計算機実験ハンドブック

2017年度

東京大学理学部物理学科

目次

第 1 章 はじめに 3

第 2 章 UNIX 入門 5

2.1 UNIX のコマンド 5
2.2 リモートログインとファイル転送 15
2.3 Emacs を使う 16
2.4 Gnuplot を使う 18

第 3 章 C 言語入門 23

3.1 C 言語の基礎知識 23
3.2 制御文 28
3.3 配列 32
3.4 文字列と標準入力 34
3.5 ポインタ 35
3.6 関数 38
3.7 構造体 41
3.8 ファイルの取り扱い 42
3.9 その他の制御文 44
3.10 コマンドライン引数の受け渡し 46
3.11 動的な配列の確保: `malloc` と `free` 49
3.12 練習問題の解答例 51

第 4 章 LaTeX 入門 57

4.1 LaTeX の実行 57
4.2 LaTeX に関する全体的なこと 58
4.3 フォント 59
4.4 論文のスタイル 60
4.5 箇条書き 62
4.6 表の作成 64
4.7 数式 65
4.8 図の貼り込み 71
4.9 参照と参考文献 72

第 5 章 バージョン管理システム 77

5.1 バージョン管理システムとは? 77
5.2 diff と patch 77
5.3 主なバージョン管理システム 79
5.4 Subversion リポジトリ 80
5.5 Subversion 実習 80

第 1 章

はじめに

理学部物理学科 3 年の講義「計算機実験 I」および「計算機実験 II」は、理論・実験を問わず、学部〜大学院〜で必要とされる現代的かつ普遍的な計算機の素養を身につけることを目標としている。基礎的な数値計算アルゴリズムとその応用を学習するだけでなく、実習を通じて、計算機の操作、C 言語を用いたプログラミング技術、LaTeX による科学論文の作成技術などを学ぶ。

本冊子は、「計算機実験 I」および「計算機実験 II」の時間に、より効果的に実習を進めることができるよう、必要とされる技術的な点をまとめたものである。大いに役立てて欲しい。また、間違いや不明な点などがある場合には、担当教員、あるいはティーチングアシスタントまでぜひ知らせてほしい。

実習は主として教育用計算機システム ECCS の端末 (iMac) を用いるが、iMac へのログイン方法や基本的な操作方法、Web やメールの利用方法などについては、本書では触れていない。教育用計算機システムの「利用の手引」(http://www.ecc.u-tokyo.ac.jp/guide/current/) に詳しい解説があるので、そちらを参照のこと。

第 2 章

UNIX 入門

この章では、UNIX と呼ばれる OS (オペレーティングシステム) の操作法について学ぶ。通常、Mac OS X や Windows では、マウスを使って画面を操作する。実は、Mac OS X の内部では Darwin と呼ばれる UNIX 系の OS が動作しており、「ターミナル」アプリケーションを立ち上げることにより、UNIX の様々な機能に直接アクセスすることができる。また、ワークステーションやスーパーコンピュータなど、より大きな規模の計算を行うことのできる計算機も、ほぼ全て Linux などの UNIX 系 OS である。

UNIX 系の OS の特徴としては、シンプル、柔軟かつオープンであり、ネットワークに強いことが挙げられる。シェルやスクリプト言語を組み合わせることにより、簡単なコマンドで複雑な処理を実現することができる。また、ネットワーク経由で使用することが前提となっており、実際に計算機がどこにあるかを意識することなしに、透過的に利用できる。UNIX を取得することにより、目の前の PC だけでなく世界中の計算機を利用することが可能となり、計算能力が一気に増えることになる。

2.1　UNIX のコマンド

この節での操作はすべて、ターミナル (「シェル」とも呼ぶ) 内でコマンドを打ち込み、最後にリターンキーを押すことより実行する。確実に操作できるまで、繰り返し練習してほしい。下線を引いた部分がキーボードからの入力である。行頭の “$” はプロンプトと呼ばれる。UNIX のシェルがユーザからのコマンド入力待ちの状態であることを示すものであり、コマンド入力時にタイプする必要はない。

2.1.1　ファイルの操作

この節では、ファイルを作ったり、消したりといった基本的な操作を紹介する。ファイルというのは計算機が情報を書き込むための一つの単位である。ファイルには名前を付けることができて、その名前を指定することでファイルを編集したり消したりできる。

ファイルの基本操作

ファイルをコピーするには cp コマンド (copy の略) を用いる。例えば、コマンドラインで

```
$ cp /usr/local/example/circle.c circle.c
```

を実行すると、/usr/local/example という名前のディレクトリにあるファイル circle.c が、カレントディレクトリにコピーされる。/usr/local/example/circle.c の部分がコピー元に対応し、後の circle.c がコピー先を意味する。コピー先がディレクトリの場合には、コピー先のディレクトリにコピー元と同じ名前のファイルが作られる。したがって、上の例は、カレントディレクトリを表す “.” を用いて、

```
$ cp /usr/local/example/circle.c .
```

としても同じ結果となる。元とは異なる名前でコピーしたい場合には、

```
$ cp /usr/local/example/circle.c oval.c
```

などとする。同じ名前のファイルがすでに存在する場合には上書きされるので注意せよ。

ここで、ls コマンド (list の略) を実行すると

```
$ ls
Desktop    circle.c
```

と表示され、正しくコピーされたことが分かる。このように、ls コマンドを使うことで、どのようなファイルがあるのかを知ることができる。さらに、ls -l を実行すると

```
$ ls -l
total 2
drwxr-xr-x  2 s001500 student      512 Apr 16 03:40 Desktop
-rw-r--r--  1 s001500 student      372 Apr 16 03:43 circle.c
```

のように、より詳細な情報[*1]を得ることができる。この-l をオプション (あるいはコマンドラインオプション) と呼ぶ。ls コマンドには他にも様々なオプションが用意されている。-F オプションをつければ、ファイルの名前の後にファイルの種別を表す文字をつけてくれる。ディレクトリには / が、実行可能ファイルには * がつく。-R オプションをつければ、カレントディレクトリ以下のすべてのディレクトリの中身を見ることができる。また、-a とすると、通常表示されない . (ピリオド) から始まるファイルも見ることができるようになる。

次にファイルの名前を変えてみよう。先の circle.c を ring.c に名前を変更するには、mv コマンド (**mo**ve の略) を使う。

```
$ mv circle.c ring.c
```

ls コマンドで実際に名前が変更されたか確かめてみよう。

続いて rm コマンド (**re**mo**ve** の略) を用いてファイルを消してみよう。

```
$ rm ring.c
```

一度消したファイルは二度と復旧できないので慎重に実行しなければならない。自信がないなら、

```
$ rm -i ring.c
```

のように-i オプションをつけるとよい。ファイルごとに消してよいかどうかの確認をしてくれるので安心である。

さて、ここまでディレクトリという言葉が何度か出てきた。UNIX では、ディレクトリという特殊なファイルを使って、たくさんのファイルを tree 状に管理することができる。ディレクトリとは書類をまとめる書類箱だと思えばよいだろう。ときに大きな書類箱のなかに小さな書類箱が入っていることもあるわけで、ディレクトリの中にディレクトリがあってもいっこうに構わない。

[*1] 表示の 1 行目は、左端の d が、そのファイルがディレクトリであることを示し、次の rwx はファイルの所有者が読み取り可能 (r)、書き込み可能 (w)、実行可能 (x) であることを示している。(ディレクトリが読み取り可能であるとは、そのディレクトリにどのようなファイルが入っているかを ls コマンドで調べられることをいい、ディレクトリが書き込み可能とは、そのディレクトリに新しくファイルやディレクトリを新しく作ったり、すでにあるファイルなどを消したりできることをいう。さらに、ディレクトリが実行可能であるとは、そのディレクトリ内部のファイルを操作できたり、そのディレクトリに移動するとか、そのディレクトリの名前をパスに含めることができることを意味する。) 次の r-x は同じグループ (学生はすべて同じグループに属する) に所属する人が、読み取りと実行可能であることを示し、その次の r-x はそのほかすべての人が読み取りと実行可能であることを示している。隣のカラムの s001500 は、そのファイルの所有者が s001500 であることを示し、次のカラムは所有グループを示している。次の 512 は、そのファイルの大きさが 512 byte であることを示している。Apr 16 03:40 は、そのファイルに最後に変更を加えた日時を示す。そして最後がファイルの名前である。

ログインして最初にいるディレクトリのことをホームディレクトリと呼ぶ。今いるディレクトリ[*2]がどこかを知るには、pwd コマンド (print working directory の略) を使う。/home/s001500 と表示されれば ルートディレクトリ[*3]の下の home というディレクトリの下の、s001500 というディレクトリにいるということが分かる。

試しに sample という ディレクトリを作ってみよう[*4]。

```
$ mkdir sample
```

これで sample というディレクトリが作成された。ls コマンドで確かめてみよう。次に、今作ったディレクトリに移動してみよう。

```
$ cd sample
```

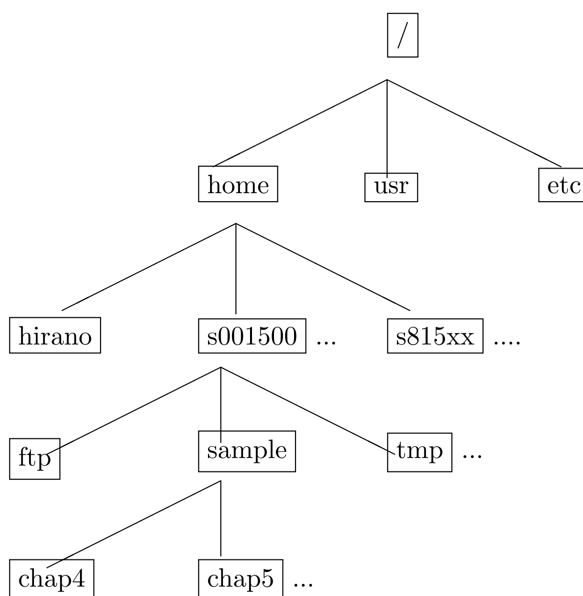
cd は change directory の略である。ここで、先ほどの cp コマンドを使えば、このディレクトリにファイルをコピーすることができる。さらにこの中でmkdir chap4と打てば、sample ディレクトリの下に chap4 ディレクトリが作成される。

では、元のディレクトリに戻るにはどうすればよいのだろうか? それには

```
$ cd ..
```

を実行する。cd と .. の間に空白があることに注意すること。

ツリー構造の中で、どのようにディレクトリやファイルを指定すればよいのだろうか。ディレクトリのツリーを図示すれば次のようになる。



このツリーの中の位置は、一番上の / で表されるルートディレクトリから順にたどることより指定することができる。これを「絶対パス」と呼ぶ。あるいは、今いるディレクトリ (カレントディレクトリ) からの相対的な位置で指定する方法もある。これを「相対パス」と呼ぶ。例えば、ホームディレクトリの下の sample の下の

[*2] カレントディレクトリ、あるいはワーキングディレクトリと呼ぶ。

[*3] ルートディレクトリとは一番上のディレクトリのことであり、最初の/がルートディレクトリを表す。

[*4] すでにあるディレクトリを消すには rmdir コマンド (remove directory の略) を使う。rmdir でディレクトリを消すときはそのディレクトリの中にファイルやディレクトリが 1 つもない状態になっていなくてはならない。消したいディレクトリの中にファイルが残っているときは rm コマンドを使い、ディレクトリが残っているときは rmdir コマンドを使う。残っているファイルやディレクトリをいちいち消すのが面倒な場合は rm -r *directory* とすれば、*directory* の中に何が残っていてもすべてきれいに消し去ってくれる。

chap4 ディレクトリは、絶対的な指定では /home/s001500/sample/chap4 と書かれるが、カレントディレクトリがホームディレクトリの場合、相対的な指定では sample/chap4 と書かれる。そのディレクトリの中にあるファイルも /home/s001500/sample/chap4/circle.c とか sample/chap4/circle.c のように指定することができる。特に一つ上のディレクトリは “..” で、カレントディレクトリは “.” であらわされる。つまり先ほどの sample/chap4/circle.c は./../s001500/sample/chap4/circle.c と書いても同じものを指す。

これで、自由にディレクトリ間を移動することができるようになった。例えば、すでに使った ls コマンドは、/usr/bin あるいは/bin ディレクトリに格納されている*5。cd /usr/bin として、そこに移動してから lsと打ってみよう。たくさんのファイルが置かれているが、ls というファイルは見付かっただろうか*6。

なお、cdとだけ打てば、いつでも自分のホームディレクトリに戻ってくることができる。ホームディレクトリは ~ とも表されるので cd ~としても同じである。

ファイルの中を見る

さて、肝心のファイルの中身を見るにはどうしたらよいのだろう。ファイルの中身を見るにはいくつかの方法がある。ファイルには大別して、中身を見ることができるテキストファイルと中身を見られないバイナリファイルがある。テキストファイルの中身は、cat コマンドで見ることができる*7。

```
$ cat circle.c
```

cat は最も手軽にファイルの中身をのぞくことのできるコマンドである。中身が長すぎる場合、cat では流れていってしまうが、

```
$ more circle.c
```

と打つことにより、ちょうどよい所で一旦止めてくれる。--More--(xx%) のような表示が左下に見えたら、スペースを叩いてみよう。次のページが表示されるはずである。hと打つと簡単なヘルプが見られる。bか ctrl-B で 1 ページ戻る。また Enter で 1 行進む。さらに、/の後に**文字列**を打つことでファイル中の文字列を検索することができる。検索された文字列を含む行はウインドウの一番上に表示される。nと打てばもう一度同じ文字列を探してくれる。また、10dのように数字を入れてからdを打てば、その数字分だけ進む。more から抜けるには、qを入力する。

more と似ているが、もう少し使い勝手のよい less というコマンドもある。less では上向きにスクロールすることができる。jで一行下に、kで一行上*8に進む。more と同じく、スペースバー、h、b、d、/、n、qのようなコマンドも使える。更に gでファイルの先頭に、Gでファイルの最後尾にジャンプすることができる*9。

次に、たくさんあるファイルの中に特定の文字列が含まれているか調べてみよう。

```
$ grep main *.c
```

*5 which lsとすることで、ls がどこにあるか調べることができる。

6 例えば、treasure.here という名前のファイルを探したい場合、そのファイルがありそうなディレクトリよりも上のディレクトリに行って、find . -name treasure.here -printとする。find の直後の . (ピリオド) は「そのファイルがありそうなディレクトリより上のディレクトリのパス」を指定する。またファイル名があやふやな場合でも、find . -name 'tre.here' -printのように * を任意の文字列として合致するファイルを表示してくれる。find はとても多機能である。例えばfind . -name 'tre*.here' -exec cat {} \;のように、探し出したファイルに対してコマンドを実行することもできる。-exec 以下の書式は コマンドの引数となるファイル名が {} となるように書き、最後に \; を加えればよい。-exec の代わりに -ok を使えばコマンドを実行する前に一々聞いて来るので rm などを find で実行したいときには安心である。find . -ctime -1 -printのように-ctime で最終変更期日が 1 日前以内のものだけ表示させるようなことも可能である。-ctime の引数を +1 にすると最終変更期日が 1 日以上前のものだけが表示される。

*7 バイナリのファイルの中身を覗くための od (octal dump) というコマンドもある。

*8 編集用のソフト vi のキー操作と同じ。

*9 tail circle.cのようにすれば、ファイルの最後だけ見ることができる。また tail -30のように指定することで最後から 30 行目以降を見ることができる。

と入力してみよう。どの“.c”で終るファイルに main という文字列が含まれているか一目瞭然である[*10]。

ファイルの保護 (1)

他の人に見られたくないファイルにはプロテクションをかけることができる。

```
$ chmod o-r circle.c
```

とすれば自分と、自分と同じグループに属する人以外はそのファイルを見ることができなくなる[*11]。

```
$ chmod g-r circle.c
```

とすれば同じグループの人からも見られなくなる。同じ要領で

```
$ chmod o-rwx circle.c
```

とすれば他の人は `circle.c` というファイルを読むことも書き込むことも実行することもできなくなり、

```
$ chmod go-rwx sample
```

とすれば自分以外のだれも `sample` というディレクトリにアクセスできなくなる。もし大切なファイルを間違いなどで変更したくない場合は、

```
$ chmod u-w circle.c
```

のようにすれば、自分自身の書き込みからもファイルを守ることができる[*12]。

ファイルの保護 (2)

ファイルにプロテクションをかける方法がわかったところで、セキュリティの面から見たファイルの保護について説明しておこう。まず、特別な事情がないかぎり、ファイルに user 以外の書き込み許可を出してはならない。次に user 以外には読み取り許可 (および実行許可) も出してはいけないディレクトリやファイルとしては、`.Xauthority`、`.ssh` などがある[*13]。

ファイルのアクセス許可について不安がある場合は、`.bashrc` などのファイルに`umask 077`の一行を入れておくとよい。こうしておくと、それ以後のログインで作られるファイルは、user 以外のアクセス許可がいっさいない状態で作られる。その後、許可を出してもよいと判断したファイルに限って、`chmod` コマンドで許可を出すようにするとよいだろう。

2.1.2 オンラインマニュアル

ここまで様々なコマンドを説明してきたが、これらのコマンドのマニュアルは `man` コマンドを使って、オンライン (計算機上) で見ることができる。まずはマニュアルのマニュアルを見てみよう。

```
$ man man
```

[*10] さらに`grep -n printf *.c`としてみると、各ファイルの何行目に printf という文字列があるのかが分かる。さらに、`grep -n -e '.*gram.*' *.c`で、`.*` のところに任意の文字列があてはまるすべての文字列について検索する。シングルクォーテーションで囲まなくてはいけない点に注意すること。

[*11] o は other の頭文字である。反対に他の人にも見えるようにするには `o-r` を`o+r` に変えて実行する。`chmod u=rwx,g+x,o-r circle.c`のように指定することもできる。そのほか `g` (group) は同じグループ、`u` (user) は所有者、`a` (all) はユーザーも含めた全員を表し、`r` 、`w` 、`x` はそれぞれ、read, write, execute を表す。

[*12] また変更したくなった場合は `chmod u+w circle.c` とすればよい。

[*13] “.” で始まるファイルは隠しファイルである。ホームディレクトリで `ls -a` とするとこれらのファイルを見ることができる。

と打つと、Reformatting page. Wait... とでた後、画面が切り替わって、man というコマンドのマニュアルが表示される。この画面を表示しているのは、すでに説明した less*14なので、1 ページ前に行ったり、スキップしたりするのも容易である。

ここまでに紹介して来た多くのコマンド*15がマニュアルに登録されている。

```
$ man command
```

のようにして、それぞれのコマンドの詳しい意味を調べてみよ。

正しいコマンド名が分かっているときはよいが、もし「〜のようなコマンドはないかな？」とか「たしかこんなコマンドがあったはず」と思ったときは、

```
$ man -k keyword
```

を使う。例えば、なにかディレクトリの操作に関するコマンドに関して調べるには、以下のようにすればよい。

```
$ man -k directory
...
mkdir (1)              - Makes a directory
mkdir (2)              - Creates a directory
mkdirhier (1X)         - makes a directory hierarchy
mkfontdir (1X)         - create fonts.dir file from directory of font files
mklost+found (8)       - Makes a lost\(plfound directory for fsck
mvdir (1)              - Moves (renames) a directory
pwd (1)                - Displays the pathname of the current (working) directory
rename (2)             -  Renames a directory or a file within a file system
rmdir (1)              - Removes a directory
rmdir (2)              - Removes a directory file
...
```

左端がマニュアルのタイトルである。右欄にある大雑把なコマンドの意味を参考に目的のものを探し出す。コマンド名の後に書いてある括弧の数字はマニュアルのセクション番号をあらわす。マニュアルは使う目的や内容に応じてセクションに分かれている。もし同じタイトルのマニュアルが二つのセクションに分かれて置かれている場合、それぞれは

```
$ man 2 rmdir
```

のように man コマンドの後にセクション番号を書くことで指定することができる。

2.1.3 プロセスの管理とトラブル時の対策

この節では、ここではトラブル時の対策を含んだプロセスの管理について説明する。多少面倒な話なので読み飛ばしておいても構わない。

ターミナル上で何かを実行していて止まらなくなったときに最も有効なのは、ctrl-c である。ctrl-c は実行しているジョブを強制的に終了する。うまくいけばプロンプトが戻って来るはずである。

それでも止まらなかった場合には、ctrl-z を入力してみよう。ctrl-z はプロセスを一時的に停止させるだけなので、その後で終了させるか続行させるか、いずれかを選ぶ必要がある。もしプロンプトが戻って来たら、jobs と入力する。おおよそ

---

*14 あるいは、環境変数 PAGER を設定している場合にはそのページャーが立ち上がる。

*15 コマンドに限らず C 言語用のライブラリ関数や設定ファイルの書式などもマニュアルに入っている。

```
$ jobs
[1]    Running                    emacs local-guide.tex
[2]+   Suspended                  a.out
```

のような出力が得られるはずである。悪さをしているのが、a.out なら、行頭の [ ] のなかの数字である 2 に % をつけた %2 を用いて

```
$ kill %2
```

として、ジョブを止める。もし、正常に走っているが単に時間がかかっているだけだとわかっているのであれば、

```
$ bg %2
```

のように打ってバックグラウンドで実行してもよいだろう[*16]。

さて、ctrl-c でも ctrl-z でもジョブが終了できないときには、別のターミナルを開き、(必要に応じて SSH ログインした後、) ps xと打つ。すると、

```
$ ps x

  PID TT STAT  TIME COMMAND
27515 p0 IW    0:03 -bash
27767 p0 TW    0:00 emacs
28123 p3 R     0:24 ./a.out
28529 p3 R     0:00 ps
```

のように表示される。左からプロセスの ID 番号、制御端末、プロセスの状態を示す略号、現在までの CPU 時間、そして実行中のコマンドが表示されている。このなかで悪さをしていそうなコマンドを探す。今の場合 ./a.out が怪しいので、

```
$ kill 28123
```

として、そのプロセスを終了する。 28123 は ./a.out のプロセスの ID 番号である。ちゃんとプロンプトが帰ってくれば一件落着である。それでもだめなときは、

```
$ kill -HUP 28123
```

としてみる。まだだめな場合は -HUP を -QUIT [*17]にしてみよう。それでもうまくいかない場合は -QUIT を -KILL にするが、-KILL は最後の手段と考えて、無闇に使わないほうが無難である。

2.1.4 パイプとリダイレクション

UNIX の設計思想の一つに小さなツールを組み合わせるということがある。いまから紹介するのがその小さなツールを組み合わせるためのコマンドの書き方である。

[*16] 最初からバックグラウンドで実行したいのであれば、コマンド実行時に最後に &をつければよい。例えばa.out &のように実行する。一方バックグラウンドで走っているジョブをフォアグラウンドに戻すには、fg %2などとする。

[*17] -QUIT を使うとプロセス (この場合は a.out) が実行中だったディレクトリに core というとても大きなファイルができることがある。通常は必要ないので消してしまってかまわない。

パイプ

例えば、ps コマンドの出力の中から、emacs のプロセス ID を調べようとするとき、ps コマンドと文字列を探し出す grep コマンドを組み合わせて、

```
$ ps | grep emacs
```

のように実行する。“|” はパイプと呼ばれ、パイプの左側のコマンドが出力するデータをパイプの後ろのコマンドの入力に繋いでくれる。ここで使えるのは、画面に結果を書き出すコマンドとキーボードから入力を読み込むコマンド[*18]の組み合わせだけであることに注意せよ。

パイプを使えば、ls などの出力で流れて行ってしまうものを、

```
$ ls -laF | more
```

のようにして、一時的なファイルを作る手間なしに、more を使ってゆっくり眺めることができる。

リダイレクション

次は、リダイレクションである。リダイレクションとは、通常画面に書かれる内容を、かわってファイルに書き出したり、通常キーボードから読まれる入力をファイルから読み込むようにする機能である。これを用いれば、コマンドの実行結果をファイルに保存したりすることも可能である。例えば、

```
$ ls -laF > ls.dat
```

のようにすれば、ls -laF の結果が ls.dat に書き出される。また、C のプログラムでの計算結果をいったんファイルに落とし、gnuplot (2.4節) で図にプロットするというといった作業も行うことができる。

2.1.5 コマンドヒストリとコマンドライン補完

コマンドヒストリとコマンドライン補完は、なれてしまうと手放せなくなる機能である。コマンドヒストリとは、一度実行したコマンドを呼び出す機能のことである。これを使えば、同じコマンドを実行するたびに最初から打つ手間が省ける。また、以前実行したコマンドをちょっと変えて実行するということも簡単にできる。以前に実行したコマンドを呼び出すためには、プロンプトの所で ctrl-p を押すか、キーボードの上矢印を押す。プロンプトの後にコマンドが表示されるはずである。ここで Enter すれば、そのまま実行される。あるいは ctrl-b (左矢印) や ctrl-f (右矢印)[*19]や Backspace あるいは Delete キーを使って編集してから実行することもできる。ctrl-n (下矢印) を押せば逆にヒストリを戻ってくることができる。また、

```
$ !string
```

とすれば、*string*[*20]から始まる一番最近のコマンドを実行してくれる。さらに ctrl-r に続けて過去に打ったコマンドを頭から入力していくと、コマンドヒストリの中から逐次候補が表示される (インクリメンタルサーチ)。大昔に打ったコマンドを捜し出すのに便利である[*21]。

```
$ history
```

---

*18 grep のマニュアルを見れば分かるが、grep は入力ファイルが指定されずに起動された場合は、標準入力 (ふつうはキーボード) を探すと書いてある。

*19 これらの矢印のついたキーのことをカーソルキーと呼ぶ。

*20 *string* は例である。もちろんどのような文字列でも構わない。

*21 実際のところ、!*string*では、そのままコマンドが実行されてしまうので、ctrl-r の方が便利である。

と打つと、これまでのコマンド一覧が表示される。その数は標準では 500 になっている*22。
次は、コマンドライン補完である。例えば

```
$ ls -aF

./          excellent-bitmaps/  work/
../         save/
archive/    TeX/
emacs.doc   temp.out$\ast$
```

のような状況で次のようなコマンドを打ちたいとする。

```
$ chmod o-rx excellent-bitmaps
```

をいちいち最初から最後まで打っていたのでは、面倒である。そこでまず、

```
$ chm
```

まで打った所で、左の方にある Tab キーを押してみよう。すると、

```
$ chmod ▌
```

のように変わる。つまり、自動的にコマンドの一部を補完してくれるのである。さらにこの機能はコマンドだけではなく、ファイルやディレクトリの名前、ユーザー名にも使える。さっきの例では

```
$ chmod o-rx ex
```

まで打った時点で、Tab *23を押せば、自動的に

```
$ chmod o-rx excellent-bitmaps
```

と補完される。これらの機能を利用すれば、これまで以上に効率よく UNIX とコミュニケーションを取れるはずである。

2.1.6 その他有用なコマンド

他によく使うコマンドを紹介しておこう。

⇒ **who**
いま使っている計算機にだれがログインしているかを表示する。

⇒ **clear**
ターミナル画面で、画面をクリアする。

⇒ **date**
現在の日付と時間を表示する。

⇒ **cal**
今月のカレンダーを表示する。`cal 2017`とすれば今年のカレンダーを表示する。

⇒ **du**

---

*22 初期設定ファイルの `.bashrc` に `export HISTSIZE=1000` と書いておくと 最大 1000 個までの過去のコマンド履歴が記録される。また、次にログインしたときにも前回ログインした時のヒストリが残されている。
*23 Tab でなく、ctrl - d を押せば、補完可能な候補の一覧が表示される。

あるディレクトリ以下について、ディレクトリごとにファイルの大きさの総和を取って表示する。du -s .として使用することで、カレントディレクトリ以下のファイルの大きさの総和を表示する。

⇒ **tar**

ファイルをまとめて一つにまとめる。**t**ape **ar**chiver の略である。ファイルを転送したり、バックアップを取るときに利用する。

```
$ tar cvf ../archive.tar .
```

のように使うと、../archive.tar というファイル[*24]にカレントディレクトリ以下のすべてのファイルのバックアップが取られる。. の代わりに *.c を使えば カレントディレクトリの .c で終るファイルがまとめられる。また、. の部分にディレクトリを指定すれば、そのディレクトリ以下のファイルすべてのバックアップを取ることができる。反対に展開するときは、

```
$ tar xvf archive.tar
```

のようにする。

⇒ gzip

ファイルを圧縮してサイズを小さくする。

```
$ gzip origin
```

のように実行すると、*origin* というファイルが消えて、*origin*.gz というファイル[*25]が作成される。解凍[*26]するには、

```
$ gzip -d oringine.gz
```

あるいは

```
$ gunzip origin.gz
```

のように実行する[*27]。しばしば tar と組み合わせて使われる。

⇒ **ln**

ファイルのリンク[*28]を作成する。

```
$ ln filename linkname
```

のように実行する[*29]と *filename* というファイルは *linkname* という名前でもアクセスできるようになる。これをリンク、とくにこの場合はハードリンクと呼ぶ。ハードリンクがコピーと違うのは、*filename* の内容を変更した場合、自動的に *linkname* の内容も変更される[*30]ことと、ディスクの使用量がハードリンクを作っても (ほとんど) かわらないことである。また誤って *filename* か *linkname* のどちらかを

[*24] ファイル名の最後に .tar を付ける習慣にしておくと、あとで混乱が少なくなる。
[*25] ファイルの最後に .gz が付く。逆にもし、ファイルの最後が .gz になっているファイルがあれば、それは gzip で圧縮されていると思ってよい。
[*26] 圧縮を元に戻すことである。
[*27] tar + gzip ファイル (ファイル名の末尾が.tar.gz になっているか、.tgz になっている場合が多い) を解凍・展開するときには、gzip -cd archive.tar.gz | tar xvf -のようにパイプをうまく使うと、中間ファイルの archive.tar を生成せずにいきなりディレクトリに展開してくれる。また、tar zxvf archive.tar.gzでも同じ結果となる。逆に、tar zcvf archive.tar.gz . で、圧縮されたアーカイブを直接作成することができる。
[*28] コンパイラのところで登場する「リンク」という言葉とは関係ない。
[*29] cp と同じ順序で引数を指定する。
[*30] ハードリンクを作ると *filename* と *linkname* の区別はなくなり、両者は対等の関係になる。

消してしまっても一方は残っているので、バックアップの役にも立つ。しかし、ハードリンクは「自分のホームディレクトリ内」のように限られた空間[*31]内でしか利用できない。

ハードリンクに対して、シンボリックリンクというものもある。シンボリックリンクは

```
$ ln -s filename linkname
```

のようにして作成する[*32]。シンボリックリンクもハードリンクと同じく、*linkname* によって *filename* にアクセスすることが可能になる。しかし、 *filename* を消したり、*filename* の名前を変更したりすると、もはや *linkname* でファイルにアクセスすることはできなくなってしまう。シンボリックリンクは、同一パーティション内にないファイルに対しても作成することができる。

⇒ **time**

コマンドの実行時間を計測する。例えば

```
$ time gzip origin
```

のように、コマンドの前に `time` コマンドをつけて実行すると、コマンドの実行にかかった実時間 (real), ユーザ CPU 時間 (user)、システム CPU 時間 (sys) が表示される。

2.2 リモートログインとファイル転送

ある計算機から別の計算機へネットワーク経由でログインし作業するには、`ssh` (secure **sh**ell の略) あるいは `slogin` コマンドを利用する[*33]。例えば、ホスト名が remote.phys.s.u-tokyo.ac.jp というマシンにログインしたい場合、

```
$ ssh -X remote.phys.s.u-tokyo.ac.jp
```

あるいは

```
$ slogin -X remote.phys.s.u-tokyo.ac.jp
```

と入力する[*34]。現在ログインしているマシン (ログイン元) と接続しようとしているマシン (ログイン先) のアカウント名が異なるときは、

```
$ ssh -X remote.phys.s.u-tokyo.ac.jp -l username
```

あるいは

```
$ slogin -X username@remote.phys.s.u-tokyo.ac.jp
```

とする。*username* はログイン先のアカウント名である。次にパスワードを入力すると、remote.phys.s.u-tokyo.ac.jp に接続され、プロンプトが表示される。ただし、初めて接続するマシンの場合、次のようなメッセージが出力される。

```
Host key not found from the list of known hosts.
```

[*31] 同一パーティション内のこと。

[*32] シンボリックリンクを作ることを、「リンクを張る」といういいかたをすることがある。

[*33] 同様の機能を持つものに `telnet`、`rlogin` というコマンドがあるが、パスワードが平文でネットワーク上を流れるなど、セキュリティー上の問題があるので、今日では使われない。

[*34] `-X` オプションはリモートマシン上で実行する Emacs、gnuplot、evince などのウィンドウを手元のマシンの画面で開くためのもの (X11 forwarding) である。

```
Are you sure you want to continue connecting (yes/no)?
```

ここで、yes と答えると、パスワードの入力に進むことができる。ログイン後、入力する命令は、すべてリモートホスト上で実行される。最後に

```
$ exit
```

と入力すると、接続が解除され、元のマシンのプロンプトに戻る。

ある計算機から別の計算機にファイルをコピーしたい場合には、scp (secure copy の略) コマンドを用いる。手元のマシンからリモートのマシン (例: remote.phys.s.u-tokyo.ac.jp) へファイル (report.pdf) を送る場合は、

```
$ scp report.pdf username@remote.phys.s.u-tokyo.ac.jp:~
```

とする。最後の (チルダ) はリモートマシンのホームディレクトリ (/home/*username*) を表す。コピー元のファイル名や、コピー先のディレクトリ名は適宜変更すること。また、.pdf という拡張子のつくファイルをすべて送りたい場合は、

```
$ scp *.pdf username@remote.phys.s.u-tokyo.ac.jp:~
```

とすればよい。report というディレクトリを送りたい場合は、

```
$ scp -r report username@remote.phys.s.u-tokyo.ac.jp:~
```

とする。逆に、リモートマシンのファイルをこちらへ取ってくる場合は、

```
$ scp username@remote.phys.s.u-tokyo.ac.jp:~/report.pdf .
```

などとする。最後の . (ドット) はカレントディレクトリを表す。

2.3 Emacs を使う

C 言語のプログラムや LaTeX のソースコードなど、テキスト形式のファイルの編集には、エディタと呼ばれるソフトを用いる。UNIX で代表的なものとしては、vi や Emacs がある。本節では Emacs の使い方を紹介しよう。

まず、Emacs を立ち上げるには

```
$ emacs
```

と入力すればよい。あるいは、

```
$ emacs &
```

としておけば、Emacs を別ウィンドウで立ち上げた後、元のターミナル内で別の作業を続けることができる。

2.3.1 チュートリアル

Emacs のコマンドの多くは ctrl-なんとか と Esc-なんとか[*35]にキー定義されている。

Emacs のチュートリアル自体も、ctrl-h T に割り当てられている。ctrl-h T を押すときは、まず ctrl-h を押してコントロールキーを離してから T を押す。ここで T は大文字なので Shift キーと同時

[*35] Esc は Control と違い、一旦 Esc を押して離した後に、他のキーを押す。

に押すことを忘れないこと。また、本来のコマンド名 Esc-x help-with-tutorial と打ってももちろんかまわない。以下、キーバインドの後にカッコ中に太字で書いてあるのがコマンドで、すべてのコマンドは Esc-x コマンド名 と打つことでも、実行が可能である。このチュートリアルは基本的な事柄を網羅しているので、載っているキー定義を覚えてしまえば相当 Emacs が使いやすくなるはずである。

Emacs を使っていて何か困ったと思ったときは、まず ctrl-g (**keyboard-quit**) を押してみよう。それでもだめなら ctrl-x ctrl-c (**save-buffers-kill-emacs**) で Emacs から抜け出ればよい。

2.3.2 編集作業

それでは Emacs の中から、何かファイルを編集してみよう。Emacs が立ち上がった状態からファイルを編集し始めるには、ctrl-x ctrl-f (**find-file**) とする。

```
Find file:  ~/
```

と聞いてくるので、既存のファイルを編集したいのならそのファイル名を、もし新たにファイルを作って編集を始めたいのならそのファイル名を入力する[*36]。ここでは、ためしに `trial.c` というファイルを編集してみよう。

普通のキーは打てばそのままキートップに書いてある文字が挿入される。何か間違えた時には Backspace あるいは Delete キーで消す。カーソルキーを使ってカーソルを移動することも可能である。カーソルキーを使わないで、ctrl-p (**previous-line**)、ctrl-n (**next-line**)、ctrl-f (**forward-char**)、ctrl-b (**backward-char**) でも構わない。

Emacs はファイルの名前から判断して、これから編集するのが C のプログラムだと認識する。Emacs のウインドウの下にあるモードライン (反転表示されている一行のこと) に C の文字が見えるはずである。

C のプログラムは、通常次のようにブロックをインデントする。

```
int main() {
  return 0;
}
```

インデントを行うのに、いちいちその数だけスペースを打つのは面倒である。Emacs は C のファイルを編集していると認識すると、Tab を打つことで適当な場所までインデントを行ってくれる。

何か文章やプログラムを Emacs 上で書いたとしよう。編集したファイルを保存するには、ctrl-x ctrl-s (**save-buffer**) を使う。

```
Wrote /home/s001500/trial.c
```

のような表示が Emacs のミニバッファに出るはずである。Emacs の隣のターミナル画面で `ls` コマンドを実行して `trial.c` というファイルが新たに生成されていることを確かめよう。さらに、`cat` とか `more` のようなコマンドでファイルの中身を確かめてみるのもよい。

さて、Emacs での作業も終ったので、Emacs から抜け出すことにしよう。Emacs から抜けるには、ctrl-x ctrl-c (**save-buffers-kill-emacs**) とする。

2.3.3 Emacs の様々なコマンド

Emacs には実に様々なコマンドがあるが、その中でも使用頻度の高いコマンドをまとめておく。コマンドは、Esc-x コマンドと打つことでも実行できる。

[*36] ここで読み込まれたファイルはバッファと呼ばれる一時的に確保される記憶領域に置かれる。編集の操作はすべて、バッファに対して行われるので、ファイルに書き出さないと編集の結果は残らない。

キー定義	コマンド	動作
ctrl-x ctrl-f	find-file	新たにバッファを作り、そのバッファに読み込むファイルの名前を尋ねて読み込む。
ctrl-x i	insert-file	現在のカーソルの位置に指定したファイルを挿入する。
ctrl-x ctrl-c	save-buffers-kill-emacs	Emacs を終了する。保存されていないバッファは保存するかどうか尋ねてくる。
ctrl-x ctrl-s	save-buffer	バッファの内容をファイルに書き出す。
ctrl-x ctrl-w	write-file	バッファの内容を書き出すファイルの名前を尋ねてから書き出す。
ctrl-g	keyboard-quit	コマンドの入力を中断する。
ctrl-v	scroll-up	画面一枚分上にスクロールする。
Esc-v	scroll-down	画面一枚分下にスクロールする。
ctrl-s	isearch-forward	下向きに文字列を検索する。検索モードから抜けるには、Esc を打つ。
ctrl-r	isearch-backward	上向きに文字列を検索する。検索モードから抜けるには、Esc を打つ。
Esc-%	query-replace	文字列の置換を行う。スペースで実行、“n” で不実行。 ! で合致するすべてを置換する。
ctrl-k	kill-line	現在いる行のカーソルより後ろを削除する。
ctrl-スペース	set-mark-command	マークを設定する。
ctrl-w	kill-region	マークから現在のカーソルの位置までを削除する。
ctrl-y	yank	カーソルの位置に削除した内容を戻す。
ctrl-x u	advertised-undo	アンドゥ、直前の操作を取り消す。
Esc-<	beginning-of-buffer	文章の始めに飛ぶ。
Esc->	end-of-buffer	文章の終りに飛ぶ。
ctrl-x 2	split-window-vertically	ウインドウを二つに分割する。
ctrl-x 1	delete-other-windows	今カーソルのいるウインドウを残し、他のウインドウを消す。
ctrl-x o	other-window	ウインドウが分割されているとき、違うウインドウに飛ぶ。
ctrl-x 5 2	make-frame	新たにフレームを作る。
ctrl-x 5 0	delete-frame	現在のフレームを消す。
ctrl-x b	switch-to-buffer	他のバッファにスウィッチする。
ctrl-x ctrl-b	list-buffers	現在のバッファ一覧を出す。
ctrl-l	recenter	ウインドウをリフレッシュし、カーソル行をウインドウ中央に移動する。
ctrl-\	toggle-egg-mode	かな変換モードを on/off する。
ctrl-x (	start-kbd-macro	キーボードマクロの定義を開始する。
ctrl-x)	end-kbd-macro	キーボードマクロの定義を終了する。
ctrl-x e	call-last-kbd-macro	一番最近定義したキーボードマクロを実行する。

ある領域を削除、移動、コピーしたいときには領域の先端でマークし、領域の終端までカーソルを移動した後、ctrl-wを押して領域を削除する。移動したいときには、さらに移動先までカーソルを持って行き、そこでctrl-yを押す。コピーしたいときには、削除した直後にctrl-yを押して復旧し、さらにコピー先までカーソルを持って行って、ctrl-yを押してコピーする。つまり一旦ctrl-wでためておいてから[*37]、ctrl-yであちこちにコピーする。また、ctrl-kを連続して使って消した場合も、ctrl-yでまとめてペーストすることができる。そのほか便利な機能としては、Esc-x `goto-line` で指定行に飛ぶ等が挙げられる。

2.4 Gnuplot を使う

計算結果をグラフに変換するツールはいろいろあるが、UNIX で最もよく使われているものの一つに `gnuplot` がある。この節では、`gnuplot` の基本的な使い方を解説する。

ターミナルで

```
$ gnuplot
```

と入力すると、`gnuplot` が起動し、プロンプトが表示される。

[*37] 実はEsc-wだと削除しないでためるだけなので、こちらの方が楽である。

```
gnuplot>
```

終了するには、

```
gnuplot> exit
```

と入力する。次に、データをプロットしてみよう。data.txt という名前で、中身が

1.0 2.0
2.0 4.0
3.0 6.0
4.0 8.0
5.0 10.0

のようなファイルを用意し、

```
gnuplot> plot "data.txt"
```

と入力するとプロットされる。 データファイルには 1 行に「横軸の値と縦軸の値」を入れる。グラフの右上にデータセット名 (デフォルトではファイル名) を示す”data.txt”が表示される。これは任意の名前あるいは非表示に変更できる。

点が線でつながったプロットを行うには、

```
gnuplot> plot "data.txt" with line
```

とする。これらを応用すると、次のようなことができる。

```
gnuplot> plot "data.txt" with line, "data2.txt"
```

では、data の点は線でつながり、data2 は点のみでプロットされる。(data2.txt も自分で用意すること。)

```
gnuplot> plot "data.txt" with line, "data2.txt" with line
```

だとどちらも線でつながる。また、常に、点ではなくつながった線で表示させるには、

```
gnuplot> set data style line
gnuplot> plot "data.txt", "data2.txt"
```

とする。

直前のプロットを再プロットするには、

```
gnuplot> replot
```

とする。

x 軸を log スケールにするには、

```
gnuplot> set log x
```

x 軸をリニアスケールにする (戻す) には、

```
gnuplot> unset log x
```

y 軸を log スケールにするためには、

```
gnuplot> set log y
```

などとする。

前述したグラフの右上にあるデータ名 (「レジェンド」と呼ばれる) を変更するには、

```
gnuplot> plot "data.txt" title "Title of data"
```

とする。また、レジェンドを非表示にするためには、

```
gnuplot> unset key
```

とする。x 軸にラベルを付けるためには、

```
gnuplot> set xlabel "x axis"
```

グラフそのものに表題を付けるには、

```
gnuplot> set title "Title of this plot"
```

とする。

gnuplot で標準に用意された関数を表示するには、

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

とする。これらを応用すると、次のようなこともできる。

```
gnuplot> f(x) = sin(x)
gnuplot> g(x) = A*cos(x)*exp(x)
gnuplot> A = 10.0
gnuplot> plot "data.txt", f(x), g(x)
```

関数の値のサンプリングのピッチを変えるには、

```
gnuplot> set sample 10000
```

とする。

x 軸のプロットの範囲を変えるには、

```
gnuplot> set xrange [0:10]
```

とする。y 軸も同様である。

図をポストスクリプト形式でファイルに出力するには、次のようにする。

```
gnuplot> set term postscript eps color
gnuplot> set out "output.eps"
gnuplot> replot
```

直前のプロットの内容がポストスクリプト形式で output.eps に出力される。もちろん、最後の行は、

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

のようにしてもよい。

その他の機能については、`gnuplot` の help を参照してほしい。help は

```
gnuplot> help
```

で表示することができる。

第 3 章
C 言語入門

プログラミング言語とは、コンピューターに仕事をさせるために、その作業内容を指示するための特別な言語である。この言語で書かれた作業の手順書を「プログラム」とよび、プログラムを作成することを「プログラミング」という。

プログラミング言語には、さまざまな種類がある。C 言語はもともと、UNIX (マルチユーザ、マルチタスクのオペレーティングシステム) を記述するためのシステム言語として開発された。現在は、システムソフトウエアの作成だけでなく、事務処理や科学技術計算、アプリケーションソフトウエア (表計算やワープロなど) の開発など、広く汎用プログラミング言語として使用されている。

3.1 C 言語の基礎知識

3.1.1 まずはコンパイル

ここでは、最も短い “コンパイル可能な C のプログラム” を作成し、それをコンパイルしてみよう。

■プログラムの構成

C のプログラムの最小単位は関数である。“C のプログラムは関数を並べたものである” ということもできる。関数を定義するときは以下のように書く。

型 関数名 (引数) { 処理 }

関数名 関数の名前は自由につけられる。ここで、C でプログラム中に使う名前 (関数以外でも) は

⇒ 先頭は英字または _ (下線)

⇒ 2 文字以降は英数字または _ (下線)

⇒ 大文字/小文字は区別される

引数 関数には引数 (パラメタ) を渡すことができる。(具体的な例については 3.6節参照)

型 関数は何かを処理して最後に値を返す (関数値)。この関数値の種類を示す。

処理 この中で、1) 関数が行う処理内容を書き、2) 最後に関数値を返す。関数値を返すには

`return` **関数値**;

と書く。

プログラム中に関数は複数作ることができるが、必ず `main` という関数が必要である。(すべての処理は `main` から始まる。) 以上をふまえて、C の関数の例を示す。

例 3.1.1.

```
int main() { return 0; }
```

■コンパイル

Emacs などのテキストエディタを使って例 3.1.1.の 1 行を入力したテキストファイルを作成し、3.1.1.c という

名前で保存する。このプログラムをコンパイルするには gcc を用いる。

```
$ gcc 3.1.1.c
```

同一ディレクトリに a.out というファイルができているので、以下のように実行する。

```
$ ./a.out
```

(ただし、現段階のプログラムでは、実行してももちろん何も起こらない。)

3.1.2 書式の慣例

先に進む前に、C のプログラムの書式の慣例について、いくつか説明する。

■自由書式 (free format)
C 言語では、1 つの行に複数の文を書いたり、または 1 つの文を複数の行に書くことができる (自由書式)。文の終わりは、記号 ; で判別する。

1 つの行に 2 つの文を書く

```
aaaaa; bbbbbbb;
```

1 つの文を 2 つの行にわけて書く

```
aaaaaaaa
aaaaaaa;
```

例えば、例 3.1.1.は、以下のようにも書くことができる。

例 3.1.2.

```
int main() {
  return 0;
}
```

本書では、例 3.1.2.のような書式を用いることにする。スペース節約のため空行は入れないが、実際のコードでは適宜空行を挟むことでプログラムの可読性が向上する。

■インデントの慣例
C の文は、意味上のレベル (入れ子構造の階層) を持つ。意味上の各レベルをわかりやすくするため、より深いレベルを右側に段付けして書くのがよい (インデント)。

```
aaaaa           /* ← 最上位のレベルの文 */
aaaaa
  bbbbb         /* ← 1 つ深いレベルの文 */
  bbbbb
    ccccc       /* ← もう 1 つ深いレベルの文 */
    ccccc
  bbbbb
aaaaa
```

例 3.1.2.の関数は、すでにインデントして書かれている。インデントは上の行から行毎に順にキーボード上の Tab を押せばできる。だたし、Emacs の場合は c-mode になっている必要がある。c-mode になっている場合は、Emacs の下の部分に (C) と表示されている。c-mode にするには、Esc を押して (離し)、x を押して (離し)、c-mode と入力して Enter を押せばよい。

■コメント

/* と */ で囲った部分はコメントとみなされる (複数行も可)。例えば、例 3.1.2.にコメントを挿入してみると、

例 3.1.3.

```
int main() {
  /* これもコメント。
     複数行にわたっても OK。 */
  return 0;
}
```

/* と */ の間にある部分はコンパイルするときは無視される。

3.1.3 コンパイル (2)

次に、“何かがおこる” プログラムをコンパイルしてみよう。

例 3.1.4.

```
#include <stdio.h>
int main() {
  printf("> %lf\n", 10.0);
  return 0;
}
```

例 3.1.4.のプログラムを Emacs などで作成し、3.1.4.c という名前で保存する。ここで、printf は、文字を出力する標準ライブラリ関数である。この例では “> 10.000000” を出力する。

それでは、保存したプログラムを、gcc を用いてコンパイルし、実行してみよう。

```
$ gcc 3.1.4.c
$ ./a.out
> 10.000000
```

コンパイル時にエラーで止まってしまう場合には、ソースコードをもう一度見直すこと。

また、次のようにすれば、a.out ではなく好きな名前の実行ファイルが作成できる。

```
$ gcc -o ten 3.1.4.c
```

この場合は、同一ディレクトリに ten というファイルができる。以下のようにすれば実行できる。

```
$ ./ten
> 10.000000
```

ここで、C 言語で使う変数の型を説明しておくことにする。主な変数の型としては以下のようなものがある。

int	整数型
double	倍精度浮動小数点数型
char	文字型
unsigned int	符号なし整数型
float	単精度浮動小数点数型

double でもよい場合は float を使う利点はない。各型の変数で表現できる値の範囲は、以下のようになっている。

int	$-2^{31} \sim 2^{31}-1$
unsigned int	$0 \sim 2^{32}-1$
double	0.0 と $\pm(4.94065645841246544 \times 10^{-324} \sim 1.79769313486231570 \times 10^{+308})$
float	0.0 と $\pm(1.40129846432481707 \times 10^{-45} \sim 3.40282346638528860 \times 10^{+38})$

ただし、値の範囲は処理系に依存する。2015 年現在、int は 32bit が主流となっている。

3.1.4 分割コンパイル

一つのソースコードが大きくなり過ぎると、何かと不便である。たとえば、大きなファイルの中から目的の編集箇所を自分で探さなくてはいけなくなったり、コンパイルに時間がかかったりする。そのようなときはソースコードをいくつかのファイルにわけるとよいだろう。また、ソースコードがそれほど大きくなくても、意味としてまとまりのある部分ごとに 1 つのファイルを作っておくとあとあとのためにもよい。プログラムを複数のファイルに分割して書いた場合、

```
$ gcc trial1.c trial2.c trial3.c
```

のように書いてコンパイルする。この場合も -o オプションで出力ファイルを指定することができる。また、それぞれのファイルを独立にコンパイル[*1]することもできる。

```
$ gcc -c trial1.c
$ gcc -c trial2.c
$ gcc -c trial3.c
```

のようにすれば、trial1.c, trial2.c, trial3.c がそれぞれ独立にコンパイルされ、trial1.o, trial2.o, trial3.o の 3 つのオブジェクトファイル[*2]が生成される。あるいは、

```
$ gcc -c trial1.c trial2.c trial3.c
```

としても同じである。オブジェクトファイルができたら、

```
$ gcc -o trial trial1.o trial2.o trial3.o
```

のように最後に一つにまとめる (「リンクする」という)。分割コンパイルの最大の利点は、コンパイル時間の短縮である。trial3.c だけを変更したのに、3 つのファイル trial1.c trial2.c trial3.c 全てをわざわざコンパイルするのでは時間がかかってしまう。分割コンパイルでは、変更のあったものだけ再コンパイルすればよい。つまり、

```
$ gcc -c trial3.o
$ gcc -o trial trial1.o trial2.o trial3.o
```

だけでよいということになる[*3]。

[*1] 分割コンパイルと呼ぶ。
[*2] コンパイルが途中まで完了しているファイルだと考えてよい。
[*3] make というツールを使うと、変更のあったもの (およびそれに依存するファイル) だけを自動的に再コンパイルしてくれる。

3.1.5　C 以外の言語のコンパイル

プログラム言語には C 以外にもさまざまなものがある。物理の世界でも、最近は C や C++ を使用することが一般的になってきたが、歴史的には Fortran が長く使われてきた。そのため、いまだに Fortran で書かれたライブラリも広く使われている。Fortran で書かれたソースコード (拡張子 .f, .F, .f90, .F90 など) をコンパイルするには、gcc の代わりに gfortran を用いる。C++ の場合 (拡張子 .C, cpp など) の場合には、g++ を用いる。分割コンパイルやリンクの方法は C の場合と同じである。

C, C++, Fortran などの言語は「コンパイル言語」と呼ばれる。一方、コンパイルして計算機の言葉にするのではなく、ソフトウェアがプログラムを直接理解してその内容を実行するプログラム言語 (スクリプト言語) もある。この「プログラムを直接理解するソフトウェア」のことをインタープリタといい、「プログラム」のことをスクリプトという。sh、perl、ruby、python などが代表的なインタープリタである。たとえば、test.py という Python スクリプトを作ったとしよう。これを実行するには 2 通りの方法がある。1 つ目は、

```
$ python test.py
```

のようにインタープリタ[*4]に test.py を読み取らせる方法である。もう 1 つの方法は

```
$ test.py
```

のように test.py を直接実行[*5]する方法である。後者の場合、test.py に読み取り許可だけでなく実行許可も必要である。

3.1.6　ライブラリのリンク

プログラムの中で $\sin x$ や $\cos x$ などの数学関数を用いた場合は、前述の方法ではコンパイルに失敗する。まずは、例 3.1.5. のプログラムを作成し、3.1.5.c という名前で保存しよう。

例 3.1.5.

```
#include <stdio.h>
#include <math.h>
/* test sign of cosine values */
int main() {
  int i = 0;
  while (i <= 180) {
    double angle;
    double cosval;
    angle = i*M_PI/180.0;
    cosval = cos(angle);
    printf("cos(%lf) is %lf\n", angle, cosval);
    i = i + 20;
  }
  return 0;
}
```

このプログラムでは math.h 内で宣言されている数学関数を使っているため、libm というライブラリが必要となる。コンパイルは次のように行う。

[*4] python が Python 言語のインタープリタである。
[*5] 実はこっそり Python インタープリタが起動されていて、それが test.py を読み取り実行する。どのインタープリタが起動されるかは、スクリプトの先頭の #! のあとに何を書いたかで決まる。Python インタープリタを起動する場合は、スクリプトの先頭に#!/usr/bin/python と書けばよい。

```
$ gcc 3.1.5.c -lm
```

`-lm` という部分は、`m` (libm のうち lib を削除した残り) を `-l` というオプション引数を使って `gcc` に渡している。例えば、libsocket を使いたい場合は、`-lsocket` と指定する。math.h の中には sin や cos だけでなく、例 3.1.5.で使われている π (=3.1415...) も `M_PI` として定義されている。例 3.1.5.では、`+` や `*` が使われているが、これらは、加減乗除を行う演算子である。算術演算子には次のようなものがある。

加算	+
減算	-
乗算	*
除算	/
剰余	%

例 3.1.5.だけでなく例 3.1.4.でも出てきた、`printf` という関数 (**print** with **f**ormatting の略) は画面に文字や数字を表示させる場合に用いられる。

`printf("`**フォーマット**`",` **変数** `1,` **変数** `2,......)`

という形で使用する。例 3.1.5.の場合は、フォーマットが "`cos(%lf) is %lf \n`" になっている。前者の `%lf` の部分が変数 1 (`angle`) の値に置きかえられ、後者の `%lf` の部分が変数 2 (`cosval`) の値に置きかえられて表示される。`\n` は改行を表す。`%lf` の他に `%d`, `%s`, `%f` などがあり、それぞれ変数の型によって使い分ける。

%lf	double
%f	float
%d	int
%s	char* 文字列
%c	char 文字

3.2 制御文

3.2.1 if 文

「... ならば... を実行して、それ以外なら... を実行する」という内容のプログラムは `if` 〜 `else` 〜を用いて書く。

例 3.2.1.

```
#include <stdio.h>
int main() {
  int a = 10;
  int b = 20;
  if (a == b) {
    printf("a is equal to b\n");
  } else {
    printf("a is not equal to b\n");
  }
  return 0;
}
```

この例では、a と b が異なるので、実行すると a is not equal to b と表示される。

```
if (A) {
    ブロック 1
} else if (B) {
    ブロック 2
} else {
    ブロック 3
}
```

この一連の if 文は次の順序で動作する。A が正しいならブロック 1 が実行される。(他のブロックは実行されない。) A が間違っていて、B が正しいならブロック 2 が実行される。(他は実行されない。) A が間違っていて、B も間違っていればブロック 3 が実行される。(他は実行されない。)

ここで、比較及び関係演算子を挙げておく。

a と b が等しい	a == b
a と b は等しくない	a != b
a は b より大きい	a > b
a は b より小さい	a < b
a は b 以上	a >= b
a は b 以下	a <= b

また、論理演算子には以下のようなものがある。

a または b	a || b
a かつ b	a && b
a でない	!a

3.2.2 for 文

繰り返しを行うためには、for を用いる。次の例では、1 から 100 までの和を for 文を用いて計算する。

例 3.2.2.

```
#include <stdio.h>
int main() {
  int sum = 0;
  int i;
  for (i = 1; i <= 100; ++i) {
    sum = sum + i;
  }
  printf("sum of integers from 1 to 100 is %d\n", sum);
  return 0;
}
```

以下のような for 文を考える。

```
for (A; B; C) {
  ブロック
}
```

この例では、A がまず評価される。その後、条件 B が満足されていれば、ブロックが実行される。次に C が実行された後、再び B がチェックされ正しければ、ブロックが実行される。結局、A→B が正しい → ブロック →C →B が正しい → ブロック →C→B が正しい → ブロック →... となる。途中で B が間違いになると for 文を終了する。

また、++i は i=i+1 と同じ意味を持っている。この ++ をインクリメント演算子という。同様に、i=i-1 と同じ意味を持つものとして --i がある。これをデクリメント演算子という。この他にも、i++ と i-- という書き方もある。これらの違いは、インクリメントされたりデクリメントされたりするタイミングが異なることにある。もし、

```
int i = 10;
int j = i++;
```

ならば、j の値は 11 ではなく 10 となる。j に代入後、i は 11 になる。ところが、

```
int i = 10;
int j = ++i;
```

ならば、j の値は 10 ではなく 11 となる。つまり、i が 11 になった後、j に代入される。特別な理由がない限り、++i と --i を使うのがよい。

また、i=i+1 と同じ意味を持つ式として、i+=1 という表現も可能である。この +=は加算代入演算子とよばれ、演算子の後に置かれた式の値を演算子の前の変数に加算する。他にも、-= (減算代入演算子)、*= (乗算代入演算子)、/= (除算代入演算子)、%= (剰余代入演算子) がある。加算演算子を用いて、例 3.2.2.の sum=sum+i は sum+=i とも書ける。

3.2.3 while 文

ある条件が満たされている間、何かを繰り返し実行したいときは、while を使う。次の例は、i が 0 より大きい間は while ブロックを実行し、その結果として、1 から 100 までの和を求めている。

例 3.2.3.

```
#include <stdio.h>
int main() {
  int i = 100;
  int sum = 0;
  while (i > 0) {
    sum += i;
    --i;
  }
  printf("sum of integers from 100 to 1 is %d\n", sum);
  return 0;
}
```

while 文は次のように動作する。

```
while (A) {
  ブロック
}
```

A をまずチェックして正しいならば、ブロックを実行する。そして、再び A をチェックして正しいならば、ブロックを実行する。A が正しい間はブロックの実行を繰り返す。A が最初から間違いであればブロックは一度も実行されない。

3.2.4 break 文

for や while のような繰り返しを行う文の中から途中で抜けたい場合には、break を用いる。例 3.2.4.では、cos の値が負になったとき強引に for 文を抜けている。

例 3.2.4.

```
#include <stdio.h>
#include <math.h>
int main() {
  int i;
  for (i = 0; i <= 180; i += 20) {
    double angle = i*M_PI/180.0;
    double cosval = cos(angle);
    printf("cos(%lf) is %lf\n", angle, cosval);
    if (cosval < 0.0) {
      break;
    }
  }
  return 0;
}
```

3.2.5 continue 文

ある条件を満たした場合に for や while の先頭に戻り、次の繰り返しに進みたい場合には、continue を用いる。(より正確には、continue から後を実行せずに、次の条件判定を行う。) 例 3.2.5.では、割り切れない場合には continue を実行してすぐに i<=number の判定を行っている。割り切れる場合は printf(...) を実行してから i<=number の判定を行っている。ちなみに、このプログラムは 80 の全ての約数を出力する。

例 3.2.5.

```
#include <stdio.h>
int main() {
  int number = 80;
  int i;
  for (i = 1; i <= number; ++i) {
    /* 'a % b' yields the residual of a/b */
    int residual = number % i;
    if (residual != 0) {
      continue;
    }
    printf("%d is a measure(YAKUSUU) of %d\n", i, number);
  }
  return 0;
}
```

■補足

ここまで説明をしなかったが、if や for や while の条件の部分で使った "正しい (満足する) 場合" と "間違いの (満足しない) 場合" という表現に関して補足しておく。まず、実例をあげると、例 3.2.5.の if (residual != 0) の部分は、if (residual) と書き換えることができる。つまり、条件判定にとって、"≠0" の整数値は "正しい (真値)" と判断され、"=0" の整数値は "間違い (偽値)" と判断される。簡単な例をあげると、while (1) {}とすると、無限に while の中を繰り返す。もちろん、この場合には break 等の適切

な処理が while の中で必要である。

練習 3.2.1. if 文の練習として、$ax+b=0$ の解を求めるプログラムを書け。a,b,x を表示させて終了させること。また、a,b はあらかじめプログラムの中で決めてよい。

練習 3.2.2. for あるいは while の練習として、$n!$ (階乗) を求めるプログラムを書け。n と結果を表示させて終了させること。また、n はあらかじめプログラムの中で決めてよい。

練習 3.2.3. 2 から 100 までの素数を表示するプログラムを書け。

3.3 配列

3.3.1 1 次元配列

1 次元配列を用いるためには、

```
int a[10];
```

のように宣言する。この場合、

```
a[0], a[1],...,a[9]
```

の 10 個の要素を持つ配列が生成される。添字が 0 から始まっていることに注意せよ。例 3.3.1.では、#define によって N_ELEMENT を 10 と定義している[*6]。for 文によって、配列 array に数字が格納されている。あとはそれを順番に表示して終了する。なお、

```
int n = 10;
int a[n];
```

はコンパイルエラーとなる。配列の大きさを動的 (プログラム実行中) に変化させることはできない[*7]。動的に変化させたい場合は、malloc と free (3.11節) を用いる。

例 3.3.1.

```
#include <stdio.h>
#define N_ELEMENT 10
int main() {
  int i, array[N_ELEMENT];
  for (i = 0; i < N_ELEMENT; ++i) {
    array[i] = i*i*i;
  }
  for (i = 0; i < N_ELEMENT; ++i) {
    printf("array[%d] = %d\n", i, array[i]);
  }
  return 0;
}
```

*6 詳しくは、参考書などの「プリプロセッサ」に関する説明を参照のこと。N_ELEMENT はプログラムの実行中に変化する数ではなく、コンパイル時に定まっている定数である。

*7 1999 年改訂の C 標準規格 C99 では認められているが、本書では ANSI C (C89/90) について説明する。

3.3.2　2 次元配列

2 次元配列は、

```
int a[2][3];
```

のように宣言する。この場合、

```
a[0][0], a[0][1], a[0][2]
a[1][0], a[1][1], a[1][2]
```

の 6 つの要素を持つ 2 次元配列が生成される。例 3.3.2.は、2×2 の行列の逆行列を求めている。

例 3.3.2.

```
#include <stdio.h>
int main() {
  double matrix[2][2];
  double det;
  double inverse[2][2];
  /*  matrix = | 10.0   5.0 |
               |  8.0  14.0 | */
  matrix[0][0] = 10.0;
  matrix[0][1] = 5.0;
  matrix[1][0] = 8.0;
  matrix[1][1] = 14.0;
  det = matrix[0][0]*matrix[1][1] - matrix[0][1]*matrix[1][0];
  if (det == 0.0) {
    printf("inverse matrix does not exist\n");
  } else {
    inverse[0][0] =  matrix[1][1]/det;
    inverse[0][1] = -matrix[0][1]/det;
    inverse[1][0] = -matrix[1][0]/det;
    inverse[1][1] =  matrix[0][0]/det;
    printf("inverse[0][0] = %lf\n", inverse[0][0]);
    printf("inverse[0][1] = %lf\n", inverse[0][1]);
    printf("inverse[1][0] = %lf\n", inverse[1][0]);
    printf("inverse[1][1] = %lf\n", inverse[1][1]);
  }
  return 0;
}
```

練習 3.3.1.　int 型で大きさが 5 の 1 次元配列 a と b を準備し、配列 a にあらかじめ数字を代入しておく。その配列 a の要素をすべて配列 b に代入し、その後、配列 a をすべて 0 にするプログラムを書きなさい。a と b をすべて表示させて終了すること。

練習 3.3.2.　double 型で大きさが 2×2 の 2 次元配列 a と b を準備し、その積を計算するプログラムを書きなさい。a と b とその積を表示させて終了すること。ただし、 a と b はあらかじめプログラムの中で決めてよい。

3.4　文字列と標準入力

文字列の取り扱いは非常に複雑である。ここではごく基本的なことに限定して解説する。また、標準入力 (キーボードなどからの入力) を受け付け、それをプログラムで利用する方法について説明する。

3.4.1　文字列

文字列とは、名前通り文字の配列である。文字列を扱うためには、`char s[10];` のように宣言する。ただし、次のような形の代入はできない。

```
char s[10];
s = "Hello";
```

文字列の宣言と同時に文字を代入するには次のように行う。

```
char s[10] = "Hello";
```

あるいは、

```
char s[] = "Hello";
```

後者は自動的に `[]` の中は 6 になる。(また、前者の 10 は適当な数字でよいが、`Hello` 5 文字を記憶するためには 6 以上の数字を用いる必要がある。) では、なぜ `[]` の中は 6 になるのか? なぜ 5 でないのか? という疑問が生じるだろう。文字列を使う場合、どこでその文字列が終了したかを判断しなければならない。ここで使っている `s` という変数は単にその文字列の先頭を示しているだけで、どこで終わるかという情報を持っていないからである (3.5.3節参照)。そのため、C 言語では文字列の最後に必ず "`\0`" という特殊な文字をつけるという決まりがある。そうすることで、文字列の最後を判断することが可能になる。このような理由で、5 ではなく 6 になるのである。

char s[ ] = "Hello";

H	e	l	l	o	\0
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]

char s[10] = "Hello";

H	e	l	l	o	\0				
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]

3.4.2　標準入力 (1)：gets 関数

例 3.4.1.を作成しコンパイルしてみよう (コンパイラによっては、「`gets` は危険だ」と警告が出るかもしれないが、実行ファイルはちゃんとできているはずである)。これを実行すると、`Input:` と表示され入力を促される。ここで、`uni` などと入力すると、`uni` と表示されてプログラムが終了する。この場合、`gets` という関数によって入力された文字列が `str` にコピーされる。注意すべきことは、`str` が `str[20]` と宣言されているので入力すべき文字列は 19 文字以内に限られるということである。しかし実際には、ユーザーは 20 字以上入力することができる。その場合、はみ出た文字は用意された領域の外に書き込まれる。もしそこが、別の変数用に使われていたら? これはとても危険なことである。`gets` を使う場合は注意が必要である。

例 3.4.1.

```
#include <stdio.h>
int main() {
  char str[20];
  printf("Input: ");
```

```
  gets(str);
  printf("%s\n", str);
  return 0;
}
```

3.4.3 標準入力 (2)：scanf 関数

例 3.4.1.ではたとえ数字 (19 桁以下) を入力してもそれは文字列として扱われるため、数字として足し算などに用いるためには atoi などの関数を用いる必要がある。数字を入力してそのまま数字として扱う方法を例 3.4.2.に示す。この場合、scanf という関数を用いている。(ソースコード中の &i や &x の & は今は気にしなくて良い。)

例 3.4.2.

```
#include <stdio.h>
#include <string.h>
int main() {
  int i;
  double x;
  printf("Input(int): ");
  scanf("%d", &i);
  printf("%d\n", i);
  printf("Input(double): ");
  scanf("%lf", &x);
  printf("%lf\n", x);
  return 0;
}
```

練習 3.4.1. 練習 3.2.2.で、標準入力から n を決定し、その答えを表示するプログラムを書きなさい。

3.5 ポインタ

ポインタの習得は C 言語の習得の中でひとつの大きな壁である。はじめは訳がわからないかもしれないが、C 言語の基本的な部分の 1 つなので、いろいろな例に触れて慣れてほしい。慣れが一番である。

3.5.1 とりあえず (1)

例 3.5.1.を作成し実行してみよう。上手くいけば、“q is 200 and *p is 200.” と表示されるはずである。この例で出てくる p がポインタである。int 型のポインタを宣言するには、

```
int *p;
```

のように*をつけて宣言する。そして、p が int のポインタで、*p が int になる。人によっては、

```
int* p;
```

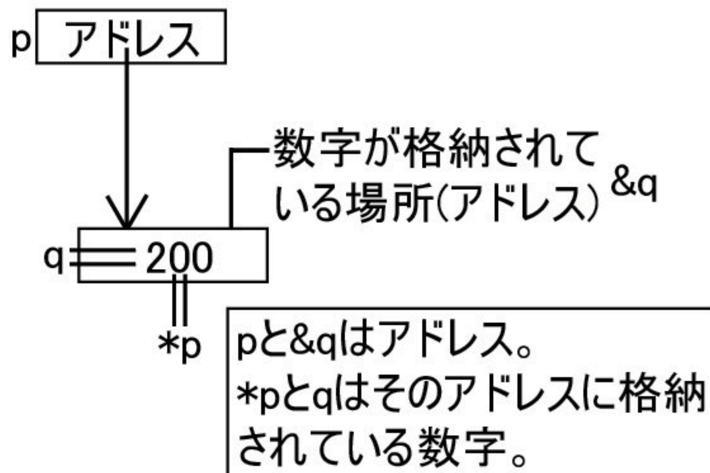
と宣言したほうがイメージしやすいかもしれない。つまり、p は int* 型 (int のポインタ) ということである。(しかし、2 個以上のポインタを宣言するためには、int *p, *q; としなければならない。int* p, q; とすると意味が変わってしまう。)

p という int 型のポインタは、int 型の変数のアドレスを指し示す働きをする。アドレスとは大雑把にいえば、次のようなものである。例 3.5.1.の int q; q = 200; の q に格納されている 200 はコンピュータのメモ

リーのどこかに電気的に存在するはずである。その物の場所を示すものがアドレスである。メモリー上の住所である。したがって、例 3.5.1.では p = &q; の文によって、p は q を指し示すことになる。(&q は q のアドレスを表す。) これにより、q は q 自身だけでなく、p というポインタによってもアクセスすることができるようになる。

例 3.5.1.

```
#include <stdio.h>
int main() {
  int *p;
  int q;
  q = 200;
  p = &q;
  printf("q is %d and *p is %d.\n", q, *p);
  return 0;
}
```



3.5.2　とりあえず (2)

では、もう 1 つ例を示そう。例 3.5.2.では、p = &q; により p というポインタで q にアクセスできる。ここでは、q に値を代入するかわりに、*p を用いて値を代入している。p はポインタで、*p は p が示しているアドレスにある値そのものである。実行結果は、“q is 300 and *p is 300.” となる。

例 3.5.2.

```
#include <stdio.h>
int main() {
  int *p;
  int q;
  p = &q;
  *p = 300;
  printf("q is %d and *p is %d.\n", q, *p);
  return 0;
}
```

記号*, &などの意味と役割をもう一度復習しておこう。

```
int *p;
int q;
```

の時、

p	ポインタ (つまりどこかのアドレス)
*p	実体
q	実体
&q	q のアドレス

注意すべきことは、例 3.5.1.や例 3.5.2.で `p = &q;` がない場合は `p` がどこを指しているか未定義であるということだ。したがって、`*p = 300;` や `printf` 内で `*p` を表示させるのは危険である。何が起こるか分からない。

この節では、静的に宣言された変数を指すためにポインタを用いてきたが、より高度な使い方については 3.11節で紹介する。

3.5.3 ポインタと配列

ポインタと配列には密接な関係がある。

```
int array[10];
```

と宣言した場合、`array[0]` や `array[5]` などで配列の要素にアクセスすることができた。実は、`array` だけでも意味を持つ。`array` は配列型 (`int[10]` 型) の変数で、プログラムコード中に登場した時は、大抵の状況では、自動的に "配列の先頭要素のアドレス" に暗黙的に変換されて解釈される。つまり、`array` は `array[0]` へのポインタとして解釈され、`*array` は `array[0]` を意味する。さらに、ポインタに整数を足すとその数だけ先の要素を指すようになる。例えば array+2 は `array[2]` へのポインタであり、`*(array+2)` は `array[2]` と等価である。

では、例 3.5.3.を見てみよう。`strlen` という関数は string.h 内で宣言されている文字列の長さを返す関数である。最初の `for` 文は文字型の配列 `str` の中身を順番に書き出している。次の `for` 文がポイントである。まず、文字型のポインタ `p` に文字型の配列 `str` の先頭ポインタ `str` を代入している。そして、文字型の実体である `*p` が `\0` でない場合はその `*p` を表示し、ポインタ `p` の指す部分をひとつ進めている。これを `*p` が `\0` になるまで繰り返すことで、最初の `for` 文と同じ結果を表示している。

例 3.5.3.

```
#include <stdio.h>
#include <string.h>
int main() {
  char str[] = "ABCDE";
  int num = strlen(str);
  int i;
  for (i = 0; i < num; ++i) {
    printf("%c", str[i]);
  }
  printf("\n");
  char *p;
  for (p = str; *p != '\0'; ++p) {
    printf("%c", *p);
  }
  printf("\n");
  return 0;
```

```
}
```

練習 3.5.1. 次のプログラムの printf の部分を配列ではなく、ポインタを使ったものに書き換えなさい。(array を使っても、あるいは、int *p; を加えてもよい。)

```
#include <stdio.h>
int main() {
  int i;
  int array[10];
  for (i = 0; i < 10; ++i) {
    array[i] = i*i;
  }
  for (i = 0; i < 10; ++i) {
    printf("array[%d] = %d\n", i, array[i]);
  }
  return 0;
}
```

3.6 関数

3.1節で C 言語の基本構造を説明したが、main(){...} だけでプログラムを書くことはほとんどない。大きなプログラムになればなるほど関数化を行い、役割分担をはっきりさせるのがよい。

3.6.1 あまり良くない例

例 3.6.1.は円の面積の計算するために直接その公式を書いている。しかし、何度も何度も円の面積を計算したいとき、半径を入力すれば面積が返ってくるような関数があれば非常に便利である。

例 3.6.1.

```
#include <math.h>
#include <stdio.h>
int main() {
  double radius = 2.0;
  double area = radius * radius * M_PI;
  printf("Radius: %lf, Area: %lf\n", radius, area);
  return 0;
}
```

3.6.2 関数化

では、例 3.6.1.を例 3.6.2.のように変更してみよう。このような小さなプログラムでは関数化の効力は乏しいが、大きくなればなるほど、また複雑になればなるほど、その効力は絶大になる。例 3.6.2.では、circle_area という関数が定義されている。この関数は、double 型の値を引数として受け取り、面積を計算してその値を戻り値にしている。関数は使う前に宣言する必要がある。したがって、circle_area 関数を main の後や他のファイルに書く場合は、例 3.6.3.のように使う前に宣言だけ行っておく必要がある。

例 3.6.2.

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

```
double circle_area(double r) {
  return r*r*M_PI;
}

int main() {
  double radius = 2.0;
  double area = circle_area(radius);
  printf("Radius: %lf, Area: %lf\n", radius, area);
  return 0;
}
```

例 3.6.3.

```
#include <math.h>
#include <stdio.h>
double circle_area(double);

int main() {
  double radius = 2.0;
  double area = circle_area(radius);
  printf("Radius: %lf, Area: %lf\n", radius, area);
  return 0;
}

double circle_area(double r) {
  return r*r*M_PI;
}
```

3.6.3 ポインタを引数にする関数

例 3.6.2.では関数の返り値が面積の 1 つだけであった。しかし、例えば割り算で商と余りを返したい場合、例 3.6.2.のような関数ではうまく実現できない。なぜなら、戻り値は 1 個しか指定できないからである。これを解決するには、ポインタを引数とする関数を作る必要がある。答えを入れる箱をあらかじめ準備しておき、それを関数の引数にポインタとして渡し、関数内でそこに答えを入れてもらって、関数の外で受け取るという方法を用いる。重要な点は、ポインタでなければ関数内で代入した値を関数の外で使うことはできないということである。

例 3.6.4.では、まず main で答えを入れてもらうための shou と amari という箱 (変数) を準備している。次に、そのポインタ (アドレス) を関数 division に渡している。そして、division 内で答えを代入してもらい、関数の外で printf を用いて答えを表示している。division という関数の先頭にある void という型は (返り値の形では) 何も返さないことを示すためのものである。

例 3.6.4.

```
#include <stdio.h>
void division(int divident, int divisor, int *quotient, int *residual) {
  *quotient = divident / divisor;
  *residual = divident % divisor;
}

int main() {
  int josuu = 3;
  int hi_josuu = 13;
```

```
  int shou, amari;
  division(hi_josuu, josuu, &shou, &amari);
  printf("%d / %d = %d ... %d\n", hi_josuu, josuu, shou, amari);
  return 0;
}
```

ポインタを使う理由がはっきりと分からない場合は、例 3.6.5.を作って実行してみてほしい。間違った答えが表示されるだろう。なぜなら、この場合 division という関数では、quotient と residual という二つの一時変数が生成され、それらに計算結果が代入されるからである。それらの一時変数は関数の処理が終了すると同時に破棄され、値は main の中の shou と amari には引き継がれない。関数に外部から値を渡すだけであれば、例 3.6.5.のような引数の宣言方法で良い (「値渡し」とよぶ) が、内での計算結果を関数の外で使いたい場合は、ポインタを使って変数のアドレスを渡す (「ポインタ渡し」とよぶ) 必要がある。

例 3.6.5.

```
#include <stdio.h>
void division(int divident, int divisor, int quotient, int residual) {
  quotient = divident / divisor;
  residual = divident % divisor;
}

int main() {
  int josuu = 3;
  int hi_josuu = 13;
  int shou, amari;
  division(hi_josuu, josuu, shou, amari);
  printf("%d / %d = %d ... %d\n", hi_josuu, josuu, shou, amari);
  return 0;
}
```

3.6.4 Fortran の関数や手続きの利用

LAPACK[*8]などの多くのライブラリが Fortran で書かれている。これらと同等の機能を持つ関数を C や C++ で作ることもできるが、既に存在するのであればそちらを使った方が便利である。ここでは、C 言語の中から Fortran の関数や手続きを呼ぶ方法を簡単に紹介する。

Fortran の関数や手続きの名前を C 言語から呼ぶためには、その名前をすべて小文字にして、最後に_ (下線) を付けなければならない。また、関数などの引数の型も適切に読み変える必要がある。例えば、

例 3.6.6.

```
      Real*8 CALC(I, X, Y)
      Interger*4 I
      Real*4 X
      Real*8 Y
```

という Fortran の関数が存在するとする。このとき C では次のように使う。

例 3.6.7.

```
extern double clac_(int*, float*, double*);
```

[*8] 行列の対角化、連立一次方程式の求解など線形計算を行うライブラリ。ほぼ全ての計算機で利用可能である。

```
int main() {
  int i = 10;
  float x = 20.0;
  double y = 30.0;
  double ret = calc_(&i, &x, &y);
  retrun 0;
}
```

全ての引数をポインタ渡しとしなければならないことに特に注意せよ。コンパイルは C と Fortran で別々に行い、最後にリンクすればよい。

3.7 構造体

データ構造を取り扱うためには構造体を用いる。

例 3.7.1.を見てほしい。個人のデータを取り扱うために personal_data という 1 つの箱を準備している。これが構造体である。もし構造体を使わなければ、同じ内容のプログラムを実現するためには複数の配列を準備する必要が出てくる。これは見た目にも格好悪いし、拡張性も乏しく複雑になる。例 3.7.1.では、struct personal_data pdata; で pdata を構造体とし宣言している。少し分かりにくいかもしれないが、int n; と比べてみると、int と “struct personal_data” が同じ立場であって、int と “struct” が同じ立場というわけではないことに注意してほしい。int という型はあらかじめ C 言語で決められた型なのに対し、構造体は自分が好きなように使いやすいように決めた型だと考えても構わない。つまり、“struct personal_data” 型というものを自分で作ったと考えるのである。そうすれば、struct personal_data pdata; という使い方も理解できるはずである。

構造体内の各データには、ドット演算子 (.) を使ってアクセスする。また、年齢を文字列 buffer[16] として受け取っているため、atoi 関数を用いて数字に変換している。atoi は文字列を整数に変換する関数である。詳しくはコマンドラインで man atoi としてみよ。

例 3.7.1.

```
#include <stdio.h>
#include <stdlib.h>
struct personal_data {
  char family_name[16];
  char given_name[16];
  int age;
};

int main() {
  struct personal_data pdata;
  char buffer[16];
  printf("Input family name: ");
  gets(pdata.family_name);
  printf("Input given name: ");
  gets(pdata.given_name);
  printf("Input age: ");
  gets(buffer);
  pdata.age = atoi(buffer);
  printf("Family Name = %s\n", pdata.family_name);
  printf("Given  Name = %s\n", pdata.given_name);
  printf("Age         = %d\n", pdata.age);
  return 0;
}
```

構造体を指すポインタの場合は、次の例 3.7.2.のように、アロー演算子 (->) でその構造体の要素にアクセスすることができる。例題中の qdata->p[0] は (*qdata).p[0] と等価である。ちなみに、この例の構造体は粒子の質量と運動量を格納している。

例 3.7.2.

```
#include <stdio.h>
struct particle_data {
  double mass;
  double p[3]; /* Momentum */
};

int main() {
  struct particle_data pdata;
  struct particle_data *qdata;
  pdata.mass = 0.14;
  pdata.p[0] = 1.2;
  pdata.p[1] = 1.3;
  pdata.p[2] = 1.4;
  qdata = &pdata;
  printf("Mass = %lf\n", qdata->mass);
  printf("Momentum = (%lf, %lf, %lf)\n", qdata->p[0], qdata->p[1], qdata->p[2]);
  return 0;
}
```

練習 3.7.1. “月”, “日”, “1 月 1 日からの日数 (うるう年ではない)” を構成要素とする構造体を作って、“月” と “日” を標準入力から入力すれば、自動的に “1 月 1 日からの日数” の部分を埋め、最後にそれを表示して終了するプログラムを書きなさい。

3.8 ファイルの取り扱い

ファイルから数字を読みこんだり、ファイルに結果を書き込んだりするプログラムを紹介する。ここで挙げた例だけでも基本的なことはできるはずである。より詳しく知りたい場合は参考書などを参照してほしい。

3.8.1 ファイルから数字の読み込み

例 3.8.1.がファイルから数字を読み込むときの雛型となる。まず、FILE *fp; でファイルを取り扱うための構造体を宣言する。char *filename の部分でファイル名を宣言する。そして、fopen でファイルを開く。("r" は read (読み取り専用) でファイルを開くことを示す。) if(fp==NULL){} の部分はエラーが起こったときに処理され、プログラムを強制終了させる。fscanf で順番にファイルに書かれている数字を double 型で読み取っている。fscanf の使い方は、

fscanf(**ファイル構造体のポインタ**, "**フォーマット**", &**変数** 1, &**変数** 2, ...);

で、ファイル構造体のポインタ部分以外は scanf と同じである。そして、最後に fclose(fp); でファイルを閉じている。

例 3.8.1.

```
#include <stdio.h>
#include <stdlib.h>
double product(double x1, double y1, double x2, double y2) {
  return x1*y1 + x2*y2;
```

```
}

int main() {
  FILE *fp;
  char *filename = "vectors.txt";
  double x1,y1,x2,y2,p;
  fp = fopen(filename, "r");
  if (fp == NULL) {
    printf("Can’t open file %s\n", filename);
    exit( 1 );
  }
  /* read first line */
  fscanf(fp, "%lf %lf\n", &x1, &y1);
  /* read second line */
  fscanf(fp, "%lf %lf\n", &x2, &y2);
  fclose(fp);
  p = product(x1, y1, x2, y2);
  printf("Product of (%lf,%lf) and (%lf,%lf) is %lf\n", x1, y1, x2, y2, p);
  return 0;
}
```

例 3.8.1.を実行するためには、vectors.txt というファイル名で以下のような内容の入力ファイルも作っておく必要がある。

```
 1.0    2.0
-1.5    3.5
```

例 3.8.1.のデータを読み込む部分を、例 3.8.2.のように while を使って書き直してみよう。こちらの方がファイルに存在するデータを最後まで読み取るという点で、より汎用性が高い。fscanf はデータの読み込みに失敗すると EOF を返すので、EOF が返ってくるまで読み込みを繰り返している。

例 3.8.2.

```
#include <stdio.h>
#include <stdlib.h>
#define N_DATA 100
int main() {
  FILE *fp;
  char *filename = "vectors.txt";
  double x[N_DATA][2];
  int index = 0;
  fp = fopen(filename, "r");
  if (fp==NULL) {
    printf("Can’t open file %s\n", filename);
    exit(1);
  }
  /* read data */
  while (fscanf(fp, "%lf %lf\n", &x[index][0], &x[index][1]) != EOF) {
    printf("Data %d : (%lf, %lf)\n", index, x[index][0],x[index][1]);
    ++index;
  }
  fclose(fp);
  return 0;
}
```

3.8.2 ファイルへの数字の書き出し

例 3.8.3.がファイルへ数字を書き出すときの雛型である。ファイルを開くところまでは例 3.8.1.とはほぼ同じで、違う点は fopen の第 2 引数が "r" ではなく "w" (書きこみ) である点である。また、書き出すときには fprintf という関数を使っている。fprintf の使い方は、

fprintf(**ファイル構造体のポインタ**, "**フォーマット**", **変数** 1, **変数** 2, ...);

で、ファイル構造体のポインタ部分以外は printf と同じである。最後に、fclose(fp); でファイルを閉じる。

例 3.8.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
double circle_area(double r) {
  return r*r*M_PI;
}

int main() {
  FILE *fp;
  char *filename = "circle_area.txt";
  double radius;
  int i;
  fp = fopen(filename, "w");
  if (fp == NULL) {
    printf("Can't open file %s\n", filename);
    exit(1);
  }
  for(i = 1; i <= 10; ++i) {
    radius = i * 1.0;
    double area = circle_area(radius);
    fprintf(fp, "%lf -> %lf\n", radius, area);
  }
  fclose(fp);
  return 0;
}
```

3.9 その他の制御文

for, while 以外の制御文の書式を紹介する。

3.9.1 switch-case 文

1 の場合には A を、2 の場合には B を、3 の場合には C を、という風に条件分岐をしたい場合に、switch-case 文を用いる。

例 3.9.1.

```
#include <stdio.h>
#include <string.h>
int main() {
  int i;
  printf("Input(int): ");
```

```
    scanf("%d", &i);
    printf("%d / 3 no amari ha ", i);
    switch (i%3) {
      case 1:
        printf("1 desu.\n");
        break;
      case 2:
        printf("2 desu.\n");
        break;
      default:
        printf("0 desu.\n");
    }
    return 0;
  }
```

i%3 の結果が 1 であれば、case 1: 以下の部分から実行する。i%3 の結果が 2 であれば、case 2: 以下の部分から実行する。i%3 の結果が 1 でも 2 でもない場合、つまり、0 の場合は、default: 以下の部分から実行する。もちろん、case 0: を作っても構わない。注意としては、もし例 3.9.1.の switch-case 文中に break; がないと、case 1: の場合は case 2: の部分も default: の部分も実行してしまうことである。つまり、case 1: 以下の部分から下の部分をすべて実行するということである。したがって、case 2: の直前で抜けるためには break; が必要となる。実際に break; を削除して試してみよう。この場合は常に ...0 desu. が表示されるはずである。switch-case 文を簡単にまとめると、次のようになる。

```
switch (X) {
  case A:  ブロック 1
  case B:  ブロック 2
  case C:  ブロック 3
  default: ブロック 4
}
```

X が A ならば、ブロック 1 以下すべてを実行する。X が B ならば、ブロック 2 以下をすべて実行する。X が C ならばブロック 3 以下をすべて実行する。X が A でも B でも C でもない場合は、ブロック 4 を実行する。

3.9.2 do-while 文

while 文と同様に、ある条件が満たされている場合に繰り返しを行うために do-while 文を使う。while 文と違う点は、条件の判定する “タイミング” である。while 文の場合は、まず条件を判定し、満足していれば while 内を実行するが、do-while 文の場合は、まず do-while 内を実行してから条件を判定する。したがって、while 文の場合は while 内を 1 度も実行しない場合があるが、do-while 文の場合は必ず 1 度は do-while 内を実行する。

例 3.9.2.

```
#include <stdio.h>
int main() {
  int i = 100;
  int sum = 0;
  do {
    sum += i;
    --i;
  } while (i != 0);
  printf("sum of integers from 100 to 1 is %d\n", sum);
  return 0;
}
```

例 3.9.2.のように、while() の後の ; を忘れないこと。do-while 文の基本的な動作は次のようになる。

```
do {
  ブロック
} while (A);
```

まずブロックを実行し、次に A をチェックして正しいならば、再びブロックを実行する。そして、再び A をチェックして正しいならば、ブロックを実行する。したがって、A が最初から間違いであってもブロックは必ず一度は実行される。

3.9.3 goto 文

最近では goto 文はあまり好まれないが、for 文がたくさん入り組んである場合 (ネストという) に、for 文全体を一気に抜けるときに必要になることがある。goto 文は無条件に指定された部分に飛んでそこから実行を続けるために使う。例 3.9.3.は goto 文を使うには全く不適切な例だが、イメージをつかむための練習として使ってほしい。簡単に説明すると、while 文は while (1) のように使っているので無限に繰り返し続けるが、i==0 となった時点で goto を使って強引に owari: という部分まで飛んでそこから実行を続ける。

例 3.9.3.

```
#include <stdio.h>
int main() {
  int i = 100;
  int sum = 0;
  while (1) {
    sum += i;
    --i;
    if (i == 0) {
      goto owari;
    }
  }
owari:
  printf("sum of integers from 100 to 1 %d\n", sum);
  return 0;
}
```

goto 文の基本的な使い方は以下の通りである。

```
  goto A;
  ...
A:
  ブロック 2
```

goto A の部分に来ると、A:までジャンプしてその後のブロック 2 を実行する。例 3.9.3.の owari: を return 0; の後にはそのままでは書けないことに注意しよう。なぜなら、owari: の後には必ず何か実行するものが必要だからである。上の基本的な使い方の部分でも書いてあるように、A:の後にはブロック (正確には “文”) が必要である。したがって、どうしても return 0; の後に owari: を書きたい場合は、owari:; とする。

3.10 コマンドライン引数の受け渡し

main 文には、int main() 以外に、int main(int, char**), あるいは int main(int, char*[]) という書き方がある。これらの書式を利用すれば、実行時に引数を与えてそれを利用することができる。つまり、今までは

```
$ ./a.out
```

であったが、これを利用すれば、

```
$ ./a.out input.data output.data
```

のように、ファイル名や数値などをコマンドライン引数としてプログラムに渡すことができる。

3.10.1 ポインタ配列

新しい main 関数の説明の前に、引数の受け渡しに使われているポインタ配列に関して説明しておこう。ポインタ配列は、名前通り、ポインタを要素とする配列 (char*[]) である。配列はポインタを使ってアクセス可能なので、“ポインタのポインタ” (char**) と同様に使うことができる。

例 3.10.1.

```
#include <stdio.h>
int main() {
  int i;
  char *name0 = "Alice";
  char *name1 = "Bob";
  char *name2 = "Claire";
  char *name3 = "David";
  char *name[4];
  name[0] = name0;
  name[1] = name1;
  name[2] = name2;
  name[3] = name3; /* A */
  for (i = 0; i < 4; ++i) { /* B */
    printf("Name%d : %s\n", i, name[i]);
  }
  for (i = 0; i < 4; ++i) { /* C */
    printf("Name%d : %c, %s\n", i, **(name+i), *(name+i));
  }
  return 0;
}
```

例 3.10.1.の結果は以下のようになる。

```
Name0 : Alice
Name1 : Bob
Name2 : Claire
Name3 : David
Name0 : A, Alice
Name1 : B, Bob
Name2 : C, Claire
Name3 : D, David
```

まず、name[0] の型は char* なので、name0 の値等を代入することができる。A の時点で、name というポインタ配列の各要素にアドレスの代入したことになる。つまり、

```
name[0] = "Alice"という文字列の先頭アドレス
name[1] = "Bob"という文字列の先頭アドレス
name[2] = "Claire"という文字列の先頭アドレス
name[3] = "David"という文字列の先頭アドレス
```

となっている。したがって、B の for 文では、printf に文字列の先頭アドレスを渡すことで、名前を表示することができる。次に、C の for 文であるが、これは若干混乱を招くが、次が理解出来れば、何とかクリアできるだろう。

name = 文字へのポインタのポインタ = char** 型
name = 文字へのポインタ = char 型
**name = 文字 = char 型 (※ 文字列ではなく文字)

printf は %s で文字列の先頭アドレス、つまり、char* を受け取り、%c で文字型そのもの、つまり、char を受け取る。また、name+i という書き方は、例えば、name+1 の場合なら、name が name[0] なので、name+1 は name[1] である。

3.10.2 新しい main 関数

例 3.10.2.がコマンドライン引数を受け取ることのできる main 関数の例である。int main(int argc, char* argv[]) は、int main(int argc, char** argv) と書いても同じである。好きなほうを使えばよい。また、伝統的に argc と argv という名前を使っているが、他の変数名でも構わない。

例 3.10.2.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
  FILE *fp;
  char *InputFileName;
  char *OutputFileName;
  int counter = 0;
  double x,y;
  double sumX = 0.0, sumY = 0.0;
  if (argc != 3) { /* A */
    printf("Usage: sumXY InputFile OutputFile\n");
    exit(1);
  }
  InputFileName = argv[1];
  OutputFileName = argv[2];
  /* input */
  fp = fopen(InputFileName, "r");
  if (fp==NULL) {
    printf("Can't open file %s\n", InputFileName);
    exit(1);
  }
  /* read data */
  while(fscanf(fp, "%lf %lf\n", &x, &y) != EOF) {
    sumX += x;
    sumY += y;
    ++counter;
  }
  fclose(fp);
  /* output */
  fp = fopen(OutputFileName, "w");
  if (fp == NULL) {
    printf("Can't open file %s\n", OutputFileName);
    exit(1);
  }
  /* write results */
  fprintf(fp, "Number of Data = %d\n", counter);
```

```
    fprintf(fp, "X Sum = %lf\n", sumX);
    fprintf(fp, "Y Sum = %lf\n", sumY);
    fclose(fp);
    return 0;
  }
```

例 3.10.2.を 3.10.2.c という名前で保存し、次のようにコンパイルする。

```
$ gcc -o sumXY 3.10.2.c
```

sumXY は例 3.10.2.の A の部分の printf のメッセージに合わせてある。次のように実行してみよう。

```
$ ./sumXY
Usage:  sumXY InputFile OutputFile
```

InputFile, OutputFile の二つのコマンドライン引数が必要であるという警告が表示される。例 3.10.2.の A の部分を変更することで、警告メッセージは変更することができる。次のように実行しても同じメッセージが出力されるはずである。

```
$ ./sumXY input.dat
```

```
$ ./sumXY input.dat output.dat 10
```

一番目の例は引数が足りず、二番目の例は引数が多すぎる。例 3.10.2.では argc が 3 以外ならエラーメッセージを表示するようにしてあるので、

```
$ ./sumXY input.dat output.dat 10
```

は 3 個の引数だからエラーが出るのはおかしい、と考えるかもしれない。しかし、これは正しい動作である。なぜなら、引数としてプログラム名 (./sumXY) も 1 個と数えられるからである[*9]。つまり、二番目の例では、argc の値は 4 であり、argv の中身は

```
argv[0] = "./sumXY"の先頭アドレス
argv[1] = "input.dat"の先頭アドレス
argv[2] = "output.dat"の先頭アドレス
argv[3] = "10"の先頭アドレス
```

となっている。

最後に、1 行に 2 個の数字で 10 行程度書いた input.dat を作成し、次のように実行してみよう。output.dat に結果が書き込まれているはずである。

```
$ ./sumXY input.dat output.dat
```

3.11 動的な配列の確保: malloc と free

これまで、ポインタはあらかじめ確保した領域に対してのみ使ってきた。しかし、実行時に、入力値あるいは計算結果を反映して、50 個のデータを取り扱いたいときもあれば、100 個のデータを取り扱いたいときもある。もちろん、配列を利用してあらかじめ十分な大きさの領域 (いまの場合は 100 個以上の数) を確保してお

[*9] argv[0] を用いると、例 3.10.2.の A のエラーメッセージは printf("Usage: %s InputFile OutputFile\n", argv[0]); と書くことができる。このようにしておくと、プログラム名をあらかじめソースコードに明示的に書き込む (「ハードコーディング」とよぶ) 必要がなくなる。

けばよいかもしれないが、平均的に 10 個ぐらいしか利用しないのに、たまに 100 個の場合があるからといって常に 100 個分の領域を確保しておくことは、無駄のように思える。これを回避するために、動的に、つまり、実行時に領域を確保する手段がある。それが、malloc (**m**emory **alloc**cation の略) である。

3.11.1 malloc の使い方

例 3.11.1.に malloc の使い方の雛型を示す[*10]。malloc は領域が確保できた場合はその領域の先頭アドレスを返すが、領域が確保できなかった場合は NULL を返す。NULL の場合は、何らかのエラー処理を行う必要がある。例 3.11.1.では、プログラムを強制的に終了している。sizeof 関数は型の大きさ (バイト数) を調べる関数である。この場合は int の大きさ (通常 4) を調べている。double を大きさを調べたい場合は sizeof(double) のように使う。例 3.11.1.では、int の領域を n_element 個分確保したいので、sizeof(int)*n_element を malloc に与えている。

例 3.11.1.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
  int* array;
  int n_element = atoi(argv[1]);
  if ((array = malloc(sizeof(int)*n_element)) == NULL) {
    printf("Can't allocate memory.\n");
    exit(1);
  }
  int i;
  for (i = 0; i < n_element; ++i) {
    array[i] = i*i;
  }
  for (i = 0; i < n_element; ++i) {
    printf("array[%d] = %d\n", i, array[i]);
  }
  return 0;
}
```

なお、コンパイル時に、malloc の部分で “型の不整合” という警告が出る場合は、つぎのように明示的に型を変換すればよい。

```
array = (int*)malloc(sizeof(int)*n_element)
```

これをキャスト (型変換) という。

3.11.2 free の使い方

例 3.11.1.には (引数の数のチェック以外にも) 適切でない部分がある。malloc で確保した領域を解放していない点である。確保した領域を解放するためには、free という関数を使う[*11]。具体的には、例 3.11.1.の return 0; の前に

```
free(array);
```

と 1 行書けば良い。

[*10] 簡単のため、コマンドライン引数のチェックは行っていない。実際のプログラムでは、配列を確保する前に確保しようとしているサイズを確認すべきである。

[*11] 例 3.11.1.のようにプログラムがすぐに終了してしまう場合には、free は必要ないと主張する人がいるかもしれない。その主張も確かに間違っている訳ではないが、本当に必要なときに忘れないために普段から書く癖をつけるべきである。

次に、例 3.11.1.を若干変更し、free が重要となる例を考えよう。

例 3.11.2.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
  int* array;
  int n_element = atoi(argv[1]);
  int i, j;
  for (j = 0; j < 3; ++j) {
    if ((array = malloc(sizeof(int)*n_element) == NULL){
      printf("Can’t allocate memory.\n");
      exit(1);
    }
    for (i = 0; i < n_element; ++i) {
      switch (j) {
        case 0:
          array[i] = i;
          break;
        case 1:
          array[i] = i*i;
          break;
        default:
          array[i] = i*i*i;
      }
    }
    for (i = 0; i < n_element; ++i) {
      printf("array[%d] = %d\n", i, array[i]);
    }
    free(array);
  }
  return 0;
}
```

もし仮に free がない場合、j=1 のとき、malloc に成功すると array は新しく確保された領域の先頭アドレスを持つ。この時、j=0 で確保した領域はどうなったのだろうか? その領域は、このプログラム自身が確保した領域として、このプログラムが終了するまで解放されない。しかも、array はすでに j=0 のときに確保した領域を忘れているので、このプログラムからもその領域に適切にアクセスすることはできない。つまり、j=0 のときの領域はこのプログラムが持っているが、使うことができない領域として存在し続けることになる[*12]。これは非常に無駄なことである。これを回避するためにも、使わなくなった領域は free することが重要である。もちろん、最近の計算機はメモリ (および、スワップ領域) をたくさん積んでいるので例 3.11.2.程度のプログラムなら free がなくても平気で最後まで動くはずであるが、習慣として、malloc した領域を使い終わったら必ず free するのがよい。

3.12 練習問題の解答例

以下に練習問題の解答例を示す。さまざまなプログラムの書き方があるので、これらの解答例にこだわらないこと。全く分からない場合に参考にする程度が望ましい。

練習 3.2.1.

[*12] これを「メモリリーク」と呼ぶ

```
#include <stdio.h>
int main() {
  /* ax+b=0 */
  double a = 4.0;
  double b = 1.0;
  printf("ax+b=0\n");
  printf("  a = %lf\n", a);
  printf("  b = %lf\n\n", b);
  if (a == 0.) {
    if (b == 0.) {
      printf("  x = all\n");
    } else {
      printf("  x = nothing\n");
    }
  } else {
    printf("  x = %lf\n", -b/a);
  }
  return 0;
}
```

練習 3.2.2.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  /* calculate "n!" */
  int n = 10;
  if (n < 1){
    printf("Can't calculate n!.\n");
    printf("n = %d\n",n);
    exit(1);
  }
  int ans = 1;
  int i;
  for (i = 1; i <= n; ++i) {
    ans *= i;
  }
  printf("%d! = %d\n",n,ans);
  return 0;
}
```

練習 3.2.2.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  /* calculate "n!" */
  int n = 10;
  if (n < 1) {
    printf("Can't calculate n!.\n");
    printf("n = %d\n",n);
    exit(1);
  }
```

```
  int ans = 1;
  int i = 1;
  while (i <= n) {
    ans *= i;
    ++i;
  }
  printf("%d! = %d\n",n,ans);
  return 0;
}
```

練習 3.2.3.

```
#include <stdio.h>
int main() {
  /* n madeno sosuu. */
  int n = 100;
  printf("%d madeno sosuu = ", n);
  int ans = 2;
  while (ans <= n) {
    int flag = 0;
    int i;
    for (i = 2; i < ans; ++i) {
      if (ans%i == 0) {
        break;
      } else if (i == ans - 1) {
        flag = 1;
      }
    }
    if (flag == 1) {
      printf("%d, ", ans);
    }
    ++ans;
  }
  printf("\n");
  return 0;
}
```

練習 3.3.1.

```
#include <stdio.h>
int main() {
  int a[5];
  a[0] = 1.0;
  a[1] = 2.0;
  a[2] = 3.0;
  a[3] = 4.0;
  a[4] = 5.0;
  int i;
  for (i = 0; i < 5; ++i) {
    printf("Start: a[%d] = %d\n", i, a[i]);
  }
  int b[5];
  for (i = 0; i < 5; ++i) {
```

```
    b[i] = a[i];
    a[i] = 0;
    printf("End  : a[%d] = %d, b[%d] = %d\n", i, a[i], i, b[i]);
  }
  return 0;
}
```

練習 3.3.2.

```
#include <stdio.h>
void seki(double a[2][2], double b[2][2], double c[2][2]) {
  c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0];
  c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1];
  c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0];
  c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1];
}

int main() {
  double a[2][2], b[2][2];
  a[0][0] = 1.0;
  a[0][1] = 2.0;
  a[1][0] = 3.0;
  a[1][1] = 4.0;
  b[0][0] = -1.0;
  b[0][1] = -2.0;
  b[1][0] = -3.0;
  b[1][1] = -4.0;
  int i, j;
  for (i = 0; i < 2; ++i) {
    for (j = 0; j < 2; ++j) {
      printf("a[%d][%d] = %lf, b[%d][%d] = %lf\n",
             i, j, a[i][j], i, j, b[i][j]);
    }
  }
  double c[2][2];
  seki(a, b, c);
  for (i = 0; i < 2; ++i) {
    for (j = 0; j < 2; ++j) {
      printf("(a x b)[%d][%d] = %lf\n", i, j, c[i][j]);
    }
  }
  return 0;
}
```

練習 3.4.1.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main() {
  /* calculate "n!" */
  int n;
  printf("Input(int): ");
```

```
  scanf("%d",&n);
  if (n < 1) {
    printf("Can't calculate n!.\n");
    printf("n = %d\n", n);
    exit(1);
  }
  int ans = 1;
  int i;
  for (i = 1; i <= n; ++i) {
    ans *= i;
  }
  printf("%d! = %d\n", n, ans);
  return 0;
}
```

練習 3.5.1.

```
#include <stdio.h>
int main() {
  int array[10];
  int *p;
  int i;
  for (i = 0; i < 10; ++i) {
    array[i] = i*i;
  }
  for (i = 0; i < 10; ++i) {
    printf("array[%d] = %d\n", i, *(array+i));
  }
  for (i = 0; i < 10; ++i) {
    p = &(array[i]);
    printf("array[%d] = %d\n", i, *p);
  }
  return 0;
}
```

練習 3.7.1.

```
#include <stdio.h>
struct date {
  unsigned int day, month;
  unsigned int days;
};

int main() {
  unsigned int nMonth[12] = { 31, 28, 31, 30, 31, 30,
                              31, 31, 30, 31, 30, 31 };
  struct date inputDay;
  printf("Input(month & day) :\n");
  printf("             month :");
  scanf("%d", &(inputDay.month));
  printf("             day   :");
  scanf("%d", &(inputDay.day));
  inputDay.days = 0;
```

```
    int i;
    for (i = 1; i < inputDay.month; ++i) {
      inputDay.days += nMonth[i-1];
    }
    inputDay.days += inputDay.day;
    printf("%d/%d = %d days from 1/1\n", inputDay.month,
           inputDay.day, inputDay.days);
    return 0;
  }
```

第 4 章

LaTeX 入門

LaTeX は L. Lamport の開発した文書清書ソフトウェアである。もともとは、D. Knuth の開発した TeX と呼ばれるソフトウェアで、これを使いやすく拡張したものである。近年では、数式関連の機能をさらに拡張した AMSLaTeX も広く使われている。LaTeX はワードプロセッサのようなソフトウェアではない。画面を見ながら文書を整形しその見たままの姿が印刷結果となるのがワードプロセッサである。例えば代表的なワードプロセッサである Microsoft Office Word では、段落や 2 段組、箇条書などの編集を画面を見ながら行えて、そして画面のイメージがほぼそのまま印刷される。一方、LaTeX では見たままが印刷結果となるわけではないが、ワードプロセッサに匹敵する強力な清書機能を持っている。とくに、LaTeX は数式の清書能力に長けている。

4.1 LaTeX の実行

では、さっそく例を見てみよう。適当なエディタで

例 4.1.1.

```
\documentclass{jarticle}
\begin{document}
$a$のまわりのテイラー展開
\begin{equation}
f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n
\end{equation}
\end{document}
```

と入力してみよう。このファイルを ex.tex という名前で保存する。このファイルを LaTeX で処理するには、platex コマンド[*1] と dvipdfmx コマンドを実行する。

```
$ platex ex.tex
$ dvipdfmx ex.dvi
```

一行目の platex コマンドで DVI ファイル (ex.dvi) が生成される。DVI は **de**vice-**i**ndependent file format の略で、最終的な出力形式 (今の例では PDF) に依存しない出力形式である。dvipdfmx コマンドは、この DVI ファイルから PDF 形式のファイルを生成する。これらのコマンドの実行後、ex.pdf というファイルができていれば成功である。では処理結果を見てみよう。Mac OS X で作業している場合には、

```
$ open ex.pdf
```

Linux で作業している場合には、

```
$ evince ex.pdf
```

[*1] 日本語版 LaTeX。英語のみを含む文書を処理する場合には、latex コマンドを使えばよい。

で PDF ファイルを開くことができる。結果は以下のようになっているはずである。

結果 4.1.1.

a のまわりのテイラー展開

$$f(x)=\sum_{n=0}^{\infty}\frac{f^{(n)}(a)}{n!}(x-a)^n \tag{1}$$

■この章のまとめ

- ⇒ LaTeX ファイルの拡張子は`.tex`である。
- ⇒ LaTeX を使うには、`platex` コマンドを実行する。
- ⇒ `platex` コマンドを実行すると、DVI ファイル (拡張子は `.dvi`) が生成される。
- ⇒ DVI ファイルを PDF 形式に変換するには、`dvipdfmx` コマンドを実行する。
- ⇒ PDF ファイルを見るには、“`open PDF` **ファイル名**” (iMac の場合)、あるいは “`evince PDF` **ファイル名**” (Linux の場合) を実行する。

4.2 LaTeX に関する全体的なこと

すでに例 4.1.1.で見たとおり、LaTeX のソースファイルには、本文以外の情報も書き込まれている。具体的には、

例 4.2.1.

```
\documentclass{jarticle}
\usepackage[dvipdfmx]{graphicx}
\begin{document}

% ここに本文を書く

\end{document}
```

のところがかならず書かなくてはならないところである。特に `\begin{document}`より前の部分をプリアンブルと呼ぶ。以下の例題では、このかならず書かなくてはいけない部分を省略してあるので注意すること。

さて、 例 4.2.1.にはパーセント記号 (%) で始まる行が書かれている。 LaTeX では、% で始まる行はコメントとみなされる。行の途中で % が現われたときは、そこから行の最後までがコメントになる。コメントは LaTeX の動作には影響を与えない。

文章はソースファイルの中にそのまま書けばよい。ソースファイルの途中で改行しても清書の結果には影響を与えない。

例 4.2.2.

```
% Belle 実験の説明
茨城県つくば市の高エネルギー加速器研究機構では、
Belle 実験と呼ばれる素粒子実験が進められています。
Belle 実験のもっとも重要な目標は
現在の素粒子物理学を支える重要な理論である
小林-益川理論の正当性を検証することです。
```

この出力は

結果 4.2.2.

茨城県つくば市の高エネルギー加速器研究機構では、Belle 実験と呼ばれる素粒子実験が進められています。Belle 実験のもっとも重要な目標は現在の素粒子物理学を支える重要な理論である小林-益川理論の正当性を検証することです。

となる。出力は、もっとも美しく清書されるように、LaTeX が自動的に改行を行なってくれる。逆にどうしても自分で改行させたいときは、\\ をソースファイル中に書き込めばよい。

例 4.2.3.

```
東京大学物性研究所 \\ \\
秋葉原からつくばエクスプレスに乗り、
柏の葉キャンパス前で下車します。\\
運賃は 670 円です。\\
所要時間はおよそ 30 分です。
```

結果は次のとおりである。

結果 4.2.3.

東京大学物性研究所

秋葉原からつくばエクスプレスに乗り、柏の葉キャンパス前で下車します。
運賃は 670 円です。
所要時間はおよそ 30 分です。

■この節のまとめ

⇒ ソースファイルには必ず書かなくてはいけない部分がある。
⇒ % から行の最後まではコメントである。
⇒ 文章はソースファイル中に直接書き込む。その際、改行は無視される。
⇒ 強制改行をするには \\ を使う。

4.3 フォント

印刷される文字を強調するために、文字を太くしたり (ボールド体)、ななめにしたり (斜体) することがある。印刷される文字の形状のことをフォントといい、その形状を変更することをフォントを変更するといいます。ちなみに、通常のフォントは立体と呼ばれる。

ボールド体はとくに強調したい言葉に使う。文章中でフォントを変更するには、

例 4.3.1.

```
東京駅から東京大学に向かうには、
\textbf{地下鉄丸ノ内線}を使うのが便利です。
\textbf{池袋}ゆきに乗り\textbf{本郷三丁目}駅で下車します。
```

のように`\textbf{文字}`命令を使う。結果は、

結果 4.3.1.

東京駅から東京大学に向かうには、**地下鉄丸ノ内線** を使うのが便利です。**池袋**ゆきに乗り**本郷三丁目**駅で下車します。

となる。斜体は、変数などのパラメータを示したり、英単語を強調するのに用いる。

例 4.3.2.

```
\textbf{ls}コマンドに\textit{file}引数を渡すと
その\textit{file}に関する情報を表示します。
```

結果 4.3.2.

ls コマンドに *file* 引数を渡すとその *file* に関する情報を表示します。

タイプライタ体は計算機への入力、計算機からの出力、プログラムなどを示すのに使う。

例 4.3.3.

```
標準ライブラリ関数には、\texttt{sin()}や\texttt{cos()}、
\texttt{tan()} といった三角関数も用意されています。
```

結果 4.3.3.

標準ライブラリ関数には `sin()` や `cos()`、`tan()` といった三角関数も用意されています。

■この節のまとめ

⇒ フォントの種類を変更するには `\textbf{文字}` 、 `\textit{文字}` 、`\texttt{文字}` などを使う。

4.4 論文のスタイル

LaTeX には論文を清書するのに便利な機能が備わっている。

4.4.1 表題

通常、論文の表題には、論文の題名、著者、著者の所属、提出日時、論文の要旨などを書く。

例 4.4.1.

```
\documentclass{jarticle}
\usepackage{graphicx}

% 表題、著者、所属、提出日時
\title{各種プログラミング言語に関する考察}
\author{物理太郎\\東京大学大学院理学系研究科物理学専攻}
\date{2015 年 4 月 8 日}
\begin{document}

% 表題の表示
\maketitle

%%% ここに論文の本文を書く %%%

\end{document}
```

レポート程度の簡単なものならば論文の要旨は不要であろう。結果は次のとおり。

結果 4.4.1.

各種プログラミング言語に関する考察

物理太郎
東京大学大学院理学系研究科物理学専攻
2015 年 4 月 8 日

4.4.2 論文の構成

通常、論文は内容にしたがって章分けされる。以下で LaTeX で章を分ける例を示す。

例 4.4.2.

```
\section{C 言語}
C 言語は、
Brain Kernighan と Dennis Ritchie によって開発された
プログラミング言語である。
簡潔性と柔軟性の両方を備えたこの言語は、
さまざまな分野のソフトウェア開発に広く利用されている。
われわれが普段利用している UNIX オペレーティングシステムも
この C 言語を用いて記述されたものである。\\
\dots

\subsection{C 言語の歴史}
C 言語はベル研究所の Ritchie によって 1972 年頃開発された。
C 言語は Algol と呼ばれるプログラミング言語の流れを受け継ぐ言語で、
直接の母体は B 言語と呼ばれる言語である。\\
\dots

\subsection{C 言語の特徴}
この節では C 言語の特徴について述べよう。

\subsubsection{構造化プログラミング言語}
C 言語にはデータのまとまりを表す構造体と呼ばれるデータ型がある。
データを集合として扱う構造データ型は FORTRAN にはなかった。\\
\dots

\subsubsection{機械指向型プログラミング言語}
ひとつの C 言語の行なう命令は FORTRAN などに比べると簡潔であり
計算機に与えられる命令に近い。
これはプログラミング言語自体による副作用なしに
計算機を思い通りに直接操作できることを意味する。\\
\dots

\section{FORTRAN}
FORTRAN は FORmula TRANslator の名が示すとおり、
科学技術計算の能力に長けたプログラミング言語である。\\
\dots
```

このように、章や節にタイトルを付けるには、`\chapter{}`、`\section{}`、`\subsection{}`、`\subsubsection{}` を使う。タイトルを`{}`の中に書き、そのあとには本文となる文章を続ける。章や節の番号は LaTeX が自動的に振ってくれる。

結果は次のようになる。

結果 4.4.2.

1　C 言語

C 言語は、Brain Kernighan と Dennis Ritchie によって開発されたプログラミング言語である。簡潔性と柔軟性の両方を備えたこの言語は、さまざまな分野のソフトウェア開発に広く利用されている。われわれが普段利用している UNIX オペレーティングシステムもこの C 言語を用いて記述されたものである。

...

1.1　C 言語の歴史

C 言語はベル研究所の Ritchie によって 1972 年頃開発された。C 言語は Algol と呼ばれるプログラミング言語の流れを受け継ぐ言語で、直接の母体は B 言語と呼ばれる言語である。

...

1.2　C 言語の特徴

この節では C 言語の特徴について述べよう。

1.2.1　構造化プログラミング言語

C 言語にはデータのまとまりを表す構造体と呼ばれるデータ型がある。データを集合として扱う構造データ型は FORTRAN にはなかった。

...

1.2.2　機械指向型プログラミング言語

ひとつの C 言語の行なう命令は FORTRAN などに比べると簡潔であり計算機に与えられる命令に近い。これはプログラミング言語自体による副作用なしに計算機を思い通りに直接操作できることを意味する。

...

2　FORTRAN

FORTRAN は FORmula TRANslator の名が示すとおり、科学技術計算の能力に長けたプログラミング言語である。

...

■この節のまとめ

⇒ 論文の表題は、\title{} 、\author{} 、\date{} などで指定する。

⇒ 表題を出力するには、\maketile を使う。

⇒ 章や節のタイトルを付けるには、\chapter{}、\section{}、\subsection{}、\subsubsection{} などを用いる。

4.5　箇条書き

論文の中では箇条書きを使うことは避けたほうがよいといわれている。特に、何かの処理の手順を説明するときは、箇条書きにするよりは言葉を使って説明するべきである。しかし、場合によっては箇条書きの方が説

明が伝わりやすいこともある。単に項目を並列に並べたいときは箇条書きでもよいであろう。この節では箇条書きの方法を説明する。

箇条書きにしたいときは次のように書く。

例 4.5.1.

```
午後の紅茶（ミルクティー）の原材料名:
\begin{itemize}
  \item 牛乳
  \item 砂糖
  \item 加糖練乳
  \item 紅茶
  \item 香料
  \item 乳化剤
  \item ビタミン C
\end{itemize}
```

このように、文章を\begin{itemize}と\end{itemize}とで囲むと、その領域は箇条書き環境になる。箇条書き環境の中では、\item というコマンドを使うと新しい項目を始めることができる。

結果 4.5.1.

午後の紅茶 (ミルクティー) の原材料名:

- 牛乳
- 砂糖
- 加糖練乳
- 紅茶
- 香料
- 乳化剤
- ビタミン C

また、itemize の代わりに enumerate を使うと、各項目のラベルが 1, 2, 3, ··· のような番号となる。

例 4.5.2.

```
この講義の評価について該当するものに丸を付けて下さい。
\begin{enumerate}
\item 大変良かった。
\item まあまあ良かった。
\item 普通だった。
\item あまり良くなかった。
\item かなり良くなかった。
\end{enumerate}
```

結果 4.5.2.

この講義の評価について該当するものに丸を付けて下さい。

1. 大変良かった。
2. まあまあ良かった。
3. 普通だった。
4. あまり良くなかった。
5. かなり良くなかった。

■この節のまとめ

⇒ \begin{itemize} と \end{itemize} で囲まれた部分は箇条書き環境になる。

⇒ \begin{enumerate} と \end{enumerate} で囲まれた部分は番号付き箇条書き環境になる。

⇒ 箇条書きの項目を始めるには\item を使う。

4.6 表の作成

表を作成するには次に示す雛型を使えばよい。

例 4.6.1.

```
\begin{table}
  \begin{center}
    \caption{月刊誌の出版社と発売日}
    \begin{tabular}{|l|l|l|}
      \hline
      雑誌名 & 出版社 & 発売日 \\
      \hline \hline
      トランジスタ技術 & CQ 出版 & 10 日 \\
      \hline
      UNIX Magazine & アスキー & 19 日 \\
      \hline
      SD ソフトウェアデザイン & 技術評論社 & 19 日 \\
      \hline
      アフタヌーン & 講談社 & 25 日 \\
      \hline
    \end{tabular}
  \end{center}
\end{table}
```

結果 4.6.1.

表 1: 月刊誌の出版社と発売日

雑誌名	出版社	発売日
トランジスタ技術	CQ 出版	10 日
UNIX Magazine	アスキー	19 日
SD ソフトウェアデザイン	技術評論社	19 日
アフタヌーン	講談社	25 日

\hline と書いたところには、横線が引かれる。\hline \hline と 2 つ続けて書くと二重線になる。表の各行は、それぞれの要素の間を&で区切り、最後に \\ で改行して行が終わったことを LaTeX に伝える。

この表は横の 1 行に 3 つの要素を持つ表である。各要素は表の 1 マスに左詰めで書かれている。このことは

```
\begin{tabular}{|l|l|l|}
```

の{|l|l|l|}によって表現されている。 l は左詰めを意味し、それを 3 つ並べることで、1 行に 3 つの要素が入ることを表現しています。左詰めの代わりに中央揃えがよければ c を、右詰めがよければ r を指定する。

それぞれの l の両隣の縦棒は、要素と要素の間を縦線で区切って印刷してほしいということを LaTeX に伝えている。この縦棒を取ってしまうと、要素間の縦線が引かれなくなる。

表の表題は

```
\caption{月刊誌の出版社と発売日}
```

のように \caption の後の {} の中に書く。最後にもう 1 つ例を挙げておこう。

例 4.6.2.

```
\begin{table}
  \begin{center}
    \caption{代表的な粒子の質量}
    \begin{tabular}{|c|r|}
      \hline
      粒子 & 質量 (MeV/$c^2$) \\
      \hline \hline
      $e ^ \pm$ & 0.5110 \\
      \hline
      $\mu ^ \pm$ & 105.7 \\
      \hline
      $\pi ^ \pm$ & 139.6 \\
      \hline
      $\pi ^ 0$ & 135.0 \\
      \hline
      $J/\psi$ & 3097 \\
      \hline
      $B ^ 0$ & 5279 \\
      \hline
    \end{tabular}
  \end{center}
\end{table}
```

結果 4.6.2.

表 1: 代表的な粒子の質量

粒子	質量 (MeV/c^2)
$e^\pm$	0.5110
$\mu^\pm$	105.7
$\pi^\pm$	139.6
π^0	135.0
J/ψ	3097
B^0	5279

4.7 数式

数式は、例 4.1.1.で見たとおり、

例 4.7.1.

```
\begin{equation}
%%% ここに数式を書きます %%%
\end{equation}
```

というスタイルで記述する。\begin{equation}と\end{equation}とで囲まれた範囲では、数式環境の中のみで有効な様々なコマンドが使える。また、数式の番号は自動的に振られる。

結果 4.1.1.で見たとおり、\begin{equation}と\end{equation}とで囲まれた領域の数式は、改まった行の中央に表示される。しかし、場合によっては

結果 4.7.2.

x の関数 $y = ax^2 + bx + c$ のうち $a \ne 0$ であるものを x の 2 次関数といいます。

のように文章中にインラインで数式を書きたいこともあるだろう。そういう場合は、文章中に$と$で囲った数式を書けばよい。結果 4.7.2.を与えるソースは、以下のとおりである。

例 4.7.2.

```
$x$の関数$y = ax ^ 2 + bx + c$のうち
$a \ne 0$ であるものを$x$の 2 次関数といいます。
```

以降の例題中ではスペースの都合上、2 つ以上の数式を並べて書いてある場合があるが、、実際には 1 つずつ \begin{equation}と\end{equation}とで囲むか、$と$とで囲まなくてならない。

4.7.1 添え字

上付き添え字は^で、上付き添え字は_で表現する。

例 4.7.3.

```
y = a x ^ 2 + b x + c
S_n = a_1 + a_2 + a_3 + \cdots + a_n
```

結果 4.7.3.

$$y = ax^2 + bx + c$$
$$S_n = a_1 + a_2 + a_3 + \cdots + a_n$$

2 文字以上からなる項を添え字にしたいときは、{}で添え字を囲む。

例 4.7.4.

```
Tr\,A = A_{11} + A_{22} + A_{33} + \cdots + A_{ii} + \cdots + A_{nn}
```

結果 4.7.4.

$$Tr\,A = A_{11} + A_{22} + A_{33} + \cdots + A_{ii} + \cdots + A_{nn}$$

添え字に限らず、LaTeX の数式コマンドを 2 文字以上の項に作用させるときはかならず項を{}で囲む必要がある。逆に 1 文字だけの項を{}で囲んでも特に問題はない。なお、例 4.7.4.の中で ‘\,’ というコマンドが使われているが、これはスペースの微調整をしたいときに使用する。

4.7.2 分数

分数は \frac{ 分子 }{ 分母 } のように表現する。

例 4.7.5.

```
1 + \frac{1}{2} = \frac{3}{2}
1 + \frac{1}{1+\frac{1}{2}} = \frac{5}{3}
```

結果 4.7.5.

$$1+\frac{1}{2}=\frac{3}{2}$$

$$1+\frac{1}{1+\frac{1}{2}}=\frac{5}{3}$$

4.7.3 関数

数学環境では文字はすべて変数と見なされるため、`cos x` という入力は $cos x$ と斜体で印刷される。これではあたかも c、o、s、x の積のようである。数学関数は正しくは立体で表示されなくてはならない。立体で印刷するためには、`\cos x` と入力する。このように数学関数の多くはバックスラッシュ (\) に関数名をつなげると表現できる。

例 4.7.6.

```
\frac{d}{dx} \sin x = \cos x
```

結果 4.7.6.

$$\frac{d}{dx} \sin x = \cos x$$

表 4.1に主な数学関数を示す。

表 4.1　主な数学関数

`\cos`	`\sin`	`\tan`
`\cosh`	`\sinh`	`\tanh`
`\log`	`\ln`	`\exp`
`\lim`		

4.7.4 フォント

前節でも触れたが、LaTeX は立体で印刷したい文字も斜体で印刷してしまうことがある。数式中でフォントを強制的に変更するときは前に説明した方法は使わない。数式中で立体の文字を印刷するには、`\mathrm`{ 文字 } と書く。

例 4.7.7.

```
\frac{\mathrm d}{\mathrm{d}x}\mathrm{e}^x = \mathrm{e}^x
```

これにより、立体で表示されるべきところは正しく立体で印刷される。

結果 4.7.7.

$$\frac{\mathrm d}{\mathrm{d}x}\mathrm{e}^x = \mathrm{e}^x$$

4.7.5 積分・和・積

積分、和、積は以下のように表現する。

例 4.7.8.

```
\int _1 ^2 \frac{1}{x^2} \mathrm{d}x = \frac{1}{2}
\sum _{k=1} ^n k = \frac{n(n+1)}{2}
n! = \prod _{k=1} ^n k
```

結果 4.7.8.

$$\int _1 ^2 \frac{1}{x^2} \mathrm{d}x = \frac{1}{2}$$

$$\sum _{k=1} ^n k = \frac{n(n+1)}{2}$$

$$n! = \prod _{k=1} ^n k$$

4.7.6 根号・ベクトル・上線

根号、ベクトル、上線の記号は以下のように表現する。

例 4.7.9.

```
\sqrt{x^2 + 2x + 1} = |x+1|
\overrightarrow{OA} + \overrightarrow{AP} = \overrightarrow{OP}
\overline{x + \mathrm{i}y} = x - \mathrm{i}y
```

結果 4.7.9.

$$\sqrt{x^2 + 2x + 1} = |x+1|$$

$$\overrightarrow{OA} + \overrightarrow{AP} = \overrightarrow{OP}$$

$$\overline{x + \mathrm{i}y} = x - \mathrm{i}y$$

4.7.7 ギリシャ文字

ギリシャ文字は文字を英語表記で綴り、前にバックスラッシュを付けると表示される。α は `\alpha` と入力する。ギリシャ文字の大文字は、英語の綴りの最初の 1 文字を大文字にすると表示できる。 `\Psi` と書けば Ψ が印刷される。

表 4.2　ギリシャ文字一覧

コマンド	小文字	大文字	コマンド	小文字	大文字	コマンド	小文字	大文字
\alpha	α	–	\iota	ι	–	\rho	ρ	–
\beta	β	–	\kappa	κ	–	\sigma	σ	Σ
\gamma	γ	Γ	\lambda	λ	Λ	\tau	τ	–
\delta	δ	Δ	\mu	μ	–	\upsilon	υ	Υ
\epsilon	ϵ	–	\nu	ν	–	\phi	ϕ	Φ
\zeta	ζ	–	\xi	ξ	Ξ	\chi	χ	–
\eta	η	–	o	o	O	\psi	ψ	Ψ
\theta	θ	Θ	\pi	π	Π	\omega	ω	Ω

– となっている箇所は、通常のアルファベットの大文字で代用する。

4.7.8　行列

行列を書く方法は表の作成と似ている。

例 4.7.10.

```
\sigma_3 = (
\begin{array}{cc}
  1 & 0 \\
  0 & -1 \\
\end{array} )
```

表のところで説明されている書き方のうち、`\begin{tabular}` から`\end{tabular}` までをまねして、`tabular` を `array` に書き換える。行列の丸カッコは自動的には印刷されないので自分で書く[*2]。

結果 4.7.10.

$$\sigma_3 = (\begin{array}{cc} 1 & 0 \\ 0 & -1 \\ \end{array})$$

しかし、これではカッコの大きさがあまり美しくない。カッコを括られている内容の高さに合わせて大きくするには、`\left(` と `\right)` を使う。

例 4.7.11.

```
\phi (x) = \sqrt{\frac{1}{2}} \left(
\begin{array}{c}
  0 \\
  v + h(x)
\end{array} \right)
```

結果 4.7.11.

[*2] ちなみに、AMSLaTeX では `pmatrix` 環境が用意されており、行列をより自然な形で記述できる。

$$\phi(x) = \sqrt{\frac{1}{2}} \left(\begin{array}{c} 0 \\ v + h(x) \end{array} \right) \qquad (1)$$

\left(と \right) の組み合わせは、中に入るものが行列でなくても使える。また、丸カッコ以外にも使える。

例 4.7.12.

```
J_{\mu}^{NC}(\nu) =
\frac{1}{2}
\left[
\overline{u}_{\nu}\gamma_{\mu} \frac{1}{2} (1-\gamma^5)u_\nu
\right]
```

結果 4.7.12.

$$J_{\mu}^{NC}(\nu) = \frac{1}{2} \left[\overline{u}_{\nu}\gamma_{\mu} \frac{1}{2} (1-\gamma^5)u_\nu \right] \qquad (1)$$

このようにカギカッコなどにも使える。

4.7.9 特殊記号

最後に LaTeX で使える特殊記号を示しておく。

表 4.3　主な特殊記号

コマンド	印刷結果
\ne	$\ne$
\le	$\le$
\ge	$\ge$
\equiv	$\equiv$
\pm	$\pm$
\times	$\times$
\infty	∞
\bullet	$\bullet$
\prime	$\prime$
\dagger	$\dagger$
\partial	∂
\to	$\to$
\nabla	∇
\triangle	$\triangle$

■この節のまとめ

⇒ 数式を書くには、\begin{equation}と\end{equation}で囲む。

⇒ 文章中で数式を書くには $ と $ で囲む。

⇒ 添え字、分数、関数、積分記号などについては、それぞれにコマンドが用意されている。

4.8 図の貼り込み

LaTeX では、gnuplot などで生成した (2.4) 節) 図のファイル (ポストスクリプト、PDF など) を文章中に貼り込むことが可能である。

4.8.1 EPS ファイルの貼り込み

図を貼り込むには以下の雛型を用いる。

例 4.8.1.

```
\begin{figure}
  \begin{center}
    \includegraphics[width=8cm,clip]{test.eps}
    \caption{$y = x^2$ のグラフ}
  \end{center}
\end{figure}
```

結果 4.8.1.

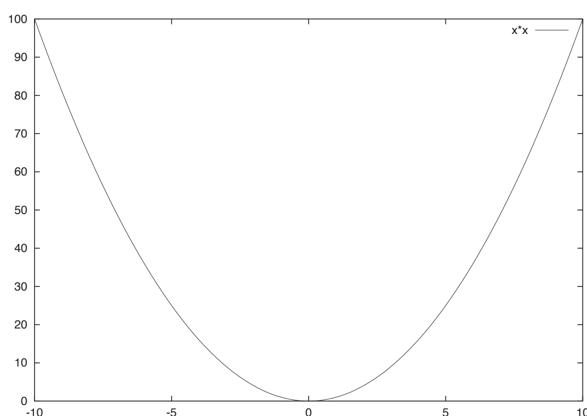


図 1: $y = x^2$ のグラフ

注意すべきところは

```
\includegraphics[width=8cm,clip]{test.eps}
```

という部分である。ファイル名は {} の中で指定する。また width=8cm という記述により図の横幅を指定している。図の表題も表のときと同じく \caption コマンドを使う。

LaTeX は図表を自動的に適切な位置に配置してくれるが、begin{figure} の後に [h] を付けると、その場に配置することができる。それでもうまくいかない場合には、here.sty を使い、**例題 4.8.2.**のように begin{figure} の後に [H] を付けることにより、強制的に指定の位置に配置することも出来る。

例 4.8.2.

```
\documentclass{jarticle}
\usepackage{graphicx}
\usepackage{here}
```

```
\begin{document}
\begin{figure}[H]
  \begin{center}
    \includegraphics{test.epsi}
    \caption{$y = x^2$ のグラフ}
  \end{center}
\end{figure}
\end{document}
```

4.8.2 図の回転

貼り込みたいファイル中の図が 90 度回転してしまっていることがある。LaTeX では図を貼り込むときに回転させることができる。

例 4.8.3.

```
\rotatebox{-90}{
  \includegraphics[width=8cm,clip]{test.eps}
}
```

このように、 \rotatebox{-90} を使うことにより、図を -90 度回転させることができる。

■この節のまとめ

⇒ LaTeX の文章には、ポストスクリプトあるいは PDF 形式の図を貼り込める。
⇒ 実際に図を貼り込むには雛型を使う。
⇒ 貼り込む図を回転させるには \rotatebox{**角度**}{} を使う。

4.9 参照と参考文献

4.9.1 参照

LaTeX には、数式や表、図に付いた番号を簡単に参照する機能が用意されている。まずは具体例から見ていこう。

例 4.9.1.

```
媒質中を物質が通過するとき、チェレンコフ光が発生するための条件は
\begin{equation}
n > \frac{1}{\beta} = \sqrt{1+\left(\frac{m}{p}\right)^{2}}
\end{equation}
である。ただし式 (1) において、
$n$ は媒質の屈折率、$m$ は物質の質量、$p$ は物質の運動量である。
```

この清書結果は次のようになる。

結果 4.9.1.

媒質中を物質が通過するとき、チェレンコフ光が発生するための条件は

$$n > \frac{1}{\beta} = \sqrt{1+\left(\frac{m}{p}\right)^{2}} \qquad (1)$$

である。ただし式 (1) において、n は媒質の屈折率、m は物質の質量、p は物質の運動量である。

例題 4.9.1. ではチェレンコフ光の発生条件の式を本文中に「式 (1)」とハードコーティングしている。

さて、 **例題 4.9.1.** の式の前に別の式を挿入することになったとする。すると、式の番号はすべて 1 つずつ繰り下がるので、文章中で「式 (1)」と書いているところをすべて「式 (2)」に書き直さなければならない。このような番号の付け替えを手で行うのはとても大変であるし、間違いが起こりやすい。

こういった手間を省くために、 LaTeX には数式や表・図に名前を割り当てる機能が用意されている。

例 4.9.2.

```
媒質中を物質が通過するとき、チェレンコフ光が発生するための条件は
\begin{equation}
n > \frac{1}{\beta} = \sqrt{1+\left(\frac{m}{p}\right)^{2}} \label{cherenkov}
\end{equation}
である。ただし式 (\ref{cherenkov}) において、
$n$ は媒質の屈折率、$m$ は物質の質量、$p$ は物質の運動量である。
```

例題 4.9.2. は**例題 4.9.1.** と同じ清書結果を与える。

数学環境中で `\label` を使うとその数式に名前が付けられる。付ける名前は `\label` の後の `\{\}` の中に記述する。**例題 4.9.2.** では `cherenkov` という名前が付けられている。

さて、名前の付けられた数式をその名前で参照すると、その箇所が数式の番号に展開される。参照は `\ref` コマンドを使う。**例題 4.9.2.** では `\ref{cherenkov}` として参照している。`\label{cherenkov}` によって名前付けした数式の番号は 1 だったので、 `\ref{cherenkov}` の参照によってこれが 1 に展開されることになる。

以下に参照の例を挙げておく。

例 4.9.3.

```
本年度の遠足について、以下のとおり決定いたしましたのでお知らせします。
\begin{table}
  \begin{center}
    \caption{遠足の目的地（学年別）}
    \label{destination}
    \begin{tabular}{|l|l|}
      \hline
      学年 & 目的地 \\
      \hline \hline
      4 年生 & 近くにある公園 \\
      \hline
      5 年生 & 遠くにある公園 \\
      \hline
      6 年生 & かなり遠くにある公園 \\
      \hline
    \end{tabular}
  \end{center}
\end{table}
表 \ref{destination} をよく読んで自分の目的地をきちんと確認し、
遠足の目的地に相応しい準備をするように心がけてください。
\\
なお、おやつの持参は
\begin{equation}
\sum^{全おやつ} 各おやつの購入金額 \le 500 円 \label{oyatu}
\end{equation}
```

```
を条件とします。
ただし、バナナは 式 (\ref{oyatu}) にかかわらず好きなだけ持参して差しつかえありません。
```

結果 4.9.3.

本年度の遠足について、以下のとおり決定いたしましたのでお知らせします。表 1 をよく読ん

表 1: 遠足の目的地 (学年別)

学年	目的地
4 年生	近くにある公園
5 年生	遠くにある公園
6 年生	かなり遠くにある公園

で自分の目的地をきちんと確認し、遠足の目的地に相応しい準備をするように心がけてください。
なお、おやつの持参は

$$\sum^{\text{全おやつ}} \text{各おやつの購入金額} \leq 500\,\text{円} \tag{1}$$

を条件とします。ただし、バナナは 式 (1) にかかわらず好きなだけ持参して差しつかえありません。

4.9.2 参考文献

論文やレポートでは、通常、末尾に参考文献リストを付ける。参考文献の番号も数式などと同じように自動的に振ることができる。

例 4.9.4.

```
第 1 象限内の複素数 $z$ について、
\begin{equation}
w(z) \equiv \mathrm{e}^{-z^2}\mathrm{erfc}(z)
\end{equation}
を近似的に計算するアルゴリズムが存在し \cite{wofz}、
その計算誤差は $10 ^ {-10}$ 以下である。
\\
このアルゴリズムは、 BELLE 実験 \cite{BELLE} で使用するプログラムの内部で利用されている。


...

\begin{thebibliography}{99}
\bibitem{wofz}
  Walter Gautschi, Comm ACM. \textbf{12} 635, (1969)
\bibitem{BELLE}
  The BELLE Collaboration, KEK Report 95-1 (1995)
\end{thebibliography}
```

結果 4.9.4.

第 1 象限内の複素数 z について、

$$w(z) \equiv \mathrm{e}^{-z^2}\mathrm{erfc}(z) \tag{1}$$

を近似的に計算するアルゴリズムが存在し [1]、その計算誤差は 10^{-10} 以下である。
このアルゴリズムは、 BELLE 実験 [2] で使用するプログラムの内部で利用されている。

参考文献

[1] Walter Gautschi, Comm ACM. **12** 635, (1969)
[2] The BELLE Collaboration, KEK Report 95-1 (1995)

文章中に書く `\cite` では引用する論文の名前を指定する。論文の名前は自分が覚えやすいものであれば何でも構わないが、`\bibitem` によって名前が定義されている必要がある。`\bibitem` は `\begin{thebibliography}` と `\end{thebibliography}` とで囲まれた領域に書く。この領域は清書すると参考文献の章となる。`\bibitem` に続く {} の中には自分で決めた論文名を書く。この名前が `\cite` によって参照されることになる。その後あとには、その論文の著者名や収録されている雑誌名などを書く。ここに書いたことは、参考文献の章が清書されるときにそのまま印刷される。

`\cite` と `\ref` を、 `\bibitem` と `\label` をそれぞれ対応付けて覚えておこう。

4.9.3 コンパイル

参照を含むソースファイルを LaTeX で処理するときは、`platex` コマンドを何回か実行する必要がある。これは、 `platex` が参照関係を解決することが、 1 回のコマンド実行だけではできないからである。参照が解決できていない状態では、展開されるべき参照の部分が ?? と表示される。`platex` の実行を何回か繰り返し、?? になっている参照がなくなったことを確認すること。

■この節のまとめ

- ⇒ 数式や表、図には `\label{`**名前**`}` で名前を付けられる。
- ⇒ 数式などに付けられた番号は、 `\label{`**名前**`}` で付けた名前を手掛かりにして `\ref{`**名前**`}` と書くことで参照できる。
- ⇒ 参考文献は `\cite{`**論文名**`}` のようにして参照する。
- ⇒ 参考にした論文は、 `\begin{thebibliography}` から`\end{thebibliography}` までの領域に書く。
- ⇒ 論文は `\bibitem{`**論文名**`}` 著者、雑誌名 ...のように列挙する。
- ⇒ 参照を含むソースファイルに対しては、何回か `platex` コマンドを実行する必要がある。

第 5 章
バージョン管理システム

5.1 バージョン管理システムとは?

計算機を使って何らかのシミュレーション (実験) を行う場合、実験の場合と同様に、どのような計算を行ったか、詳細な記録を残さなければならない。行った計算、特に結果を論文として発表した計算については、結果を再現するのに最低限必要な情報を正確に残しておくことが重要である。また、作業効率という観点からも、履歴をきちんと管理し、系統的にバックアップを残しておくことのメリットは大きい。計算機上においたファイルの場合、手書きのノートと異なり簡単に上書きできてしまう。さらに通常 (最後に変更された日時以外) 何の痕跡も残らない。その意味では、実際のノート以上に履歴管理に注意する必要があるといえる。

「ファイル管理」の方法として、ファイル名・ディレクトリ名による管理が広く行われている。例えば、名前に最終変更日付や、変更を行った人の名前、適当なバージョン番号などを付ける等々。あるいは、手書きのログファイルによる記録の場合もあるだろう。これらの方法の問題は、人間は記録を付け忘れる、あるいは記録を間違うことが多いという点である。あるいは、結果を再現するには記録が不完全な場合も多い。人により命名規則がばらばらであったり、コンピュータ間でコピーを繰り返すと、どれを修正したか、どれが新しいか分からなくなるといったことも頻発する。さらに面倒なのは、同じバージョンを元に、複数の人が独立に修正を行ってしまった場合である。いったん分岐してしまったバージョンを人間が手でマージするのは、ファイルを一から作成する以上の手間となってしまう場合さえある。

バージョン管理システム (Version Control System) とは、ファイルの変更履歴をリポジトリと呼ばれるデータベースで一括管理するシステムである。バージョン管理システムでは、更新 (「チェックイン」と呼ばれる) 毎に一意なバージョン番号 (リビジョン) を付与し、全ての修正履歴 (差分) をデータベースに保存する。これにより、任意のリビジョン間の比較も簡単に行うことができる。バージョン管理システムは、もともとは C などのプログラムのソースコードを管理するために開発されたシステムであるが、例えば LaTeX のソースコードなど、それ以外の種類のファイル管理にも同様に使うことができる。また、ファイルの更新や参照はネットワーク経由で行うことが可能である。これにより、閉じた環境だけでなく、チームや分散環境においても一貫したバージョン管理が可能となる。複数箇所から同時に更新した場合には更新箇所の衝突 (コンフリクト) が起こる場合も多いが、そのような衝突を検出し、矛盾なく解決するための仕組みもある。また、ブランチ・タグといったソースコードの公開やメンテナンスに不可欠な機能も備えている。バージョン管理システムは、一言でいうと、超優秀な (かつ超まじめな) 秘書のようなものであり、一人で使っても複数人で使っても非常に便利なツールである。

5.2 diff と patch

バージョン管理システムについての説明を始める前に、UNIX で差分を管理するためのツール (コマンド) について紹介しておこう。UNIX では、プログラムのソースコードや、LaTeX のソースコードなど、人間が直接作成・修正するファイルは、バイナリ形式ではなくテキスト形式のファイルであることが多い。以下で見るように、`diff` や `patch` といった UNIX のツールは、その動作は非常にシンプルであるが、テキスト形式のファイルの差分を管理するには、非常にパワフルなツールである。

⇒ diff: 2 つのテキストファイルの差分を出力するコマンド
- ファイル全体を保存するよりコンパクト
- 変更点を確認しやすい

  $ diff -u file1.txt file2.txt > file.diff

⇒ patch: diff コマンドが生成した差分をファイルに適用するユーティリティー
- もとのファイルと差分から変更後のファイルを生成できる

  $ patch < file.diff

単一ファイルの例を示す。まず、ファイルのオリジナルのコピーを取り、エディタを使って内容を修正してみる。

```
$ cp /home/public/ce2015/ex2/prologue.txt prologue.txt
$ cp prologue.txt prologue-orig.txt
$ emacs prologue.txt
```

次に diff コマンドを用いて、差分を prologue.diff に出力する。

```
$ diff -u prologue-orig.txt prologue.txt > prologue.diff
```

less コマンドで中身を見てみると、例えば以下のようになっているはずである。

例 5.2.1.

```
$ less prologue.diff
--- prologue-orig.txt 2015-04-01 14:57:47.000000000 +0900
+++ prologue.txt 2015-04-01 14:58:57.000000000 +0900
@@ -5,6 +5,6 @@
 misadventured piteous overthrows Do with their death bury their
 parents' strife.  The fearful passage of their death-mark'd love, And
 the continuance of their parents' rage, Which, but their children's
-end, nought could remove, Is now the two hours' traffic of our stage;
+end, zero could remove, Is now the two hours' traffic of our stage;
 The which if you with patient ears attend, What here shall miss, our
 toil shall strive to mend.
```

diff コマンドでは、二つのファイルの差分が出力される。行頭の「-」は元ファイルからの削除を、「+」は追加を、つまり、「-」の行が「+」の行で置き換えられてたことを表している。

次に patch コマンドを使ってみよう。

```
$ cp /home/public/ce2015/ex2/prologue.txt prologue.txt
$ patch < prologue.diff
$ less prologue.txt
```

patch コマンドを使うことにより、先ほど作成した差分 (prologue.diff) をオリジナルのファイルに適用し、修正点を反映することができる。

単一ファイルだけではなく、以下のようにディレクトリ内のファイルをまとめて差分をとることも可能である。

```
$ cp -r /home/public/ce2015/ex2 shake
$ cp -r shake shake.orig
$ emacs shake/verona.txt
$ emacs shake/prologue.txt
$ diff -urN shake.orig shake > shake.diff
$ less shake.diff
```

この例では、shake.orig と shake に含まれている全てのファイルの差分が shake.diff に保存される。shake.diff を適用するには-p0 オプションを指定する。

```
$ rm -rf shake
$ cp -r /home/public/ce2015/ex2 shake
$ patch -p0 < shake.diff
```

このように diff と patch により、テキストファイルやそれらを含むディレクトリの差分をかなり容易に扱うことが可能になるが、依然として、何らかの方法で履歴の記録は別途管理しておかなければならないことに変わりはない。

5.3 主なバージョン管理システム

- ⇒ BitKeeper - かつて Linux のカーネルのソース管理に使われていた。
- ⇒ CVS (Concurrent Versions System) - ネットワークでの利用を考慮とした初めてのバージョン管理システム。最近はあまり使われない。
- ⇒ Git - Linux のカーネルの開発に使われている。分散型リポジトリ。最近利用が急速に伸びてきている。
- ⇒ Mercurial - Git のライバル。分散型リポジトリ。
- ⇒ SCCS (Source Code Control System) - 70 年代にベル研で開発された世界初のバージョン管理システム。現在では使われない。
- ⇒ Subversion - CVS の改良版として開発された。現在最もポピュラーなバージョン管理システムの一つ。Mac OS X や多くの Linux には最初からインストールされている。

本ハンドブックでは、Subversion について解説する。Subversion に関するより詳細な資料は、オンラインで参照できる。

- ⇒ CVS/Subversion を使ったバージョン管理（前編：バージョン管理の基礎)
  - http://sourceforge.jp/magazine/08/09/09/1038233
- ⇒ CVS/Subversion を使ったバージョン管理（後編：SVN を使ったバージョン管理)
  - http://sourceforge.jp/magazine/08/09/24/113215
- ⇒ Subversion によるバージョン管理
  - http://discypus.jp/svnbook/svnbook_ja_html
- ⇒「Subversion によるバージョン管理」の読み方
  - http://exa.phys.s.u-tokyo.ac.jp/ja/members/wistaria/log/subversion-intro

Subversion ではなく、Git を使いたい場合には、以下の資料を参考にすると良い。

- ⇒ CMSI ハンズオン - バージョン管理システム
  - http://www.cms-initiative.jp/ja/research-support/develop-support/how-to-publish/develop-apps/dt0133

5.4 Subversion リポジトリ

Subversion や他のバージョン管理システムでは、「リポジトリ」とよばれる「データベース」にソースコードの全ての履歴を保存する。このリポジトリからソースコードの最新版を引き出したり (チェックアウト)、リポジトリに修正版を登録 (チェックイン) したりすることになる。リポジトリの実体は、ハードディスク上のディレクトリ内に保存されている一連のファイルであるが、それらのファイルをユーザが直接触る機会はない。(むしろ直接触ってはいけない。) リポジトリへのアクセスは専用のコマンド (Subversion の場合、`svn` コマンド) を使って行う。

前述の通り、リポジトリにはネットワークを通じてアクセスすることが可能である。Subversion では、様々なアクセス方法が提供されているが、SSH (Secure Shell, 2.2節) を利用するのが便利である。リポジトリは、`svn+ssh://`**ユーザ名**`@`**ホスト名**`/`**リポジトリ名**のような書式で指定する。例:

```
svn+ssh://ce05151598@cmp.phys.s.u-tokyo.ac.jp/home/ce05151598/svnroot
```

このような書式は、URI (Uniform Resource Indicator) と呼ばれ、(ネット上の) 場所や名前を一意に表す方法として、広く使われている。例えば、web のアクセスで用いられる

https://wistaria@itc-lms.ecc.u-tokyo.ac.jp/lms/course/view.php?id=74564

のような形の URL (Uniform Resource Locator) も URI の一種である。ここで、`https:`の部分は「スキーム (scheme)」と呼ばれ、プロトコル・アクセス方法を指定する。その後ろに、ユーザ情報、ホスト名・サーバ名が続くが、この部分は「オーソリティ (authority)」と呼ばれる。ホスト名の後には、「パス」(`/lms/course/view.php` や「クエリ (query)」(`?iid=74564`) が続く。クエリは、サーバへの指示や命令を表す。

Subversion のリポジトリ名を表す URI も同様に、スキーム (`svn+ssh:`)、ユーザ情報、ホスト名・サーバ名、パスからなる。次節で見るように、パスは、サーバー上でリポジトリが置かれている実ディレクトリ名と、リポジトリ内の論理的なディレクトリ名が続けて書かれることに注意せよ。

5.5 Subversion 実習

以下、実習ワークステーション名を `cmp.phys.s.u-tokyo.ac.jp`、実習ワークステーションでのユーザ名を `ce00` と仮定して進める。実習では自分のユーザ名 (`ce05151598` 等) に適宜読み替えること。

5.5.1 リポジトリの作成

Subversion の管理用コマンド `svnadmin` を使い、実習用ワークステーション上の`$HOME/svnroot` にリポジトリを作成する。この作業はリポジトリ作成時に一回だけ行えば良い。作業は、実習用ワークステーションに SSH ログインして行う。

```
$ ssh cmp.phys.s.u-tokyo.ac.jp -l ce00
$ svnadmin create $HOME/svnroot
$ exit
```

以下では、iMac に戻り作業を続ける。

5.5.2 リポジトリ内のディレクトリ作成

リポジトリ内に `prog` という名前のディレクトリを作成する。Subversion のコマンド名 (`svn`) の後に、ディレクトリを作成するサブコマンド (`mkdir`) を指定する。

```
$ svn mkdir -m 'test folder' svn+ssh://ce00@cmp.phys.s.u-tokyo.ac.jp/home/ce00/svnroot/prog
ce00@cmp.phys.s.u-tokyo.ac.jp's password:
Committed revision 1.
```

コマンドを実行すると、実習用ワークステーションのパスワードを聞かれるので入力する[*1]。-m オプションの後の文字列はコミットログと呼ばれ、リポジトリにログとして記録される。なるべく分かりやすいログを残すのがよい。リポジトリ上に正しくディレクトリが作成されたかどうか確認してみよう。

```
$ svn ls svn+ssh://ce00@cmp.phys.s.u-tokyo.ac.jp/home/ce00/svnroot
prog/
```

5.5.3 リポジトリからのチェックアウト

prog ディレクトリをリポジトリからチェックアウトし、作業コピーを作成する。

```
$ cd $HOME
$ svn co svn+ssh://ce00@cmp.phys.s.u-tokyo.ac.jp/home/ce00/svnroot/prog
Checked out revision 1.
```

今、作業をしているディレクトリ内に空のディレクトリ prog が作成されていることを確認する[*2]。

```
$ ls -l
...
drwxr-xr-x  3 s001500 student      102 Apr 16 03:40 prog
...
```

5.5.4 ローカルファイルの作成とリポジトリへのチェックイン

エディタを使って、簡単なソースコード (hello.c) を作成する。

```
$ cd $HOME/prog
$ emacs hello.c
```

作成後、ファイルを Subverion の管理下に置く (Subversion で管理することを Subversion に伝える)。

```
$ svn add hello.c
A    hello.c
```

svn stat コマンドで状態を確認すると以下のように表示されるはずである。

```
$ svn stat
A    hello.c
```

行頭の「A」の記号は、ファイル hello.c が追加されたが、まだリポジトリには反映されていないことを示す[*3]。

それでは、今作成した hello.c をリポジトリにチェックイン (サーバーに送信) しよう。

[*1] svn co や svn update など、リポジトリとの通信をともなうコマンドの実行時には毎回パスワードを聞かれるが、以下では省略する。

[*2] 正確には、prog ディレクトリの中には .svn という名前のフォルダ (名前が.「ピリオド」で始まっているので、-a オプションを付けない限り、ls コマンドでは表示されない) が作成されている。このディレクトリには、Subversion の管理用データが収まっているので、決して、中身を消したり変更したりしてなならない。

[*3] 同じディレクトリに他のファイルやフォルダが存在するときには、ファイル名の前に「?」が表示される。これは、そのファイルやフォルダが Subversion の管理下にないことを示す。

```
$ svn ci -m 'Created my first program'
Adding    hello.c
Transmitting file data .
Committed revision 2.
```

無事チェックインされ、リビジョン番号 2 が付けられた。

5.5.5 別の場所での編集

先程チェックインしたバージョンを別の作業ディレクトリでチェックアウトしてみよう。

```
$ cd $HOME
$ svn co svn+ssh://ce00@cmp.phys.s.u-tokyo.ac.jp/home/ce00/svnroot/prog other
A    other/hello.c
Checked out revision 2.
```

`svn co` コマンドの第 2 引数に適当な名前 (ここでは `other`) を指定すると、その名前のフォルダの下にチェックアウトされる。ディレクトリ `other` の中には、`hello.c` が存在しているはずである。このファイルを少し編集してみよう。

```
$ cd $HOME/other
$ emacs hello.c
$ svn stat
M    hello.c
```

`svn stat` コマンドの出力の行頭の「`M`」の記号は、作業コピーが修正されており、その修正がまだリポジトリには反映されていないことを示す。それではチェックインしよう。

```
$ svn ci -m 'Fixed a serious bug'
Sending    hello.c
Transmitting file data .
Committed revision 3.
```

リビジョン 3 としてリポジトリに登録された。

さてここで、もとのフォルダ ($HOME/prog) に戻ってみると、(当然のことながら) `hello.c` は変更されていない。先ほど別のディレクトリからリポジトリに登録した修正を反映するには、`svn update` コマンドを使う。

```
$ cd $HOME/prog
$ svn update
Updating '.':
U    hello.c
Updated to revision 3.
```

このように、複数の場所 (複数の人) でファイルを更新していく場合、作業コピーのあるディレクトリで `svn update` ⇒編集作業 (修正・追加・削除) ⇒チェックイン (`svn ci`) を繰り返すことになる。

5.5.6 ファイルの削除・移動

ファイルを新たに Subversion の管理対象とするには、すでに見たとおり `svn add` コマンドを使う。すでに管理対象となっているファイルを削除するには `svn delte`、移動 (名前変更) するには `svn move`、複製するには `svn copy` コマンドを使う。`svn mkdir` コマンドにより作業コピーの下に新たにディレクトリを追加することもできる。これらのコマンドの後は、`svn ci` でリポジトリに変更点をチェックインする。

5.5.7 変更履歴の参照

ファイルの変更履歴を見るには、svn log コマンドを使う。

```
$ svn log hello.c
```

また、ファイルの差分を出力することもできる。最後のチェックアウト・アップデートからの作業コピーの修正点を見るには、

```
$ svn diff
```

を実行する。特定のリビジョンからの差分を見るには、-r オプションでリビジョン番号を指定する。

```
$ svn diff -r 1
```

二つのリビジョンの間の差分を見ることもできる。

```
$ svn diff -r 1:2
```

svn annotate コマンドを使うと、ファイルのそれぞれの行が、どのリビジョンの時点で誰によって修正されたかを見ることができる。

```
$ svn annotate hello.c
```

さらに詳しくは、

⇒「Subversion によるバージョン管理」– 履歴の確認
http://discypus.jp/svnbook/svnbook_ja_html/ch03s06.html

を参照のこと。

5.5.8 リポジトリ・作業コピーについての情報の取得

作業ディレクトリ内で svn info コマンドを実行することにより、リポジトリや作業コピーについての情報を取得することができる。

```
$ cd $HOME/prog
$ svn info
Path: .
Working Copy Root Path: ...
URL: svn+ssh://ce00@cmp.phys.s.u-tokyo.ac.jp/home/ce00/svnroot/prog
Relative URL: ^/prog
Repository Root: svn+ssh://ce00@cmp.phys.s.u-tokyo.ac.jp/home/ce00/svnroot/
Repository UUID: d30291ec-23e1-4ba7-bfc0-364b1455df9a
Revision: 7
Node Kind: directory
Schedule: normal
Last Changed Author: ce00
Last Changed Rev: 7
Last Changed Date: 2015-04-01 19:52:28 +0900 (Wed, 1 Apr 2015)
```

5.5.9 コンフリクト

チェックインしようとしたファイルがすでに他の人 (or 他の場所) により更新されている場合、チェックインはエラーとなる。以下、$HOME/prog と$HOME/other で同時に hello.c を編集することで、わざとコンフリクトを発生させてみよう。

まずは、$HOME/prog と$HOME/other の両方のディレクトリで最新版にアップデートしておく。

```
$ cd $HOME/prog
$ svn update
$ cd $HOME/other
$ svn update
```

次に、$HOME/prog/hello.c を修正し、チェックインする。

```
$ cd $HOME/prog
$ emacs hello.c
$ svn diff
Index: hello.c
===================================================================
--- hello.c    (revision 5)
+++ hello.c    (working copy)
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main() {
-  printf("> %lf\n", 10.0);
+  printf("> %lf\n", 1.0);
   return 0;
 }
$ svn ci -m 'Change value'
```

次に、$HOME/other/hello.c の同じ行を異なる値に修正し、チェックインしてみよう。

```
$ cd $HOME/other
$ emacs hello.c
$ svn diff
Index: hello.c
===================================================================
--- hello.c    (revision 5)
+++ hello.c    (working copy)
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main() {
-  printf("> %lf\n", 10.0);
+  printf("> %lf\n", 2.0);
   return 0;
 }
$ svn ci -m 'Change value, too'
Sending    hello.c
Transmitting file data .svn:  E160028:  Commit failed (details follow):
svn:  E160028:  ファイル '/prog/hello.c' はリポジトリ側と比べて古くなっています
```

このようにチェックインは失敗する。この問題は以下のような手順で解決することができる。

まず、$HOME/other で svn update を実行する。

```
$ cd $HOME/other
$ svn update
Updating '.':
C    hello.c Updated to revision 6.
Conflict discovered in file 'hello.c'.
Select:  (p) postpone, (df) show diff, (e) edit file, (m) merge,
   (mc) my side of conflict, (tc) their side of conflict,
   (s) show all options:  p
Summary of conflicts:
  Text conflicts:  1
```

Subversion がコンフリクト (変更の衝突) を発見し、解決策を訪ねてくるので、ここでは `p` (postpone) と答える[*4]。作業ディレクトリには、`hello.c` 以外に、`hello.c.mine`、`hello.c.r5`、`hello.c.r6` などのファイルが作成されているはずである。それぞれ、先ほど修正をチェックインしようとした内容、修正の元となったリビジョン (この例ではリビジョン 5)、リポジトリに登録されている最新リビジョン (この例ではリビジョン 6) のファイルとなっている。また、`hello.c` の中を見てみると、どこがどのようにコンフリクトしているかが書き込まれている。

```
$ cat hello.c
#include <stdio.h>
int main() {
<<<<<<< .mine
  printf("> %lf\n", 2.0);
=======
  printf("> %lf\n", 1.0);
>>>>>>> .r6
  return 0;
}
```

ここで「`<<<<<<< .mine`」から「`=======`」までが、作業コピーの変更点、「`=======`」から「`>>>>>> .r6`」までが、リポジトリに登録済みの変更点である。`hello.c.mine`、`hello.c.r5`、`hello.c.r6` を参照しながら `hello.c` を納得のいくまで修正する。ここでは、`hello.c.mine` の内容を採用することにする。

```
$ emacs hello.c
$ cat hello.c
#include <stdio.h>
int main() {
  printf("> %lf\n", 2.0);
  return 0;
}
```

編集が完了したら、`svn resolved` コマンドで Subversion にコンフリクトが解消されたことを知らせ、チェックインする。

```
$ svn resolved hello.c
Resolved conflicted state of 'hello.c'
$ svn ci -m 'Merge r5 and r6'
Sending    hello.c
Transmitting file data .
Committed revision 7.
```

無事、コンフリクトが解消され、リポジトリに反映された。さらに詳しくは、

[*4] 同じファイルが同時に変更されている場合でも、変更点が互いに離れた行である場合には、Subversion は修正点を自動でマージしてくれる。

⇒「Subversion によるバージョン管理」– 競合の解消 (他の人の変更点のマージ)

`http://discypus.jp/svnbook/svnbook_ja_html/ch03s05.html#svn.tour.cycle.resolve`

を参照のこと。