第2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18日

目 次

第1章　シェルの活用

1.1　シェルとは

シェルとは，Linux のコマンドインタープリタのことを言います．コマンドインタープリタはユーザが入力したコマンドを受け取り，いくつかの解釈や展開などを行った後，Linux 本体に処理を依頼するという働きを持っています．

Linux 上には数種のシェルがありますが，今回は Linux 標準のシェルとして一般によく使用されている bash について説明します．

1.1.1　シェルの位置づけ

Linux は，本来のオペレーティングシステムとしての働きをするカーネルと呼ばれる部分と，ユーザーとカーネルの仲介をするシェルと呼ばれる部分からなります．

Fig. 1.1 のように，シェルはログインしているユーザに１つずつ用意されます．

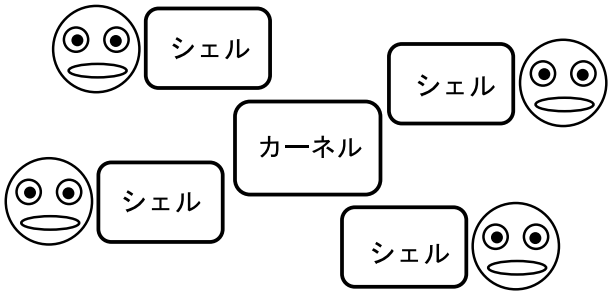

Fig. 1.1　シェルとカーネル

1.1.2　シェルの仕事

シェルは，ユーザからのコマンド入力を受け取ると，Fig. 1.2 のように，ヒストリーリストの展開，ファイル名の展開，エイリアス（別名）の置き換え，リダイレクション解釈などを行い，カーネルに処理を依頼します．

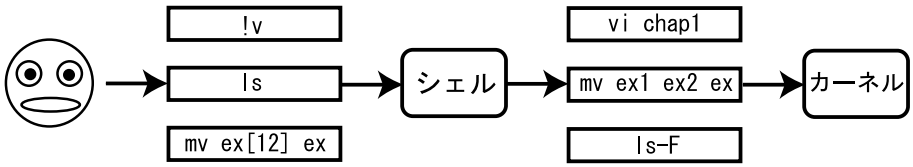

Fig. 1.2　シェルの仕事

1.2　コマンドラインの編集

通常，正しく設定されている bash の環境では，コマンドラインを編集するために，[←]と[→]でのカーソル移動，[BackSpace]による文字削除，[↑]と[↓]による履歴の参照などができます．

ここでは，それ以外に覚えておくと便利な bash のキー機能について説明します．なお，Emacs や bash のマニュアルでは，キー操作の表示に特殊な記法を使います．その記法に対するキー操作の対応表を Table 1.1 に示しておきます．

1.2.1　行頭・行末への移動

cho hello_　e が足りない

Table 1.1　コマンドとキー操作の対応表

コマンド	操作
C-a	Ctrl を押しながら，a を押す.
M-b	ESC を押した後に，b を押す.
M-C-h	ESC を押した後に，Ctrl を押しながら，h を押す.

↓　C-a で行頭に移動

cho hello

↓　e を入力

echo hello

行頭への移動は C-e を使います.

1.2.2　単語単位の移動

echo ello_shota　h が足りない

↓　M-b で１つ左の単語の先頭に移動する

echo ello shota

↓　h を入力

echo hello shota

１つ右の単語の先頭に移動するには M-f を使用します.

1.2.3　単語単位の削除

echo hello_shota　hello を bye に変更

↓　M-C-h でカーソルの左の単語を削除

echo _shota

↓　bye を入力

echo bye shota

カーソルの右の単語を削除するには M-d を使用します.

1.2.4 行単位の削除

echo I am hungry　カーソル以下を削除したい

↓　C-k でカーソルから行末まで削除

echo _

行すべてを削除するには C-u を使用します.

1.2.5 削除した文字列の挿入

M-C-h，M-d，C-k，C-u で削除された文字列は，bash が一時的に記憶しています．その記憶している文字列を C-y でカーソル位置に挿入することができます.

echo bye and hello

↓　C-k でカーソルから行末まで削除

echo _

↓　C-y で記憶されている文字列を挿入

echo bye and hello_

1.3 履歴機能

bash はユーザが入力したコマンドを記憶しています．履歴を表示するには history コマンドを使用します.

```
$ history Enter
1 ls
2 date
3 history
```

ここで，最も最近に実行されたコマンドは「history」であり，その前が「date」，さらにその前が「ls」となります.

1.3.1 直前のコマンドの実行

まず最初に

```
$ echo hello Enter
hello
```

とコマンドを実行します．このとき，最も最近に実行されたコマンドは「echo hello」になります．ここで,「!!」というコマンドを実行すると直前のコマンドである「echo hello」が実行されます.

```
$ !! Enter
echo hello
```

hello

また，ここで「^hello^bye」というコマンドを実行すると，直前のコマンドである「echo hello」の「hello」を「bye」に変更して実行されます.

```
$ ^hello^bye Enter
echo bye
bye
```

1.3.2 履歴番号での実行

ここでもう一度「history」と実行します.

```
$ history Enter
1 ls
2 date
3 history
4 echo hello
5 echo hello
6 echo bye
7 history
```

次に,「!2」というコマンドを実行すると履歴のリストのなかの２番のコマンド，つまり「date」が実行されます.

```
$ !2 Enter

 date
Tue May 15 21:38:04 JST 2001
```

1.3.3 履歴の検索による実行

さて，履歴のなかのコマンドをある文字列で検索して実行するには,「!l」というコマンドを実行します．この場合は,「l」で始まるコマンドを履歴の中から検索し実行しますから,「ls」が検索されて実行されます.

```
$ !l Enter
ls
‥(省略)
```

1.4 ファイル名の省略

bash ではメタキャラクタという特別な文字を使ってファイル名を省略することができます．メタキャラクタには Table 1.2 に示すものがあります.

Table 1.2　メタキャラクタ

*	任意の文字列を示す（空を含む）
?	任意の１文字を示す
[文字１文字２]	文字１，文字２のいずれか１文字を示す
{文字列１, 文字列２, ‥}	文字列１，文字列２，文字列 ‥ のいずれかを示す
~ログイン名	指定のログイン名を持つユーザのホームディレクトリの絶対パスネームを示す（ログイン名を省略した場合は自分自身のホームディレクトリ）

1.4.1　メタキャラクタの使用例

それでは，メタキャラクタを簡単な例で使用してみます．まず，最初にカレントディレクトリに「test1_dir」「test2_dir」という名前のディレクトリを作成します．

```
$ mkdir {test1,test2}_dir Enter
↓　「test1_dir」「test2_dir」を作成
$ ls Enter
‥　test1_dir test2_dir
↑　「test1_dir」「test2_dir」が作成されているのを確認
```

次に，いま作成した「test1_dir」「test2_dir」ディレクトリを削除します．

```
$ rm -r test[12]_dir Enter
↓　「test1_dir」「test2_dir」を削除
$ ls Enter
‥
↑　削除されているのを確認
```

以上が簡単なメタキャラクタの使用例になります．

1.4.2　補完機能

メタキャラクタとは違いますが，bash には補完機能という別のファイル名の省略機能があります．補完機能は，ファイル名やディレクトリ名をすべて入力しなくても途中で TAB を押すことで補完することができます．

簡単な例で使用してみましょう．まず，最初にカレントディレクトリに「test」というディレクトリを作成します．

```
$ mkdir test Enter
↓　「test」ディレクトリを作成
$ ls Enter
‥　test
↑　作成されているのを確認
```

次に，補完機能を利用して「test」ディレクトリを削除します．

```
$ rm -r t TAB
```

↓　「t」で始まるディレクトリを検索して，残りの文字を補完

$ rm -r test/ Enter

↓　「test」ディレクトリを削除

$ ls Enter

..

↑　削除されているのを確認

候補が複数ある場合は，TAB を２回押すと候補一覧が表示されます.

1.5　エイリアス機能

bash では「alias」コマンドを使ってコマンドに別名を付けることができます．例えば,「alias h=”history”」とすれば，それ以降において「h」だけで「history」コマンドを実行することができます．一度設定したエイリアスを解除するには「unalias h」のようにします．また，設定しているエイリアスの一覧を見るには,「alias」と入力します.

しかし，エイリアス機能はコマンド入力において混乱する要因になりやすいので，通常はあまり使われません．知識として知っておいてください.

1.6　リダイレクション機能

Linux には，コマンド実行時などに利用する３つの標準ファイルが用意されています（Table 1.3 を参照）．通常のコマンドは，これらの標準ファイルをコマンドの入力と実行結果やエラーメッセージの出力先に指定するように設計されています．そして，それぞれがデフォルトで端末に割り当てられています.

リダイレクション機能とは，これら標準ファイルを端末以外のファイルに切り替える機能です.

Table 1.3　標準ファイルとリダイレクション記号

名前	使用目的	リダイレクション記号
標準入力ファイル	コマンドがユーザからの入力として扱う．デフォルトは端末（キーボード）	< ファイル名
標準出力ファイル	コマンドの実行結果の出力先．デフォルトは端末（画面）	> ファイル名
標準エラー出力ファイル	コマンドからのメッセージの出力先．デフォルトは端末（画面）	>&ファイル名

例えば，次のように利用できます.

$ cat > test1 Enter

↓　これにより，キーボードの入力が「test1」ファイルに出力されます

This is a sample text. Enter

C-d

↓　サンプル文を入力します.

$ cat test1 Enter

This is a sample text.

↑　「test1」ファイルの内容を確認

また，いま作成した「test1」ファイルを用いて，次のように利用することもできます.

$ cat test1 > test2

↓　これにより,「test1」ファイルの内容を「test2」ファイルに出力

$ cat test2

This is a sample text.

↑　「test2」ファイルの内容を確認

以上がリダイレクション機能の簡単な使用例になります．最後にいま作成した「test1」「test2」ファイルを削除しておきましょう．

$ rm test[12] Enter

↓　メタキャラクタで２つのファイルを同時に削除

$ ls Enter

..

↑　削除されているのを確認

1.7　ジョブの実行と管理

UNIX では複数のジョブ[1]を同時に処理することができます．そのため，あるコマンドの実行が終了しないうちに，別のコマンドを実行することができます．

すぐに別のコマンドを実行したい場合は，コマンドラインの末尾に「 &」と記述します．これにより，そのコマンドはバックグランドジョブ[2]として扱われ，ジョブ番号が表示された後，すぐにシェルプロンプトが表示され，次のコマンドが実行できるようになります．

リモートログインで UNIX を用いる場合，ログアウト後も UNIX に処理をさせておきたいという場合があります．その場合は，さらにコマンドラインの頭に「nohup 」と記述します．

また，普通に何も記述しないでコマンドを実行した場合は，フォアグランドジョブ[3]として扱われ，そのジョブが終了するまで，次のコマンドは実行できません．

1.7.1　「xeyes」を用いた実行例

それでは，実際にフォアグランドとバックグランドでジョブを実行させてみましょう．ここでは,「xeyes」というコマンド（Fig. 1.3 のようなウィンドウが表示される）を使ってジョブの実行と管理についての例を示します．

$ xeyes Enter

[1] bash などのシェルが管理しているコマンドの実行単位
[2] つまり，バックグランド（裏側）で実行されるジョブのこと
[3] フォアグランド（表側）で実行されるジョブ

↓ 「xeyes」をフォアグランドジョブとして実行

C-z

[1]+ Stopped xeyes

$

↓ フォアグランドジョブを一時停止

$ bg Enter

[1]+ xeyes &

$

↓ バックグランドジョブとして再開

$ jobs Enter

[1]+ Running xeyes &

$

↓ 現在実行中のジョブ一覧を表示

$ fg Enter

xeyes

↓ バックグランドジョブをフォアグランドに移動

C-c

$

↓ フォアグランドジョブを中断

$ xeyes & Enter

[1] 13577

$

↓ 「xeyes」をバックグランドジョブとして実行

$ jobs Enter

[1]+ Running xeyes &

↓ 現在実行中のジョブ一覧を表示

$ kill %+ Enter

$

↑　バックグランドジョブを中断

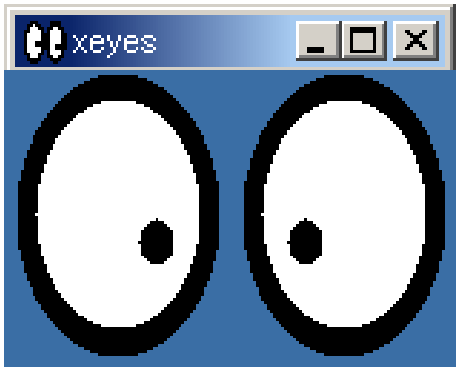

Fig. 1.3　表示されるウィンドウ

1.8　シェルスクリプト

UNIX のシェルには，ユーザがプロンプトに対して入力したコマンドを実行するほかに，シェルスクリプトと呼ばれる独自のプログラミング言語を実行する機能を備えています．簡単なシェルスクリプトを実行する前に，変数についての説明をします．

1.8.1　シェル変数と環境変数

シェルスクリプトではシェル変数という変数を利用してプログラムを実行できます．シェル変数はコマンドラインから簡単に設定できます．例えば

$ name=shota Enter

$ echo $name Enter

shota

のように，name という変数を定義し，値として shota と設定すると，「$変数名」と記述することでシェル変数を参照できます．またシェル変数の一覧を表示するには set コマンドを使用し，シェル変数を削除するには unset コマンドを使用します．

ここで，シェル変数はその起動しているシェルにのみ有効な変数であり，それ以外[4]においては使用することはできません．したがって，使用できるようにするには

$ export name Enter

[4]例えば，bash から起動された bash においてなど

とすることでシェル変数を環境変数にする必要があります．ここで，環境変数とは UNIX 全体の環境における共通の変数であり，どこでも使用することができます．

1.8.2 シェルスクリプトの実行

それでは，実際にシェルスクリプトを作り，それを実行してみましょう．

$ cat > test Enter

#! /bin/sh Enter

echo $name Enter

C-d

↓ 「test」という名前でシェルスクリプトを作成

$ chmod 744 test Enter

↓ 「test」に実行属性を付ける

$ ./test Enter

shota

↑ 作成したシェルスクリプトが正しく実行されているのを確認

以上のようにシェルスクリプトでは，コマンドを組み合わせてオリジナルの新しいコマンドを作ることができます．

1.9 環境の自動設定

環境変数の設定やエイリアスの設定などはログアウトすると無効になります．常に同じ環境で作業をしたい場合には，ログインするたびに同じ設定を繰り返し行わなければなりません．

そこで，これらの設定をログイン時に自動的に行う機能を説明します．

1.9.1 .bashrc ファイル

ホームディレクトリにある「**.bashrc**」ファイルの内容は，ユーザが **bash を起動したとき**に，自動的に読み込まれ実行されます．

内容の記述方法は，通常のコマンド行の入力と同じように１行ずつ記述するだけです．以下に簡単な例を示します．

```
export LANG='japanese'\\

export PAGER='less'\\

alias ls='ls -F'\\

alias rm='rm -i'\\

alias h='history'
```

このファイルはほとんどの場合，システム管理者があらかじめ各ユーザのホームディレクトリに作成してあります．内容もそれぞれのシステムにあったものが記述されています．特に必要のない限り，元からある行は変更せずに必要な設定を追加してください．

もし，ホームディレクトリに「.bashrc」がない場合には，vi エディタなどで作成してください．

1.9.2 .bash_profile ファイル

ホームディレクトリの「**.bash_profile**」は,「.bashrc」と同様にコマンドを記述しておくと，**ログイン時**に自動的に読み込まれて実行されます．

.bashrc をログイン時にも実行したい場合には.bash_profile ファイルの最後に次の１行を書いておきます．

```
source \verb|~/.|bashrc
```

1.9.3 実用例

簡単な実用例として，パスの設定について説明します．ここでは,「/usr/local/java/bin」の中のコマンドを使いたい場合について説明します．

$ java Enter

bash: java: command not found

↓ 「java」というコマンドは使えない

$ /usr/local/java/bin/java Enter

...

↓ 場所を指定すると,「java」コマンドが使える

$ export PATH=$PATH:/usr/local/java/bin Enter

↓ パスで場所を指定しておく

$ which java Enter

/usr/local/java/bin/java

↓　「java」コマンドの場所がパスによって確認される

$ java Enter

…

↑　パスにより「java」コマンドの場所を探して実行

しかし，ここで設定したパスは一旦ログアウトすると無効になってしまいます．つまり，ログインするたびに設定する必要がでてきます．そこでこれを自動化するために，.bashrc ファイルを用います．具体的には，.bashrc ファイルのパスの設定の行に通したいパスを記述しておきます．

第2章　viとgccをつかってプログラムの作成

2.1　viで書く

viはコマンドラインから

```
$ vi math.c
```

というようにして起動します.

viは，コマンドを入力するモードと文字を入力するモードがあります.

文字を入力するモードからは，[ESC]を押す事によりコマンドを入力する

モードに入れます.

表2.1に利用しそうなコマンドの一覧があります．viをもっと詳しく知りたい場合は，viの説明書を読んで下さい.[1]

2.2　gccでのコンパイル

まず,

```
$ vi math.c
```

というように入力して，viを起動し，以下のようなCのプログラムを入力します.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (int ac, char ** av )
{
  printf ("%g\n", sqrt ( atof ( av[1] ) ) );
  return 0 ;
}
```

そうして，コマンドラインでCコンパイラであるgcc[2]を実行してみることにします.

まず，コンパイルとリンクを同時にする方法を試してみると，-lmがないとsqrtが見付からないためエラーがでます.

```
[21:52:01]atoron:/tmp>gcc -o math math.c
math.o: In function ‘main’:
math.o(.text+0x27): undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
[21:52:01]atoron:/tmp>gcc -o math math.c -lm
```

同様にして，コンパイルとリンクを別々にやる方法も試してみます．こちらも，-lmがないと，エラーがでます．-lmがついているとめでたくリンクに成功します.

[1]マクロなどを触りはじめるとかなり複雑な事までできるようです.

[2]GNU Compiler Collection の GNU C Compiler

Table 2.1　vi のキーバインド

キー	意味	解説
i	insert	カーソルのある位置で文字入力モードにはいる
a	append	カーソルのある位置の一文字あとから入力モードにはいる
J	Join line	次の行を現在の行の最後に繋げる
x	cross	現在カーソルがある文字を消す
h		左
j		下
k		上
l		右
:q `[enter]`	quit	終了
:wq `[enter]`	write and quit	セーブして終了
:q! `[enter]`	quit force	セーブしないで終了
:w `[enter]`	write	セーブ
dd	delete line	一行削除
de	delete word	一単語削除
yy	yank？	一行コピー
p	paste	ペースト
r	replace	次に入力する文字と一文字置き換え
o	open line？	次の行を一行あけて入力モードに移行
/*keyword*	search	文字列検索

```
[21:52:01]atoron:/tmp>gcc -c math.c -o math.o
[21:52:05]atoron:/tmp>gcc -o math math.o
math.o: In function ‘main’:
math.o(.text+0x27): undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
[21:52:08]atoron:/tmp>gcc -o math math.o -lm
```

実際に実行してみると，以下のようになります．

```
[21:52:10]atoron:/tmp>./math
セグメンテーション違反です
[21:52:11]atoron:/tmp>./math 100
10
```

ここで math.h を利用しているため，-lm をつけないとエラーがでるのに注意して下さい．数値計算ライブラリというものをリンクしないと，sqrt がどこにも定義されていないためにエラーになります．[3]

-c というオプションは，コンパイルだけしてリンクをしないという物です．その場合は，オブジェクトファイルを出力するので，実行形式にするために，後で再度 gcc でリンク処理をします．gcc のオプションの重要だと思われる物は表 2.2 に示しておきます．さらに複雑なオプションが大量にありますが，それを知りたい場合は，info gcc や，man gcc，gcc --help で調べて下さい．最適化すると，書いた順番と実行される順番が違ったりする事があるので，デバッグが難しくなる可能性があります．

「セグメンテーション違反」というのは，Windows 等で良く出て来る「不正な命令が実行されました」というのと同じようなものです．gdb というデバッガでチェックしてみると，パラメータを与えられていない場合に，atof(av[1]) が NULL であるために，エラーが発生しているという事がわかります．

```
[22:50:05]atoron:/tmp>gcc math.c -o math -g -lm
[22:50:07]atoron:/tmp>gdb ./math
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) run
Starting program: /tmp/./math

Program received signal SIGSEGV, Segmentation fault.
0x4006f77c in __strtod_internal () from /lib/libc.so.6
(gdb) back
#0  0x4006f77c in __strtod_internal () from /lib/libc.so.6
#1  0x40066b12 in atof () from /lib/libc.so.6
#2  0x804849d in main (ac=1, av=0xbffff82c) at math.c:7
#3  0x4005616b in __libc_start_main () from /lib/libc.so.6
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

[3]printf などは，libc で定義されており，-lc は自動的につくので省略して良い．

Table 2.2　gcc のオプション

オプション	意味	解説
-c	compile-only	コンパイルだけをして，リンクをしません．実行できるようになるには，後で出力されたファイルを gcc を利用してリンクする必要があります．
-o	output	出力バイナリファイルの名前を設定します．設定しないとデフォルトでは `a.out` になります
-l	link library	ライブラリをリンクする．`-lm` 等
-g	gdm-compliant	デバッグようのシンボル表をつくる
-O	optimization	最適化オプションを設定．`-O2` で多くの最適化オプションがオンになります．glibc は`-O99` でコンパイルされています．
-f	flags	微妙なフラグを与えます．`-funroll-loops` 等があります．
-S	aSsembler	機械語 (ニーモニック) の出力を *filename*.s というファイルに出力
-Wall	warning	コンパイル時に可能な限り警告を出す
-funroll-loops		コンパイルした時点でループする回数が分かっている場合にループを展開して繰り返し命令のところを，実行される命令をループの回数だけ繰り返して出力する．
-ffast-math		IEEE と ANSI の規則に反した数値計算ルーチンの高速化を行う．(計算結果が正しくならない場合があるので，気を付けて利用すること)

通常は，`-Wall` をつけてコンパイルするのが良い習慣だと言われています．`-Wall` を付けると，文法エラーではないが C としてあまり良くない構文であると思われる物が検出されます．

2.3　g++でのコンパイル

C++のプログラムをコンパイルする場合は，g++を利用します．利用方法は，gcc と同様です．

まず，`vi hello.cc`[4]と入力して vi 起動して，C++のプログラムを書いてみます．

```
#include <iostream>


int main (int ac, char ** av)

{
  cout << "Hello world\n" ;
}
```

このプログラムをコンパイルして実行してみます．

[4]C++の拡張子は cc です．

```
[00:13:26]atoron:/tmp>g++ hello.cc -o hello
[00:18:50]atoron:/tmp>./hello
Hello world
```

2.4 gcc の最適化

gcc で最適化するオプションの意味について具体例で解説します．プログラムとしては，

```
#include <stdio.h>
int main (int ac, char ** av)
{
  int  i;
  int  j = 0;

  for (i = 0 ; i < 10; ++i)
    printf ("%i\n", j+=i);
  return 0;
}
```

を利用します．これを，gcc を-O なし，-O1，-O2 を与えた場合にわけ，それぞれを-funroll-loops を与えた場合とそうでない場合に分けます．それぞれのオプションの効果を出力された機械語[5]の出力の行数を表 2.3 に示します．

最適化をするというように指定しないと-funroll-loops は効力を発揮していないのが分かります．全く最適化していない場合は，以下のようになります．

[5] -S で，アセンブラの出力を出させた

```
  .file "one.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "%i\n"
.text
.align 4
.globl main
.type  main,@function
main:
pushl %ebp
movl %esp,%ebp
subl $24,%esp
movl $0,-8(%ebp)
movl $0,-4(%ebp)
.p2align 4,,7
.L3:
cmpl $9,-4(%ebp)
jle .L6
jmp .L4
.p2align 4,,7
.L6:
addl $-8,%esp
movl -8(%ebp),%eax
movl -4(%ebp),%edx
addl %edx,%eax
movl %eax,%edx
movl %edx,-8(%ebp)
pushl %edx
pushl $.LC0
call printf
addl $16,%esp
.L5:
incl -4(%ebp)
jmp .L3
.p2align 4,,7
.L4:
xorl %eax,%eax
jmp .L2
.p2align 4,,7
.L2:
leave
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.4 20010319 (Debian prerelease)"
```

そして，-O2 で最適化した場合は，以下のようになります.

```
.file "one.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "%i\n"
.text
.align 4
.globl main
.type  main,@function
main:
        pushl %ebp
movl %esp,%ebp
subl $16,%esp
pushl %esi
pushl %ebx
xorl %esi,%esi
xorl %ebx,%ebx
.p2align 4,,7
.L21:
addl $-8,%esp
addl %ebx,%esi
pushl %esi
pushl $.LC0
call printf
addl $16,%esp
incl %ebx
cmpl $9,%ebx
jle .L21
xorl %eax,%eax
leal -24(%ebp),%esp
        popl %ebx
popl %esi
leave
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.4 20010319 (Debian prerelease)"
```

そして，-O2 と-funroll-loops を指定した時には以下のようになります.

Table 2.3　最適化オプションを変化させた時の機械語の出力

	-funroll-loops なし	-funroll-loops あり
-O 無し	47	47
-O1	38	69
-O2	38	69

```
.file "one.c"
.version "01.01"
gcc2_compiled.:
.section .rodata
.LC0:
.string "%i\n"
.text
.align 4
.globl main
.type  main,@function
main:
pushl %ebp
movl %esp,%ebp
subl $8,%esp
addl $-8,%esp
pushl $0
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $1
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $3
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $6
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $10
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $15
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $21
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $28
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $36
pushl $.LC0
call printf
addl $16,%esp
addl $-8,%esp
pushl $45
pushl $.LC0
call printf
xorl %eax,%eax
leave
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.95.4 20010319 (Debian prerelease)"
```

この結果を見るとわかるように，`-funroll-loops` は，繰り返しをそのまま展開していると言う事が分かります．具体的には，

```
addl $16,%esp
addl $-8,%esp
pushl $21
pushl $.LC0
call printf
```

が繰り返しあります．これは，ループの処理よりも，こちらの方が高速に実行される可能性があるからです．しかしながら，常に高速に実行去れると言う事ではないので，十分検証してから使用するようにして下さい．

第3章　各種コマンドの説明

3.1　find の使い方

3.1.1　機能

ファイルの検索

3.1.2　使用方法

```
$find dir [options...]
```

find 開始ディレクトリ…　検索条件…　アクション

3.1.3　説明

指定されたディレクトリ (dir) の下にあるディレクトリを、再帰的に検索し、 オプションに一致するパターンのファイルを 捜し出すコマンドです.

ディレクトリの下にある、ひとつひとつのファイル・ディレクトリについて、 options を左から順番に調べていきます.

3.1.4　オプション

検索条件は次のオプションで指定します.

-name	名前からファイル検索
-size	サイズからファイル検索
-mtime	作成時刻からファイル検索
-atime	アクセス時刻からファイル検索
-ctime	ファイル属性変更時刻から検ファイル索
-inum	i ノード番号からファイル検索
-user	ユーザ名からファイル検索
-nouser	ユーザ名が存在しないファイルを検索
!	条件の論理式　（否定）
-o	条件の論理式　（和）
( )	条件の論理式　（グループ化）

3.1.5　find の使用例

先程作成した、math.c を名前から検索してみます.

```
$find . -mtime math.c -print
```

math.c を時刻で検索してみます.

```
$find . -mtime l -ls
```

math.c を時刻で検索してみます.

```
$ find ~ -perm 644 -ls
```

3.2 grep の使い方

3.2.1 機能

文字列の検索

3.2.2 使用方法

```
$grep [option] pattern [files ... ]
```

3.2.3 説明

入力データの中から、特定の文字列 (pattan) を行単位で検索するコマンドである。
ファイル名 (files) が省略された時は、標準入力からの入力になる。
pattern の部分には正規表現を使用できる。

3.2.4 オプション

-i	pattern の大文字・小文字を区別しない (abc を探すと ABC、Abc、AbC なども表示される)
-l	pattern を含んだファイル名のみの表示となる
-v	pattern を含まない行を検索する
-c	表示した総行数を表示する
-n	行番号をつけて表示する

3.2.5 正規表現の補足

*	前の文字の 0 回以上の繰り返し
.	任意の 1 文字
[str]	str 内の文字リストの人文字に一致
[^ str]	str 内の文字リスト以外の人文字に一致
^	行頭に一致
$	行末に一致

3.2.6 使用例

'include' という単語を含む行を、math.c というファイルの中から探したい場合は、以下のようにします

```
$grep 'include' math.c
```

'printf' で始まり';' で終る文字列

```
$ grep -n 'printf'.*';' math.c
```

行の先頭から int となっている文字列

```
$grep ^int math.c
```

3.3 正しい tar の使い方

3.3.1 機能

tar コマンドは、テープやファイルに複数のファイルをまとめる (アーカイブする) ためのコマンドである.

テープにファイルをバックアップしておくためによく使用されるが、 その他にも、すぐには使わないファイルを一時的にまとめて gzip などで圧縮しておくとディスクの節約にもなる.

3.3.2 使用方法

```
tar key [directory] [tapefile] [ name ... ]
```

3.3.3 オプション

key の部分に使用されるのは以下のものがある.

実際は、この他にも多数のオプションがあるが、詳細は man tape で調べること.

c	アーカイブを新たに作る
x	アーカイブからファイルを取り出す
t	アーカイブの中のファイルの一覧を出力する
f	アーカイブを書き込むファイル tapefile の指定
v	実行過程の表示

3.3.4 使用例

ディレクトリ test の下にあるディレクトリ、ファイルを全部 backup.tar にバックアップ。

```
$ tar cvf backup.tar test
```

ここでディレクトリ test を削除してみます.

```
$rm -r test
```

ファイル backup.tar の中からファイルを全て取り出す.

```
$ tar xvf backup.tar
```

test ディレクトリの下にあるファイルを test.tar というファイルにアーカイブして、gzip で圧縮する.

```
$ tar cvf - test | gzip > test.tar.gz
```

上の圧縮されたアーカイブからファイルを取り出す.

```
$ zcat test.tar.gz | tar xvf -
```

3.3.5 便利な検索方法（find & grep）

find と grep を組み合わせることで、ディレクトリ階層を探索してファイル検索を 行えます

```
find . -type f -name "*" -print | xargs grep -n キーワード /dev/null
```

ファイルのみを対象としたいときは find のオプションで -type f とします。 -exec を使って grep を起動することもできますが、ファイルが見つかる度に grep が 起動されることになり速度が低下します。そこで、xargs を利用して見つかった ファイルを一度に grep へ渡すことで高速化します。grep に/dev/null を渡している のは、検索対象ファイルが 1 つのみの場合は grep 出力結果にファイル名が含まれない からで、擬似的に/dev/null という空のファイルを指定することでそれを回避して います.

3.3.6 使用例

c 言語ファイルの'int' という文字列をさがせ！

```
$find . -type f -name '*.c' -print | xargs grep -n 'int' /dev/null
```

3.4 正しい man の使い方

3.4.1 説明

man は UNIX においてテキストベースのオンラインマニュアルを見るためのコマンドである..
UNIX では、使用方法のわからないコマンドがあったら、まず man コマンドで調べる.
多くのマニュアルは日本語で表示されるが、一部英語のものもある.

3.4.2 使用方法

```
man [-f] [-k] [セクション] コマンド名またはキーワード ...
```

3.4.3 オプション

-f	コマンドの使い方だけ表示する
-k	キーワードで探す

コンソールの 1 画面より長いマニュアルを表示する場合は Space を押すと次の画面に進む.

3.4.4 使用例

cal というコマンドを man コマンドで調べてみよう.

```
$ man cal
```

名称

cal - カレンダを表示する

書式

cal [-jy] [[month] year]

解説

cal は簡単なカレンダを表示します. もし引数が指定されなかった場合は今月のものを表示します。オプションは以下

-j　　ユリウス歴で表示します（日にちは 1 月 1 日からの日数です).

-y　　今年のカレンダを表示します。

パラメータを 1 つだけ指定した場合には、その数値の年（1 - 9999）のカレンダを表示します。ただし、年は完全な

1 年は 1 月 1 日から始まります。

グレゴリオ暦への切り替えは 1752 年の 9 月 3 日に行われたと仮定しています。この時にほとんどの国が切り替えを

使用例

以下に cal の利用例を示します。

```
$ cal
```

```
      April 1995
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 29 21 22
23 24 25 26 27 28 29
30
```

```
 $ cal -j
```

```
            April 1995
  S   M  Tu   W  Th   F   S
                          91
 92  93  94  95  96  97  98
 99 100 101 102 103 104 105
106 107 108 109 110 111 112
113 114 115 116 117 118 119
120
```

次に man コマンドを man コマンドの-f オプションで調べてみよう.

```
$ man -f man

man (1)              - an interface to the on-line reference manuals
man (7)              - macros to format man pages
```

man という同じ名前で 2 種類表示されているが最初のものが上で見た man コマンドです.

2 番目は同じ名前でも違う種類のものです. (マニュアルを表示する時に使用するマクロというもの) (1)、(7) などの番号はセクションというコマンド・キーワードの種類を表すものです. もし (7) の方のマニュアルを表示したい時は以下のように実行する.

```
$ man 7 man
```

次に date というキーワードで-k オプションを使って関係のあるコマンドを一覧表示してみると以下のようになります.

```
$ man -k date

localtime (3)        - transform binary date and time to ASCII
gmtime (3)           - transform binary date and time to ASCII
netdate (8)          - set date and time by ARPA Internet RFC 868
822-date (1)         - Print date and time in RFC822 format
ftime (3)            - return date and time
mktime (3)           - transform binary date and time to ASCII
date (1)             - print or set the system date and time
asctime (3)          - transform binary date and time to ASCII
ctime (3)            - transform binary date and time to ASCII
strftime (3)         - format date and time
rdate (8)            - set the system's date from a remote host
```