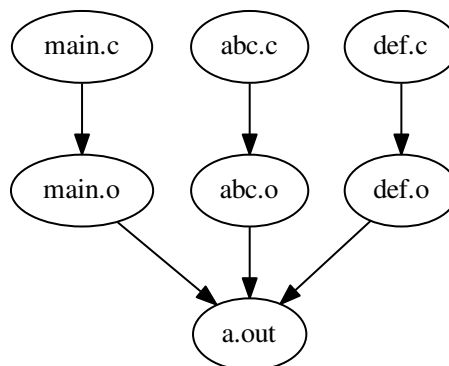
```

make は、main.c は再びコンパイルし、a.out は再リンクするものの、残りの abc.c、def.c は再コンパイルしない。

make はそれを、依存関係を見ることで判断する。今、指定されている依存関係は、

- a.out が、abc.o、def.o、main.o に依存すること (明示的に書かれている)
- abc.o が abc.c に依存 (make がもともと知っている)
- def.o が def.c に依存 (make がもともと知っている)
- main.o が main.c に依存 (make がもともと知っている)

で、図に書けば以下の通り。



make は、この図で、変更された main.c および、その下流 (矢印をたどった先) にあるファイルだけを再び、構築しようとする。

A.3.5 一般の Makefile

Makefile には一般にいくつもの依存関係やコマンドを書くことができる。つまり、

```
1 a.out : $(OBJS)
2     gcc $(LDFLAGS) $(OBJS) $(LIBS)
```

のような行をいくつも書くことができる。一般の書式は、

```
1 T : P1 P2 ...
2     コマンド
```

という形になる。ここで、 T をターゲット、 P_i を事前要件 (prerequisite) と呼ぶ。

make コマンドを実行した時に、make がどのターゲットをつくろうとするかは、コマンドライン引数で指定できる。例えば、

```
1 make xyz
```

は、xyz という名前のターゲットを Makefile 中から探し出し、そのファイルをビルドしようとする。このようにして、一つの Makefile で色々なターゲットをビルドすることができる。

なお、コマンドラインでターゲットが指定されなければ、Makefile 中に一番最初に現れたターゲットになる。

A.3.6 clean

Makefile 中のターゲットは、多くの場合作りたいファイルの名前である。が、実際にはターゲットはファイル名とは無関係な名前でもよい。これによって、実際のビルドとは無関係だが、便利でよく使うコマンドを、Makefile 中に書いておく、ということがよく行われる。実は、これまでに出てきた、make install, make clean の “install”, “clean” もターゲット名で、それらは実際に install や、clean という名前のファイルを作るわけではない。

clean は、ビルドの結果で来たファイルを消去するためのターゲットとしてしばしば用いられる。今の我々の例であれば、以下のように書いておけばそれが行える。

```
1 CC:=gcc
2 CFLAGS:=-O0 -g -Wall -Wextra
3 LDFLAGS:=-O0 -g
4 LIBS:=-lm
5
6 OBJS:=abc.o def.o main.o
7
8 a.out : $(OBJS)
9     gcc $(LDLAGS) $(OBJS) $(LIBS)
10 clean :
11     rm -f $(OBJS) a.out
```

A.3.7 コマンドラインを隠す人

Make の特徴の一つに、実行するコマンドを自動的に表示してくれる、ということがある。これは大いに重宝すべき機能で、オープンソース・ソフトウェアをビルドしている最中にコンパイルエラーが出た場合も、make が実行しているコマンドラインを見て、具体的にどんなコマンドがエラーになったのかを突き止める手がかりが得られる。

この方に大事な機能なのだが、一方で、長ったらしいコマンドラインを表示したくないと思うのか、「見た目の美しさ」にこだわって、実行されているコマンドラインを「隠そう」とする人がいる。そのような人はたいてい、本物のコマンドラインの代わりに、簡素化されたメッセージだけを表示しようとする。make で実行されるコマンドラインを「隠す」には、コマンドの前に、@を置けばよい。例えば、我々の Makefile を以下のように修正すると、

```

1 a.out : $(OBJS)
2     @echo "LD a.out"
3     @gcc $(LDLFLAGS) $(OBJS) $(LIBS)

```

例えば以下のような出力になる。

```

1 $ make
2 gcc -O0 -g -Wall -Wextra -c -o abc.o abc.c
3 gcc -O0 -g -Wall -Wextra -c -o def.o def.c
4 gcc -O0 -g -Wall -Wextra -c -o main.o main.c
5 LD a.out

```

もちろん「隠したがる派」は、こんなことでは満足できず、なんとかして、以下のような表示になるまで頑張ることだろう。

```

1 $ make
2 CC abc.c
3 CC def.c
4 CC main.c
5 LD a.out

```

だがそんなことを苦労してやっても誰のためにもならない。

ここでそれを説明した理由は、そのような邪悪な Makefile が世の中に存在することを事前に教えておくためと、それを「剥がす」には、Makefile 中でコマンドの前におかれた@記号を除去してまわればいいのかということ伝えるためである。

A.3.8 make と configure

make は元来この程度の記述を自分で行って、複数ファイルから成るプログラムのビルドを自動化するための、手軽なツールであった。

しかし現在、多くのオープンソースソフトウェアでは、Makefile を手書きで書くことはなく、configure によって生成される。そしてそれは、とても人間が見るに耐えない汚く、大きな Makefile になっている。ましてやそうして生成された Makefile を直接人間がいじるということは、想定外である。

もしいじるのであれば、configure が Makefile を生成する際に、用いている「元ファイル」をいじることになるが、これは Makefile.in という。つまり、多くのオープンソースソフトウェアのアーカイブを解凍すると、Makefile は含まれていないが、代わりに Makefile.in が含まれている。

ではこの Makefile.in はシンプルで修正の聞かぬファイルなのかというと、そうではなく、これもまた自動生成されている。その自動生成のもととなっているのは Makefile.am というファイル。この Makefile.am → Makefile.in という生成を行うのは、automake というツールである。

したがって Makefile に恒久的な修正を加える正統的な方法は Makefile.am を修正し、automake を使って Makefile.in を生成、さらにそこから ./configure で Makefile を生成する、という方法なのだが、この automake というツールが曲者で、日々バージョンが変わり、バージョンが変わるたびに、非互換性が生じる。したがって、どこからダウンロードしてきたソースの Makefile.am から、無事 Makefile.in が作れるという保証はどこにもない、という八方ふさがりな状態になっている。結局、Makefile に気軽に修正を加える方法はない、ということである。

A.3.9 それでも Makefile に修正を加えなくてはならない場合

それでも Makefile に修正を加えなくてはならない場合が存在する。

1. 実行中のコマンドを隠す邪悪な Makefile に出会ってしまった場合
2. プログラムに新しいソースファイルを加えたい場合

前者とどう渡り合うかは第 A.3.7 節で説明した。

後者について。

方法 1: 本来のやり方. それらしい場所にある Makefile.am を開くと, 大概, ソースファイル名が列挙されているので, それを変更し, automake を実行する. 「それらしい場所」というのは, 例えば gnuplot であれば, src/Makefile.am のことで, それを開くと, そのディレクトリにある.c や.h などのファイルを列挙した変数定義らしきものが見つかる.

```
1 gnuplot_SOURCES = alloc.c alloc.h axis.c axis.h breaders.c breaders.h bitmap.h \
2   ...
```

ただし, automake がうまく行かない可能性が高いのと, それに対処するには本腰を入れて automake と向き合わなくてはならないので, ここでは深入りしない. automake, autotools に習熟したい, という意欲のある人は試してみると良い.

方法 2: 付け焼刃的, かつ, 変更を恒久的にするための方法としては不適切だが, ad-hoc に Makefile.in をいじって我慢する. Makefile.in 中で, それらしい変数を見つけて修正する. 今の場合であれば, Makefile.am 中で見つけた gnuplot_SOURCES という変数が, やはり Makefile.in 中にも存在するので, それを変更する.

A.4 configure

A.4.1 configure とは

ほとんどのオープンソースソフトウェアには configure というファイル (実体はシェルスクリプト) が同梱されており, make に先立って実行される慣例になっている. configure の主要な役割は,

1. ビルドに必要なソフトウェア (ヘッダファイル, ライブラリ, コマンドなど) がビルド環境に揃っているかをチェックする.
2. 揃っていたら, make が用いる Makefile を生成する.
3. ユーザが指定したビルド方法 (コンパイル時やリンク時のオプション) やインストール先を, オプションとして受け付け, 生成される Makefile に反映させる

ということである.

configure という汎用的なコマンドがあるわけではなく, それぞれのソフトウェアに, そのソフトウェア用の configure が同梱されている. しかし configure 自身, configure.ac というファイルから自動生成されているので, 多くの configure が共通の慣例を守っている. 以下は, ほとんどの configure に共通のオプションや環境変数である.

--help その configure が受け付けるオプションや環境変数を表示する. どんなオプションがあるかを見るには, これを使うのが一番信頼できる. 以下は, ほとんどの configure が受け付けるオプションや環境変数.

--prefix=ディレクトリ インストールするディレクトリの指定

(環境変数) CFLAGS="..." コンパイル時に C コンパイラに渡すオプション. 典型的には, -O0, -g, -D..., -I... など.

(環境変数) LDFLAGS="..." リンク時に C コンパイラ (リンカ) に渡すオプション. 典型的には, -L および -Wl, -R.

(環境変数) LIBS="..." リンクするライブラリを指定する. 典型的には -l ライブラリ名.

A.4.2 configure でエラーが出た時の対処

移植性の高い、よく出来たオープンソースのソフトウェアをビルドする際に起きるエラーの大半は configure 時に起きる。configure は、そもそもそのソフトウェアをビルドするための要件を、ビルド環境が満たしているか—必要なヘッダファイル、ライブラリ、コマンドが揃っているか、など—を検査するのが主目的で、それらが無い環境でエラーになるのは、不可避である。

もちろんパッケージ管理ツールのように、必要なパッケージも自動的にインストールする、という方針がないわけではないが、ソースからビルドする場合、そこまで勝手にやることはまず行われない。

configure でエラーが出た場合、エラーメッセージからエラーの原因を特定 (想像?) し、必要なパッケージ (多くの場合、ライブラリ) をインストールすることで対処する。

また、エラーが出なくても、よく見ると警告が出ているということもある。警告が出たままコンパイルすると一部の機能が使えない状態で、プログラムがビルド、インストールされる、ということもあり得るので、configure の出力した警告は見ておく習慣をつけると良い。

例えば以下は gnuplot の configure を、とあるシステムで実行した時の出力 (一部) である。

```

1  $ CFLAGS="-O0 -g" ./configure --prefix=/home/denjo/gnuplot_install
2
3  ... <中略> ...
4
5  checking for LIBCERF... configure: WARNING:
6  Package requirements (libcerf) were not met:
7
8  No package 'libcerf' found
9
10 ... <中略> ...
11
12 configure: WARNING: GNU readline not found - falling back to builtin readline
13
14 ... <中略> ...
15
16 configure: WARNING: libgd not found or too old, version >= 2.0 is required
17
18 ... <中略> ...
19
20 checking for LUA... no
21 configure: WARNING: Could not find support for lua using pkg-config.
22
23 ... <中略> ...
24
25 checking for wx-config... no
26 configure: WARNING: wxWidgets can't be found. You can try --with-wx=DIR to give the right path
    to wx-config. The wxWidgets terminal will not be compiled.
27
28 ... <中略> ...
29
30 checking for QT... configure: WARNING:
31 Package requirements (Qt5Core Qt5Gui Qt5Network Qt5Svg Qt5PrintSupport) were not met:
32
33 ... <中略> ...
34
35 configure: WARNING: The Qt terminal will not be compiled.
36 checking that generated files are newer than configure... done
37 configure: creating ./config.status
38 config.status: creating Makefile
39
40 ** Configuration summary for gnuplot 5.0.1:

```

```

41 ... <後略> ...
42

```

よく見ると色々な警告が出ている。本来であればこれら一つ一つに対処していくべきだが、その対処の仕方については、完全な処方箋はない。configure 自身はどんな環境でも走ることを想定しているので、特定のシステムに関する知識 (例えば Ubuntu ではこのパッケージをインストールすれば良いとか) は持ち合わせていない。

具体的な対処方法はある程度、勘に頼って行う。例えば、

```

1 configure: WARNING: GNU readline not found - falling back to builtin readline

```

の行については、readline と名のつくパッケージを探してみて、ライブラリらしきものをに入れてみるというのが一つの手である。

```

1 $ apt-cache search readline

```

とすると多数のパッケージが表示されるが、

```

1 libreadline-dev - GNU readline and history libraries, development files

```

がそれっぽい。もう少し確度の高い情報を得たければ、config.log というファイルの中身を覗くと良い。ここには configure がどんなコマンドを実行して、GNU readline not found という結論に至ったのかが書いてある。大概の場合、GNU readline を使う小さなプログラムのコンパイル (リンク) が失敗したことを持って、無いと判断している。失敗したコマンドやその際のメッセージがわかれば、結局何を直せばこの警告が消えるのかが判断できる。

そこで、config.log を開いて、その中から、WARNING: GNU readline not found という文字列を探すと、

```

1 configure:10370: result: no
2 configure:10375: WARNING: GNU readline not found - falling back to builtin readline

```

という行が見つかり、その少し上に、失敗したプログラムやその際のエラーメッセージが書いてある。今の場合、

```

1 ... <前略> ...
2 | #ifdef __cplusplus
3 | extern "C"
4 | #endif
5 | char remove_history ();
6 | int
7 | main ()
8 | {
9 |     return remove_history ();
10 | ;
11 |     return 0;
12 | }

```

というプログラムのコンパイルにしっばしており、更にその上には

```

1 configure:10336: checking for remove_history in -lhistory
2 configure:10361: gcc -o conftest -O0 -g    conftest.c -lhistory -lncurses -ldl -lm  >&5
3 /usr/bin/ld: cannot find -lhistory
4 collect2: error: ld returned 1 exit status

```

というエラーメッセージが書かれている。ポイントは、-lhistory というライブラリのリンクに失敗しているということである。更にその上には、

```

1 configure:10269: gcc -o conftest -O0 -g    conftest.c -lreadline -lncurses -ldl -lm  >&5
2 /usr/bin/ld: cannot find -lreadline
3 collect2: error: ld returned 1 exit status

```

というものも見える。より明確な目標として、`-lreadline` が成功するようになることということを意識して、パッケージを探す。

少し別のパターンとして、

```
1 checking for QT... configure: WARNING:
2 Package requirements (Qt5Core Qt5Gui Qt5Network Qt5Svg Qt5PrintSupport) were not met:
```

の対処方法を追求する。このエラーを出すに至った経緯は、やはり `config.log` 中に書かれており、

```
1 configure:14248: checking for QT
2 configure:14256: $PKG_CONFIG --exists --print-errors "QtCore >= 4.5 QtGui >= 4.5 QtNetwork >=
   4.5 QtSvg >= 4.5"
```

のように、`$PKG_CONFIG` というコマンドに失敗したことが原因のようである。`$PKG_CONFIG` はシェルスクリプトの中の変数であり、その定義は `config.log` の中を検索すると、

```
1 PKG_CONFIG='/usr/bin/pkg-config'
```

という行が見つかることから、おそらく `/usr/bin/pkg-config` なのであろうと想像がつく。そこで上記の通りのコマンドを酒盗で実行してみても、似たようなエラーが表示されることから、

```
1 $ /usr/bin/pkg-config --exists --print-errors Qt5Core
2 Package Qt5Core was not found in the pkg-config search path.
3 Perhaps you should add the directory containing 'Qt5Core.pc'
4 to the PKG_CONFIG_PATH environment variable
5 No package 'Qt5Core' found
```

ここでは、`Qt5Core` というパッケージがなんとか見つかるように、パッケージを入れることになる。何と言うパッケージを入れればよいかは、

```
1 $ apt-cache search Qt5Core
```

で以下のような表示がなされることから、

```
1 $ apt-cache search Qt5Corelibqt5core5a - Qt 5 core module
```

おそらく `libqt5core5a` なのだろうと想像がつく。

以下のような感じで、勘も使いながら必要なパッケージを入れていくのが、`configure` のエラーや警告に対する実際的な対処法である。

B ソースツリーを探索するツールあれこれ

本演習では、ある目的を達成するためにソースコード中でどこをいじればいいのかの答えにたどり着くために、デバッガを使いこなせるようになることを強調している。

一方で、デバッガを用いる方法は、目的の箇所にたどり着くまでに、追跡しなくてはならないステップが多いと、目的地へたどり着くまでに時間がかかるなどの欠点もある。

実際に多数のソースツリーを探索するときは、デバッガで精密に実行を追跡する以外に、より原始的な方法として、ソースコードを適当なキーワードで検索する、ということもよく行う。その方が多数のファイルを相手に一網打尽に検索をかけることができるので、効率が良いことも多い。TPO に応じて使い分けられるようになると良い。

例えて言うならば、ここで紹介するツールは、森全体を一望するためのツール、デバッガは一本一本の木を精査するためのツールである。

B.1 grep

言わずと知れたコマンドだと思うが、いくつか、ソースコードを検索するために便利なオプションを紹介しておく。もちろんこんなものは、man を見れば載っている。以下を見なくても、自分で grep を使っていて「ひょっとしてこんな機能はあるんじゃないか?」と思って man を見てたどり着けるようになるのが最高である。

`-r` : あるディレクトリ以下すべてのファイルに同じ検索をかける。例:

```
$ grep -r g_signal_connect .
```

`-I` : 検索をする際、バイナリファイル (テキストではないと思われるもの) を無視する。

```
$ grep -I -r g_signal_connect .
```

これはソースコードを検索するときに特に重宝する。ソースコード中に現れる多くの文字列 (関数名など) は、それをコンパイルしてできたオブジェクトファイル (.o) や実行可能ファイルからも見つかってしまうので、それを排除できる。さらに、それらを排除せずに検索すると、検索に無駄に時間がかかるようになる。

`"..."` : これは grep の話ではないが、スペースを含んだ文字列を検索したければ、クオートすればよい。

```
$ grep -I -r "x = y" .
```

正規表現: 基本中の基本だが、grep は、文字列そのものを見つけるだけでなく、パターン (正規表現) にマッチした行を見つけることができる。

```
$ grep -I -r 'x *= *y' .
```

は、`x = y`, `x=y`, `x= y` などにマッチする。

M-x grep : これは grep の機能そのものではないが、Emacs の **M-x compile** と似た機能で、Emacs 内で grep をかけられ、マッチしたファイルと行へ **C-x '** (バッククオート) で飛べる。多数のファイルに対して大量に引っかけた結果を効率的にチェックしたい時に重宝する。

B.2 GNU global (gtags, htags, global)

ソースコードを探索しているとある関数が定義されている場所を突き止める、逆にある関数が使われている場所をすべて突き止める、ということをししばしやりたくなる。そのようなことを手助けするツールは一般にクロスリファレンスツールと呼ばれる。一度ソースコードをスキャンして、何がどの場所で定義され (参照されて) いるかのデータベースを作るコマンド、そのデータベースを検索するコマンドからなるのが普通である。また、それらのコマンドを Emacs, vi などのエディタから呼び出せるようにしたり、ウェブブラウザで閲覧できる形式にしたりなど、色々な形式がある。ここでは、GNU global というツールを紹介する。

B.2.1 インストール

gtags, htags, global というコマンドが使えればすでにインストールされている。使えなければ、Ubuntu であれば global というパッケージをインストールすれば良い。

```
$ sudo apt-get install global
```

以下に書かれていない使い方は、man global や、<https://www.gnu.org/software/global/manual/global.html>などを参照。

B.2.2 htags : ブラウザでソースを閲覧

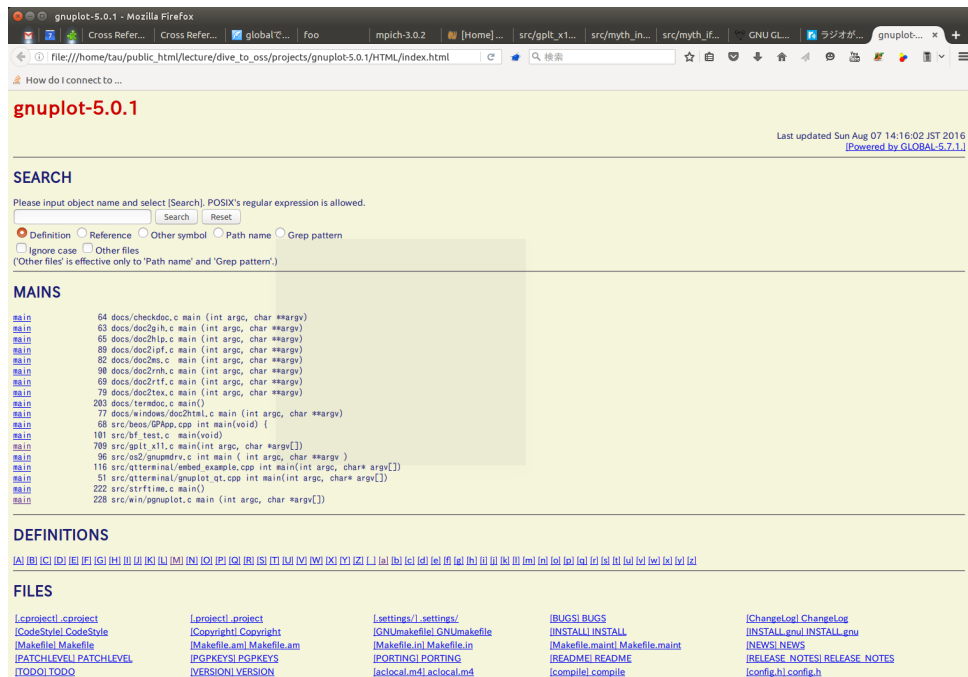
htags は、ソースツリーをブラウザでソースを閲覧できるようにする強力なツール。ブラウザ上で、ある関数名や型名などが定義、参照されている場所へ簡単に飛ぶことができる。

ソースを含むディレクトリで htags を実行すると、HTML というフォルダの中に、多数の html ファイルが生成される。あとは、HTML/index.html をブラウザで開けばよい。

無引数で起動するだけで良いが、`--suggest` というオプションを指定すると、便利なオプションを色々と指定してくれるのでつけておくと良い。

```
1 $ cd <どこか>/gnuplot-5.0.1
2 $ htags --suggest
3 ...
4 $ firefox HTML/index.html # HTML/index.html をブラウザで開く
```

すると、以下のようなページが表示される。



ソース内で見つかった main 関数が並んでおり、それらをクリックするとその定義が表示される。以下は、

- 表示されたソースコード中で、あるシンボル (関数名や型名など) が参照さ (使わ) れている場所で、そのシンボル名をクリックすると、その定義へ飛ぶ。
- あるシンボル (関数名や型名など) が定義されている場所で、そのシンボル名をクリックすると、それが参照さ (使わ) れている場所のリストが表示される。

という動作をする。

B.2.3 gtags と global

gtags が定義や参照のデータベースを構築するコマンド。global がそれを検索するコマンドラインツールである。htags よりも低レイヤに属するツールで、実は htags `--suggest` を実行した時点で、gtags がこっそり実行されている。ソースをブラウズするツールとしての使い勝手という意味では、htags の方が圧倒的に優れているので、あえて global をコマンドラインで使う意味は少ないのだが、Emacs や vi など他のエディタと統合して使われるため、一度コマンドラインでも使っておくと良い。

1. すべてのソースコードを含むディレクトリに移動して gtags を実行 (ソースコードを変更しない限り一度だけやればよい).
2. global で検索

という手順. 例:

1. データベースを作る. gtags を無引数で起動するだけ.

```

1 $ cd <どこか>/gnuplot-5.0.1
2 $ gtags
3 $ ls
4 BUGS          INSTALL        README        config.status* missing*
5 ChangeLog     INSTALL.gnu    RELEASE_NOTES configure*     mkinstalldirs*
6 CodeStyle     Makefile       TODO          configure.in*  pm3d/
7 Copyright     Makefile.am    VERSION       configure.vms* share/
8 FAQ.pdf       Makefile.in    aclocal.m4    demo/         src/
9 GNUMakefile   Makefile.maint compile        depcomp*      stamp-h
10 GPATH         NEWS          config/       docs/         stamp-h1
11 GRTAGS        PATCHLEVEL    config.h      install-sh*   term/
12 GSYMS         PGPKEYS       config.hin    m4/           tutorial/
13 GTAGS         PORTING       config.log    man/          win/

```

すると, GPATH, GRTAGS, GSYMS, GTAGS というファイルができています (htags を以前に実行していれば, 既に出てくるかも知れない). リセットしたければこれらを消せばいいという以外に, あまり中身について気にする必要はない.

2. 定義の場所を調べる. GPATH, GRTAGS, GSYMS, GTAGS が存在するディレクトリで, global に調べたいシンボル名を与えるだけ. -x オプションを与えると, ファイル名だけでなく, 行番号なども表示してくれる.

```

1 $ global plot_command
2 src/command.c
3 $ global -x plot_command
4 plot_command      1591 src/command.c      plot_command()

```

3. 参照の場所を調べる. -r オプションを与えると与えたシンボルが参照されている (つかわれている場所) を知ることができる. -x で行番号などが表示される点も同じ.

```

1 $ global -r plot_command
2 src/command.h
3 src/tables.c
4 $ global -xr plot_command
5 plot_command      166 src/command.h      void plot_command __PROTO((void));
6 plot_command      74 src/tables.c        { "p$plot", plot_command },

```

B.2.4 Emacs と gtags

Emacs 内で global を実行して, 該当するファイルを自動的に開いてくれるツールがある.

基本となる手順 global をインストールすると, Emacs で gtags-mode というコマンドが使えるようになる.

1. 事前に gtags を実行しておく
2. C ソースファイルを開いた状態で gtags-mode コマンドを実行 (M-x gtags-mode) して, gtags-mode を有効にする

```

emacs@nanamomo
File Edit Options Buffers Tools Minibuf Help
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "stdfn.h"

/* prototypes from "alloc.c" */
generic *gp_alloc __PROTO((size_t size, const char *message));
generic *gp_realloc __PROTO((generic *p, size_t size, const char *message));

#endif /* GNUPLOT_ALLOC_H */

--:-- alloc.h Bot L41 (C/I Abbrev)
M-x gtags-mode

```

3. gtags-mode を有効にすると, gtags-find-tag コマンドで定義の検索, gtags-find-rtag コマンドで参照 (使用) の検索ができるようになる。

```

emacs@nanamomo
File Edit Options Buffers Tools Minibuf Help
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "stdfn.h"

/* prototypes from "alloc.c" */
generic *gp_alloc __PROTO((size_t size, const char *message));
generic *gp_realloc __PROTO((generic *p, size_t size, const char *message));

#endif /* GNUPLOT_ALLOC_H */

--:-- alloc.h Bot L41 (C/I Gtags Abbrev)
M-x gtags-find-tag

```

```

emacs@nanamomo
File Edit Options Buffers Tools Minibuf Help
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "stdfn.h"

/* prototypes from "alloc.c" */
generic *gp_alloc __PROTO((size_t size, const char *message));
generic *gp_realloc __PROTO((generic *p, size_t size, const char *message));

#endif /* GNUPLOT_ALLOC_H */

--:-- alloc.h Bot L41 (C/I Gtags Abbrev)
Find tag: main

```

どちらのコマンドも, 見つけたいシンボルの上にカーソルがある状態で実行すると, シンボル名を入力しなくてもそのシンボルを検索してくれる。

4. それぞれ検索を実行すると, 見つかった定義が並んだバッファが開かれるので, その中のそれっぽい行で Enter キーを押すとそこへジャンプできる。

```

emacs@nanamomo
File Edit Options Buffers Tools Minibuf Help
main      203 docs/termdoc.c  main()
main      77 docs/windows/doc2html.c  main (int argc, char **argv)
main      68 src/beos/GPApp.cpp  int main(void) {
main      101 src/bf_test.c  main(void)
main      709 src/gplt_x11.c  main(int argc, char *argv[])
main      96 src/os2/gnupdrv.c  int main ( int argc, char **argv )
main     116 src/qtterminal/embed_example.cpp  int main(int argc, char*
main      51 src/qtterminal/gnuplot_qt.cpp  int main(int argc, char* arg*
main     222 src/strftime.c  main()
main     228 src/win/pgnuplot.c  main (int argc, char *argv[])

U:%*- *GTAGS SELECT* (D)main Bot L12 (Gtags-Select)

```

5. 実は, gtags-find-tag コマンドは, M-. というキーが割り当てられているので, 定義を参照したいシンボル名にカーソルを移動して M-. そして Enter キーを押すだけで, 定義へ飛ぶことができる。

Emacs への設定 これが有用だと思ったら, C や C++ のソースファイルが開かれたら自動的に, gtags-mode が実行されるようにしておくくと便利である。以下の手順で実行できる。以下が Emacs 起動時に実行されるようにしておけばよい。具体的には, /.emacs に以下の 2 行を追加する。

```

1 (setq c-mode-hook '(lambda () (gtags-mode 1)))
2 (setq c++-mode-hook '(lambda () (gtags-mode 1)))

```