第3回 UNIXゼミ

指導：長谷，チーフ：上川，サブチーフ：吉田，迫田

2001年5月22日

目 次

第 1 章 emacs の使い方 **2**
1.1 emacs とは . . . 2
1.2 キーボードのコマンドの表記方法 . . . 2
1.3 基本的なコマンド一覧 . . . 2
1.4 ファイルの作成，保存 . . . 2
1.5 終了の方法 . . . 2
1.6 文字の入力，移動 . . . 4
1.7 コンパイルの仕方 . . . 4
1.8 デバッガの起動 . . . 4
1.9 info を読む . . . 5
1.10 設定の方法 . . . 5

第 2 章 gcc **6**
2.1 gcc の処理 . . . 6
2.2 コンパイルの各段階における出力ファイルの抽出 . . . 6
2.3 定数の定義 . . . 7
2.4 ライブラリとのリンク . . . 7
2.5 コンパイラのオプション . . . 7
2.5.1 コンパイラの振る舞いの表示 . . . 7
2.5.2 C 言語オプション . . . 7
2.5.3 プリプロセッサのオプション . . . 8
2.5.4 ライブラリを指定するオプション . . . 8
2.5.5 デバッガやプロファイラのオプション . . . 9
2.5.6 最適化 . . . 9
2.5.7 アセンブラやリンカへのオプションの渡し方 . . . 10

第 3 章 makefile **11**
3.1 機能 . . . 11
3.2 目的 . . . 11
3.3 使用方法 . . . 11
3.4 makefile の指定 . . . 12
3.5 複数ファイルのコンパイル . . . 13
3.6 その他の便利な使用方法 . . . 15

第1章　emacsの使い方

1.1　emacs とは

emacs とは，LISP によるマクロが利用でき，オンラインドキュメントが充実したエディタのことです．現在 emacs というと，FSF Emacs と，XEmacs とのことを指す事が多いです．他には microemacs がしばしば使われているようです．Windows 版にも多数ありますが，そのなかでも一番多く利用されている物は Meadow と呼ばれています．

FSF Emacs とは，GNU プロジェクトの基本的な柱となるソフトウェアで，最強のエディタと呼ばれています．嫌う人も多いですが，十分その能力を理解して使いこなせれば，さまざまな事ができるエディタなので，他のエディタを使いにくく感じることもあるかも知れません．とくに mule と呼ばれる多言語サポートは他のエディタでは見られないレベルの物です．日本語の文字コードも JIS と EUC と SJIS と UTF-8 を自動認識して変換してくれるなどの他に類を見ない機能を有しています．

1.2　キーボードのコマンドの表記方法

emacs では，前に説明したように，他ではつかわないようなキーバインドの表記方法をします．`C-x` とかいてあったら，コントロールキーをおしながら x を押すという意味です．`C-M-t C-a` と書いてあったら，コントロールキーとメタキーを押しながら t を押し，それらのキーから一旦手を話します．その後，コントロールキーをおしながら a を押すという意味です．emacs のデフォルトの設定では複雑なキーバインドが多いので注意してください．

C はコントロール，S はシフト，M はメタキーです．

ところで，メタキーというのは，あまり聞かない名前ですが，PC のキーボードでは，Alt キーであったり，Windows キーだったりします．SPARC だったら◇キーです．

1.3　基本的なコマンド一覧

図 1.2 に，参考までにデフォルトでのキーバインドを示します．これは全て変更可能です．

1.4　ファイルの作成，保存

EMACS でファイルを作成する時には `C-x C-f` を押してから，ファイル名を入力します．そしてそのファイルが存在する場合には，オープンし，存在しない場合には新規に作成します．

保存する場合には，`C-x C-s` を押します．

1.5　終了の方法

`C-x C-c` を押します．

Table 1.1　用語の意味

単語	意味
バッファ	開いて編集中のファイルの事.
C-?	コントロールを押しながら?キーを押す
M ー?	ALT を押しながら?キーを押す
ウィンドウ	emacs の画面分割の単位
フレーム	X のウィンドウ

Table 1.2　一般操作用のキーバインド

キー	動作	備考
C-f	一文字前	
C-b	一文字後ろ	
C-p	一行上	
C-n	一行下	
M-b	単語一つ分後ろへ	
M-f	単語一つ分前へ	
M-d	カーソルの位置にある単語を削除	
C-d	カーソルの位置にある文字を削除	
DELETE / BS	カーソルの後ろにある文字を削除	
C-h	ヘルプ	
TAB	行を規則通りにインデントする	
C-a	行頭へ移動	
C-e	行末へ移動	
C-j	改行して規則通りにインデント	
C-s	文字列を検索する	そのまま一度エンターをおすと，日本語の文字列も検索できる.
C-v	一画面下に移動	
M-v	一画面上に戻る	
M-/	単語の補完	開いている全てのバッファから手がかりを得て，現在の単語の残りを入力.
M-.	タグを検索	現在カーソルがあるところの単語がどこで定義されているのかという情報の検索 (C のプログラムなど).

Table 1.3　ファイル関係のコマンド

キー	動作	備考
C-x C-s	ファイルをセーブする	
C-x C-c	emacs の終了	
C-x C-f	ファイルを開く	
C-x C-b	現在開いているバッファの一覧の表示	今開いているファイルが全部リストされる.
C-x C-o	別ウィンドウに移動	
C-x C-k	現在のファイルを閉じる	
C-x b	バッファの選択	今開いているファイルの一覧から編集したいファイルを選ぶ.

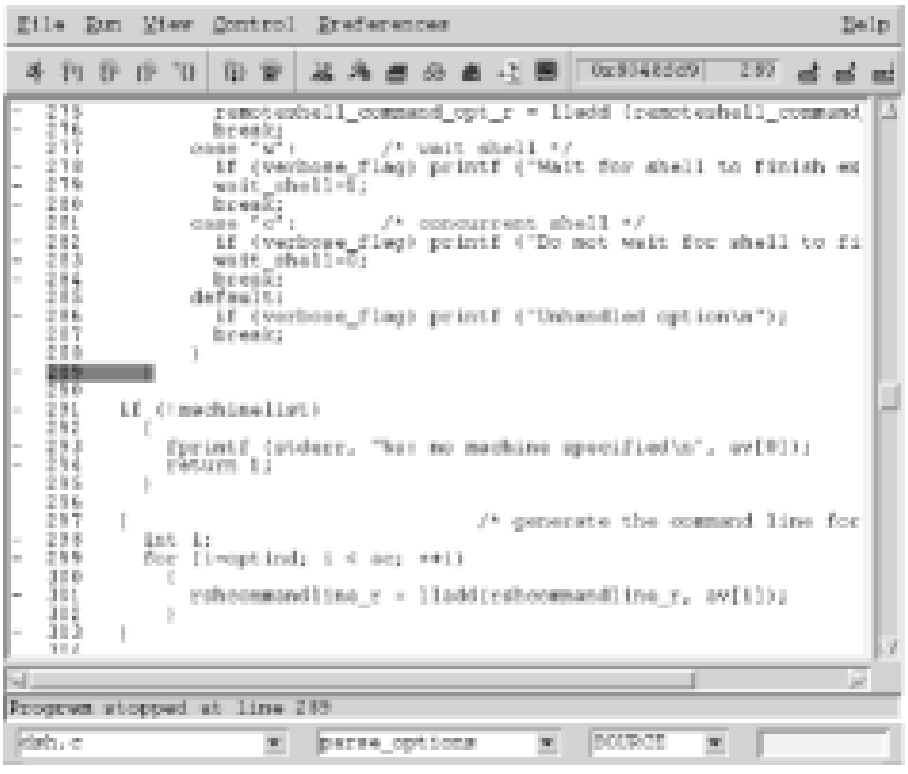

Fig. 1.1　Insight の画面

1.6　文字の入力，移動

文字の入力と移動は，常識的な範囲で行えます[1]．より詳細な使い方に付いては，C-h b を押すと，コマンドのリストが出て来ます．大量にあるので，全部を覚えている人は居ないと思われます．好きなように設定して下さい．

文字領域の選択はコントロールとスペースにより領域選択の開始となります．それを C-w で切り取り，C-y で張り付けるということになります．

1.7　コンパイルの仕方

コンパイラを起動するには，XEmacs では，compile ボタンをおします．Emacs では，M-x compile [Enter] とおします．そうすると make を起動してくれます．これでうまくコンパイルが実行されるようにするためには，Makefile を書く必要があります．

これは不便なので，これにショートカットを設定する場合には，ホームディレクトリにある.emacs というファイルを書き換え，以下の設定を追加すると，X-x m でコンパイルが実行されるようになります．

```
(global-set-key "\C-xm" 'compile)
```

1.8　デバッガの起動

GDB というデバッガを起動するには，M-x gdb というようにします．詳細については，gdb の解説を見てください．emacs では，ソースファイルの実行中の箇所に対応するような行に→がでたりします．これは emacs の GUD[2]モードと呼ばれています．

テキストモード以外も利用できるのなら，図 1.1 に示すような Insight デバッガインタフェースなども利用できます．そちらのほうが見ためも良いと思われます．ただ，emacs を使って全てが行いたいという人や，テキストベースで全てが行いたい人にとっては，emacs は理想的です．

[1] bash と共通しています
[2] grand unified debugger

1.9 info を読む

emacs は説明書がたっぷりと付いてきます．C-h ?と押すと，リストが出て来ます．C-h i で info が出て来ます．info は GNU のアプリケーションの標準となっている説明書の形式です．TeX で書かれていて，ハイパーリンクがされています．一般的な info 文書は，100 ページを越えた大作のマニュアルが多いです．

emacs の info の説明書は，印刷すると 700 ページ程度あります．

1.10 設定の方法

emacs は lisp で全て設定ファイルを書く事ができます． /.emacs というファイルを起動時に読み込むので，そこに自由に設定を書いておく事ができます．例えば，

```
(global-set-key "\C-s" 'save-buffer)
(global-set-key "\C-q" 'save-buffers-kill-emacs)
(global-set-key "\C-c" 'kill-ring-save)
(global-set-key "\C-v" 'yank)
(global-set-key "\C-x" 'kill-region)
(global-set-key "\C-f" 'isearch-forward)
(global-set-key "\C-o" 'find-file)
```

とすると，Windows で使いなれているようなキーバインドで利用する事ができます．emacs のデフォルトは伝統的かつ複雑怪奇なので，積極的に利用しやすいように変更する事を推奨します．ただ，一応 bash[3]と同じキーバインドであるため，デフォルトのまま使い，使いなれてしまうということも悪くは無いでしょう．

[3]デフォルトで bash は emacs のデフォルトのキーバインドと似ているキーバインドになっている

第2章　gcc

2.1　gcc の処理

実行形式の作成における gcc の処理は，正確にはいくつかの工程に分かれています．ここで，それぞれの工程でどのようにしてデータが流れているか，またはどのようなファイルを作成していくかを Fig. 2.1 に示します．

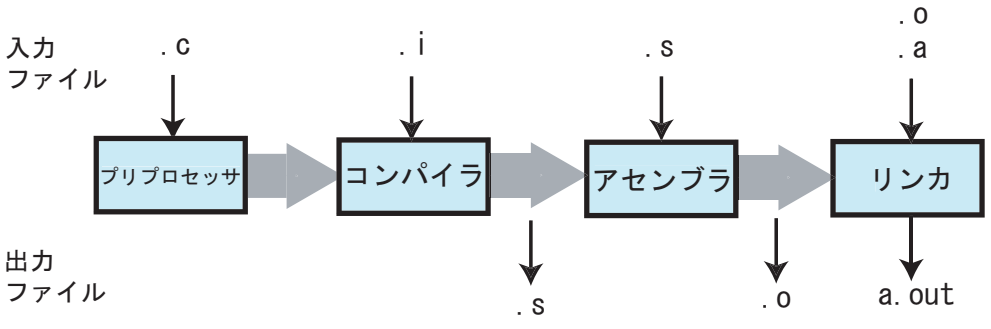

Fig. 2.1　コンパイルの各段階

- **前処理**
  この工程は，#define や#include，#if のような指示文を処理します．
- **コンパイル**
  入力ファイルからアセンブル言語ソースコードを生成します．アセンブラは一般にすぐに呼び出されるので，出力結果は通常はファイルに保存されません．
- **アセンブル**
  入力としてアセンブラ言語ソースコードを取り込み，.o という拡張子のオブジェクトファイルを生成します．
- **リンク**
  最終段階になります．.o モジュールが実行形式ファイル内の適切な場所に配置されます．プログラムが参照するライブラリ関数も実行形式ファイル内に配置されます．

2.2　コンパイルの各段階における出力ファイルの抽出

ある中間段階での gcc の出力結果を保存しておくと，デバッグや中間コードの直接操作などのために使用することができます．それぞれのファイルをどのようにして抽出するかを以下に示します．

- **前処理**
  gcc の-E オプションにより，プログラムをコンパイルするのでなく，前処理されたソースコードを標準出力から得ることができます．
- **コンパイル**
  アセンブラ言語の出力結果を保存するには，-S オプションを指定して gcc を実行します．ソースコード名の.c ではなく.s で終わる名前のファイルが生成されます．

- **アセンブル**
  前述のとおり，-c を指定して実行すると，.o で終わる名前のオブジェクトファイルが生成されます.

2.3 定数の定義

よく使われるオプションとして-D オプションがあります．これは，ソースコードに#define を記述したような振る舞いをします．例えば，以下のように記号の値を設定します.

$ **gcc -c -DDOC_FILE=¥"info¥" -DUSE_POLL filter_driver.c**

１番目の-D オプションは，DOC_FILE という値に「info」という文字列を設定しています．このオプションは，プログラムがオープンしているファイルを制御する場合に便利です．２番目の-D オプションでは，USE_POLL に記号を定義しています（ここでは既定値の１を設定しています）.

標準のディレクトリのパスに取り込むべき（include）ファイルがない場合は，-I オプションを指定します．例えば，ソースコードが２つのディレクトリにまたがっている場合，つまり，ソースコードが/usr/src で，ヘッダファイルが/usr/headers にあるとします．そこで，/usr/src ディレクトリでコンパイルしている場合は，次に示すコマンドにより，ヘッダファイルを探す場所を gcc に指示することができます.

$ **gcc -c -I../headers filter_driver.c**

2.4 ライブラリとのリンク

もう１つよく使われるコンパイラのオプションとして，ライブラリを指定する-l があります．以下にその典型的な例を示します．入力はオブジェクトファイルなので，コマンドはリンカを実行しているだけです.

$ **gcc -o plot main.o plot_line.o -lm**

-lm オプションは，数学ライブラリを指定します．-l*name* を指定すると，システムは標準的なライブラリが保存されているディレクトリ（通常は/usr/lib）から lib*name*.a を探します．したがって，/usr/lib/libm.a（または libm.so）から数学ライブラリを探します.

ライブラリが標準的なところにない場合があります．その場合は，-L オプションを使うと，ライブラリのために特定のディレクトリを探すことができます．例えば，次のように指定します.

$ **gcc -o plot -L/src/local/lib main.o plot_line.o -lm**

このコマンドは，/src/local/lib にあるライブラリを最初に探し，次に標準の場所を探すように gcc に指示しています．ローカルなバージョンのライブラリを/src/local/lib に置いておいたとすると，/usr/lib にある標準バージョンよりも優先して使用することになります.

2.5 コンパイラのオプション

以降においては，gcc で使用できるその他の重要なオプションについて説明します.

2.5.1 コンパイラの振る舞いの表示

-v オプションは，コンパイラのバージョン番号と，各パスがどのように実行されていくかの完全な詳細情報を出力します．このオプションは，どのオプションを使ってプログラムがリンクされているかを正確に調べる場合に特に有用です.

2.5.2 C 言語オプション

gcc が生成する警告メッセージを制御するオプションがいくつかあるので，以下で説明します.

- **-w**
  すべての警告メッセージを抑止します.
- **-W**
  正当ソースコード上の習慣に関する追加警告メッセージを出力します.
- **-Wall**
  疑わしいソースコード上の習慣に関するさらに多くの警告メッセージを出力します.

- **-Wtraditional**
  「Kerninghan and Ritchie」の C 言語の定義と「ANSI C 言語定義」の両方とも正当だが，振る舞いが異なるソースコードに関して警告メッセージを出力します.

- **-Werror**
  すべての警告をエラーとします．つまり，警告が発生した場合には，オブジェクトコードを生成しないようにします.

次に，C 言語の様々な機能を制御する方法について説明します.

- **-traditional**
  ANSI C 言語規格よりも以前の C 言語のソースコードは，このオプションを使ってコンパイルする必要があります.

- **-ansi**
  ANSI C 言語規格に従って書かれた比較的最近のコードは，このオプションを使ってコンパイルする必要があります.

- **-pedantic**
  ANSI C 言語規格に求められている警告メッセージをすべて表示します．C 言語に関する FSF の拡張機能の使用を全面的に禁止し，拡張部分の利用はエラーとみなします.

2.5.3 プリプロセッサのオプション

以下では，cpp というプリプロセッサをコマンド行から制御するオプションについて説明します.

- **-M**
  ソースコードを読み込んで，どのようなファイルを取り込んでいるかを調べ，make 向けの依存リストを出力します.

- **-C**
  プリプロセッサは通常，すべての注釈をプログラムから削除するのですが，このオプションを指定すると注釈を削除しません．このオプションを使用するときは，必ず-E オプションを併用します.

2.5.4 ライブラリを指定するオプション

以下については，あまり使われないが特定の状況では有用なオプションになります.

- **-nostartfiles**
  リンクする際にシステムに標準の起動ファイルを使いません．クロスコンパイルや，組み込みシステム向けのコンパイルを行う場合，つまり固有の起動ファイルを用意したい時に有用です.

- **-nostdlib**
  リンク時に標準のライブラリと起動ファイルを使いません．ここでも，ユーザ固有のライブラリを利用したり，既定のライブラリよりも優先させたいときに便利です.

- **-static**
  共有ライブラリではなく，静的ライブラリのみリンクします.

- **-shared**
  静的ライブラリではなく，共有ライブラリが使用可能であれば，できるだけそれを使用します.

2.5.5 デバッガやプロファイラのオプション

以下に示すオプションを用いて，コンパイラが様々なプロファイラやデバッガのために追加のコードや拡張の記号表を生成するように指示をします．これらのオプションは，開発中のソースコードのデバッグやチューニングの際に極めて有効であるが，最終版のプログラムには使用しません．

- **-pg**
  gprof コマンドによるプロファイルをとるため，必要なプログラムをリンクします．このオプションでコンパイルしたプログラムを実行すると，gmon.out という名前のファイル（プログラムの実行集計情報が入っている）が生成されます．gprof というプロファイラはこのファイルを読み込み，プログラムそれぞれの実行時間を記述している表を作成します．

- **-g**
  デバッグ用の拡張記号表を生成します．これにより，gdb をつかってデバッグすることができます．

2.5.6 最適化

gcc は洗練された最適化コンパイラでもあります．ほとんどのシステムでは通常，システム標準のコンパイラよりも高速なコードを生成します．ここに，最もよく使われるコンパイルオプションを示します．

- **-O**
  -O1 と同じ．

- **-O0**
  最適化を行いません．最適化を無効にすると，gcc はデバッグしやすいコードを生成しようとします．つまり，任意の２つの文の間にブレークポイントを設定し，変数を修正することが可能になり，プログラムはそうあるべき通りに振る舞うようになります．

- **-O1**
  コンパイラは，コンパイルされたコードのサイズとその実行時間の両方の削減をはかります．コンパイル時間は，-O0 の場合よりもかかり，コンパイル時に必要なメモリも多くなります．

- **-O2**
  -O1 よりもさらに最適化を行います．生成されたコードはより高速になります．

- **-ffast-math**
  ANSI や IEEE といった規格に準拠していない浮動小数点数の数学的最適化を行います．このオプションを使ってコンパイルされたコードは，正しくない結果をもたらす可能性があるが，若干高速になります．

- **-finline-math**
  「単純な関数」をすべて呼び出し側で展開します．コンパイラは，どの関数が「単純」なのか否かの決定にとりかかります．

- **-fno-inline**
  すべてのインライン展開を無効にします．ソースコード内の inline キーワードで指定されているインライン展開をも無効にします．

- **-funroll-loops**
  コンパイル時に繰り返し回数が固定と判明したすべてのループを展開する．

2.5.7　アセンブラやリンカへのオプションの渡し方

gcc は，コマンド行からオプションをアセンブラやリンカへ直接渡すことができます．

-Wa,*option-list*　　*option-list* をアセンブラへ渡す．

-Wl,*option-list*　　*option-list* をリンカへ渡す．

どちらも，*option-list* は，アセンブラやリンカが認識するオプションの単なるリストです．リスト内には空白があってはいけません．リスト内にオプションを複数指定する場合にはコンマで区切ります．

第3章　makefile

3.1　機能

主にコンパイル作業に使われますが，もちろん単なるプログラム実行にも使うことができます.

3.2　目的

hello.c というプログラムをコンパイルするには

```
$ gcc hello.c
```

とコマンドするのですが，オプションなどをつけると

```
$ gcc -Wall -O2 -ascii -pedantic -g hello.c -lm -o hello
```

などと，複雑になります．そこでプログラマは Makefile という名前のファイルにどのようにコ ンパイルするかの指示を書いておき，make はそれを読んで依存関係に従っ てコンパイルを実行することができます.

3.3　使用方法

Makefile はコンパイルしたいファイル hello.c のあるディレクトリに置いてください.

```
hello : hello.o
hello.o : hello.c
```

この hello は,「生成したい」実行ファ イル名です．生成したい対象のことを「ターゲット」と呼びます．Makefile には，ターゲットは行頭からつめて書いてください．でないと動きません.

また,「方法」を省略すると Default になるが，自分で「方法」指定することができる.「方法」は TAB 一つ血ン伝とを入れて書く.

例：

```
%.dvi: %tex
 platex $<
```

ここで＄＜は「：」の右側のファイルで，＄@は「：」の左側のファイル.「%」は「：」の右と左で同じファイルの radix 出ある場合のその radix を示す.

hello.c と Makefile のあるディレク トリで make を実行します.

```
$ ls
Makefile    hello.c
$ make
cc     hello.c   -o hello
$ ls
Makefile    hello.c    hello
$ ./hello
```

3.4 makefile の指定

コンパイラを gcc に指定したり，コンパイル時の作成されたファイルなどを消去する際には makefile に指定します.

gcc でコンパイルするならば,

```
CC = gcc
```

と指定します.

-Wall と-02 オプションを付けてコンパイルしたいなら

```
CFLAGS = -Wall -O2
```

を makefile に指定します.

-lm などのリンクオプションを付けたいなら

```
LOADLIBS = -lm
```

不要なファイルを削除するためには （例は.c の拡張子のファイルを削除する）

```
clean :  【改行】
【タブ】 rm -f *.c*~
```

を追加します．これらはすべて，最初のターゲット (例では hello:) より 前に書きます．以上の全てを追加した Makefile は次のようになります.

```
CC = gcc
CFLAGS = -Wall -O2
LOADLIBES = -lm
clean :
     rm -f *.c*~
hello : hello.o
hello.o : hello.c
```

この Makefile を使って make してみましょう.

```
$ touch hello.c
$ make
gcc -Wall -O2    hello.c -lm  -o hello
```

gcc が使われてますし，指定したオプションがついていますね？リンカへのオプションも LDFLAGS で指 定できます.

単独のファイルのコンパイルの時でも，make の 4 文字で，Mule/Emacs の 中からなら C-cC-c でコンパイルできるのはとっても楽だと思 いませんか？なにせプログラミングにはデバッグ，つまり再コンパイル がつきもの，日を置いてからのコンパイルのときにも どんなオプションが必要だったから悩むこともなくなります.

3.5 複数ファイルのコンパイル

今度は複数のファイルからなるプログラムをコンパイルしてみましょう！

test.c

```
#include"header.h"
int main()
{
print_sub();
return 0;
}
```

sub.c

```
#include<stdio.h>
void print_sub()
{
      printf("welcome!makefile\n");
}
```

header.h

```
void print_sub();
```

サンプルプログラムの test は，３つのファイル test.c と sub.c と header.h から出来ていて，これらから実行ファイル test を作るには

```
$ gcc -c test.c  -o test.o
$ gcc -c sub.c  -o sub.o
$ gcc test.o sub.o test
```

としなければならないのですが，makefile へ指定すればコンパイルが楽になります.

makefile

```
cc = gcc
CFLAGS = -wall, -02
test : test.o sub.o
test.o : test.c
sub.o  : sub.c
test.c : header.h
```

```
$ make
cc -wall, -02   -c -o test.o test.c
cc: unrecognized option ‘-wall,’
cc: unrecognized option ‘-02’
cc -wall, -02   -c -o sub.o sub.c
cc: unrecognized option ‘-wall,’
cc: unrecognized option ‘-02’
cc   test.o sub.o   -o test
$
$ ./test
welcome!makefile
sakota@forte:~/test1$
```

と実行できます.

もしも，test.c を

test.c

```
#include"header.h"
 int main()
{
      int i;
      for(i=0;i<5;i++)
           print_sub();
       return 0;
}
```

と変更した場合でも，全てのファイルをコンパイルせず，make を実行すればよいのです.

```
$ make
cc -Wall, -O2   -c -o test.o test.c
cc   test.o sub.o   -o test
$
$ ./test
welcome!makefile
welcome!makefile
welcome!makefile
welcome!makefile
```

```
welcome!makefile
sakota@forte:~/test1$
```

といったコンパイルの際はとても便利です.

3.6 その他の便利な使用方法

tex ファイルを dvi へ変換する為の makefile

LaTeX のファイルの例：

```
RADIX=unix_resume_3

all: $(RADIX).pdf $(RADIX).gs.pdf

%.pdf:%.ps
 ps2jpdf $< $@

%.ps:%.dvi
 dvips -o $@ $<

%.dvi:%.tex
 platex $<

%.gs.pdf:%.ps
 ps2pdf $< $@

print: $(RADIX).ps
 lpr $<

preview: $(RADIX).ps
 gnome-gv $<
```