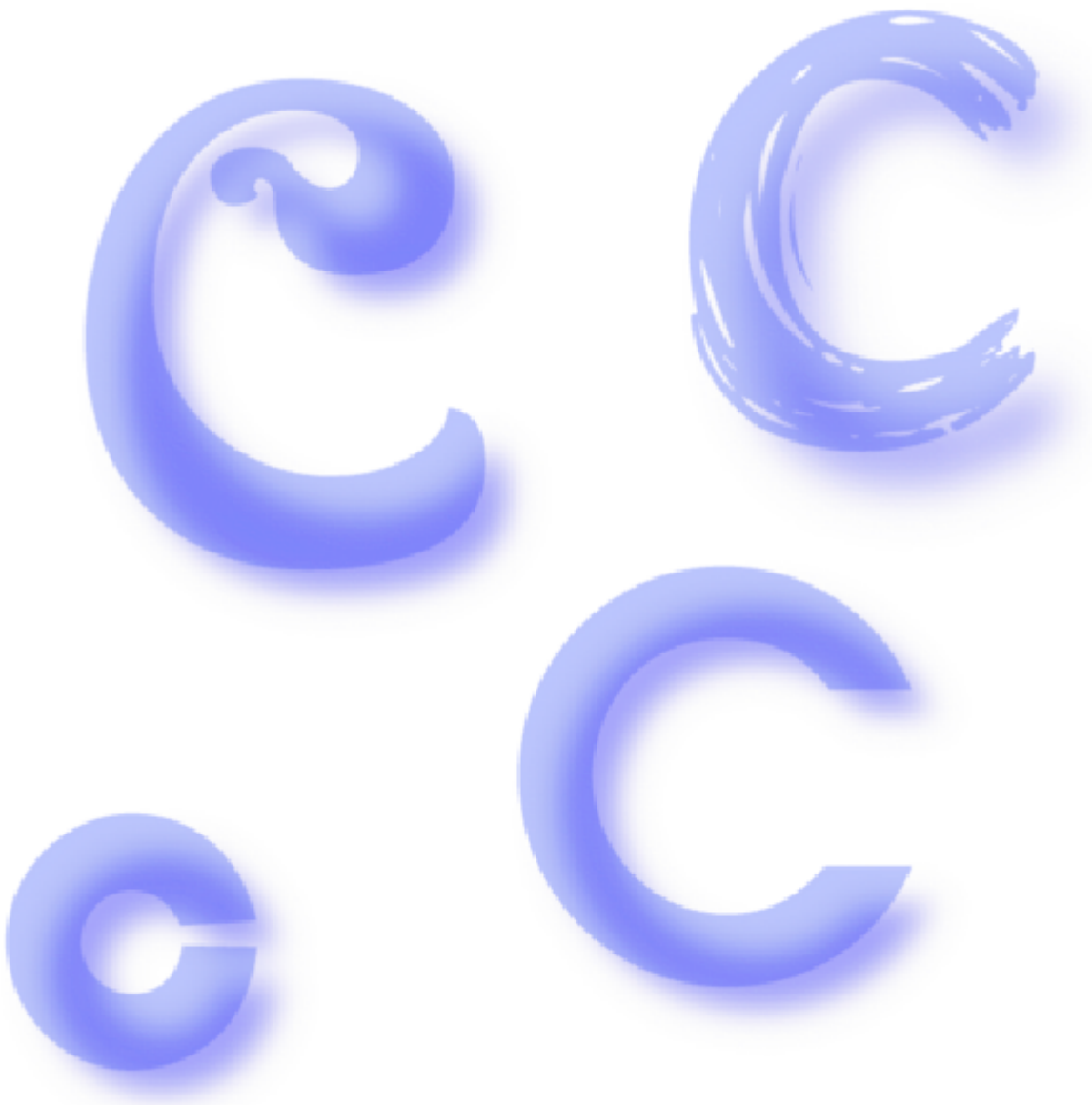


早稲田大学 理工学術院  
情報関連科目

# Cプログラミング入門



木曜日 1 限  
2018年度

# 目次

<b>1 はじめてのプログラミング</b>	<b>5</b>	<b>6 関数</b>	<b>27</b>
1.1 ソースの編集と実行ファイルの作成	5	6.1 関数宣言と関数定義	27
1.2 具体的手順	6	6.2 関数の呼出し	27
1.3 四則計算結果の表示	7	6.3 引数のない関数	28
<b>2 変数と型</b>	<b>8</b>	6.4 アドレス渡し	28
2.1 識別子	8	6.4.1 配列を渡す	29
2.1.1 予約語	8	6.4.2 変数のアドレスを渡す	29
2.2 基本型	9	6.5 再帰関数	30
2.2.1 整数型	9	6.6 標準数学関数	31
2.2.2 浮動小数点型	9	<b>7 文字列</b>	<b>33</b>
2.2.3 void 型	10	7.1 文字列: char 型の配列	33
2.3 書式付き出力: printf()	10	7.2 文字列の操作	34
2.4 main 関数への引数	11	7.2.1 文字列の長さ	34
<b>3 条件分岐</b>	<b>13</b>	7.2.2 文字列の複写 (copy, duplicate)	34
3.1 真偽値	13	7.2.3 文字列の比較 (compare)	34
3.2 関係演算子 (relational operator)	13	7.2.4 文字列の連結 (concatenate)	36
3.3 論理演算子 (logical operator)	13	7.2.5 文字列中の文字の検索	36
3.4 条件分岐命令	13	7.3 文字の入出力	36
3.4.1 if...else	13	7.4 ライン入出力	36
3.5 switch...case	15	<b>8 ポインタ</b>	<b>38</b>
3.6 条件演算子	15	8.1 定数ポインタ	38
<b>4 繰り返し (loop)</b>	<b>17</b>	8.2 ポインタ変数	38
4.1 while 文	17	8.2.1 ポインタ, 配列, 関数の優先順位	38
4.1.1 while	17	8.3 ポインタと配列名	38
4.1.2 do...while	18	8.3.1 配列名が先頭要素へのポインタでない場合	39
4.2 for 文	18	8.4 ポインタ変数の初期化	39
4.2.1 多重ループ	18	8.4.1 関数へのアドレス渡し	39
4.3 無条件分岐	19	8.5 ポインタの演算と配列の添字演算子	40
4.3.1 break	19	8.5.1 ポインタの演算	41
4.3.2 continue	19	8.6 ポインタの配列	41
4.3.3 goto	20	<b>9 構造体</b>	<b>42</b>
4.3.4 return	20	9.1 構造体の宣言, 初期化	42
4.4 実数によるループ制御	20	9.2 構造体の代入	42
<b>5 配列</b>	<b>22</b>	9.3 メンバへの個別アクセス	43
5.1 配列要素への値の代入	22	9.3.1 単純に宣言した場合	43
5.2 配列の宣言時の初期化	22	9.3.2 ポインタ型の場合	43
5.2.1 要素数を省略した初期化	22	9.4 typedef	44
5.3 多次元配列	23	9.5 3次元ベクトルへの応用	44
5.4 多次元配列と1次元配列	24	9.6 構造体の応用例: struct tm	45
5.5 配列の応用	24	<b>10 プリプロセッサ命令</b>	<b>46</b>
5.5.1 素数を求めるプログラム	24	10.1 #include	46
5.5.2 頻度分布への応用	25	10.2 #define	46

<b>11 ファイル入出力</b>	<b>47</b>
11.1 FILE 構造体へのポインタ取得 : <code>fopen()</code>	47
11.2 ファイルのクローズ: <code>fclose()</code>	48
11.3 文字および文字列の入出力関数	48
11.4 出力バッファの掃き出し : <code>fflush()</code>	48
11.5 内部表現のままのデータ保存	48
11.6 内部表現のままの入出力関数 : <code>fread()</code> , <code>fwrite()</code>	49
<b>12 プロセス制御と <code>system()</code></b>	<b>49</b>
<b>13 複素数の計算 C99</b>	<b>50</b>
13.1 <code>complex.h</code> と <code>tgmath.h</code>	50
<b>14 アルゴリズムの学習</b>	<b>51</b>
14.1 整列 (sorting)	51
14.1.1 単純選択ソート	52
14.1.2 バブルソート	52
14.1.3 クイックソート	53
<b>付録</b>	<b>55</b>
<b>A 文法の補足</b>	<b>55</b>
A.1 演算子	55
A.1.1 優先度について	55
<b>B プログラムの図式化</b>	<b>56</b>
B.1 フローチャート	57
B.2 PAD	57
B.2.1 <code>pad2ps</code>	57
<b>C Linux および UNIX</b>	<b>58</b>
C.1 Login/Logout	58
C.1.1 <code>login</code>	58
C.1.2 <code>logout</code>	58
C.1.3 シェル	58
C.2 ディレクトリとファイル	58
C.2.1 木構造とリンク	58
C.2.2 カレント, ホーム	59
C.2.3 絶対パス, 相対パス	59
C.3 UNIX の基本コマンド	59
C.3.1 コマンドの書式	60
C.3.2 ディレクトリに関するコマンド	60
C.3.3 ファイルに関するコマンド	61
C.3.4 テキスト整形	62
C.4 リダイレクトとパイプ : <code>&gt;</code> <code>&gt;&gt;</code> <code> </code>	63
C.5 知っている便利なコマンド	64
C.5.1 <code>indent</code>	64
C.5.2 <code>grep</code>	64

C.5.3 <code>diff</code>	64
C.5.4 <code>date</code>	64
C.5.5 <code>sort</code>	65
C.6 ファイルの保管	65
C.6.1 <code>tar</code>	65
C.6.2 <code>gzip</code> , <code>gunzip</code>	65
C.6.3 <code>bzip2</code> , <code>bunzip2</code>	65
C.6.4 <code>compress</code> , <code>uncompress</code>	66
C.6.5 <code>lha</code> , <code>zip</code>	66
C.6.6 <code>mttools</code>	66
C.6.7 <code>split</code>	66
C.6.8 <code>uuencode</code> , <code>uudecode</code>	66
C.7 <code>bc</code>	67

## 索引

69

## はじめに

C 言語は現在最も普及しているプログラミング言語であり、今後も使われ続けていくと思われます。UNIX という歴史あるコンピュータの OS (Operating system) が C 言語で開発されたという事実はあまりにも有名です。また、ほぼ全てのプログラミング言語についても、それ自身は C 言語で書かれているということからも (C 言語も当然 C 言語で書かれています)、学習すべき最も重要で基本的なプログラミング言語といえましょう。

**参考書** 以上のような事情から、C 言語に関しては多くの書籍が出版されており、入門レベルでは、どの一冊を選んでもあまり差はないでしょう。最後まで読み続けられそうな、自分に合った書籍を手許に置いてください。あえて参考書をあげるとすると、

- 柴田望洋 著: "明解 C 言語" (ソフトバンクパブリッシング, 1999)

は、「**分かりやすい C 言語教科書・参考書の執筆の業績が認められ、(社) 日本工学教育協会より著作賞を授与された。**」(裏帯より引用) 評判の高い書籍です。多色刷のレイアウトが却って五月蠅いとみるむきもありますが、大変丁寧な説明があつて、持っていて良い一冊と思います。

**インターネット上の役立ちサイト** また、インターネットで「C 言語入門」をキーワードに検索してみると多くのサイトを見つけることができます。検索結果の上位にランクされている人気サイトは、(例外もありますが) かなり良くできていて、そこを利用しても十分な学習成果をあげることができそうです。以下に、役立ちそうなサイトを列挙しますので、訪れてみてください。

### ☞ 参考ウェブサイト: C 言語入門

- **初心者のためのポイント学習 C 言語**「ドリル&ゼミナール C 言語」という入門書の執筆者が運営する読みやすく親切的なページです。
- **C 言語学習塾**: オーキッドという会社が運営するページ。Study C という製品の販売の一環として運営されているようです。しっかりしています。(私はオーキッドとは何の関係もありません、念のため)。
- **金山典世「C 言語」(稚内北星ビブليون)**: 稚内北星学園短期大学の公開図書 (1998 年) です。情報専門の大学の教材だけあって非常にきちんとしています。
- **明解 C 言語入門編**: ドクターボーヨーで有名な柴田望洋さんが著した、人気の高い入門書の宣伝ページです。カラフルな本の一部を見ることができます。
- **C プログラミング入門**: 東京高専情報工学科小坂先生のページ。

- **The C Programming Language, Second Edition**: C 言語の開発者 B. W. Kernighan (カーニハン) 博士と Dennis M. Ritchie (リッチー) の著書。C 言語のバイブルとも言われますが、初心者にはやや難しいかもしれません。通称「K&R 本」ともいわれ、この本を知らない人モグリであるいわれてしまうようです。
- **Dennis M. Ritchie**: C 言語の本当の産みの親、リッチー博士のホームページ。C 言語が誕生するまでの歴史に関する文献を読むことができます (ただし、英語)。

**学習方法** 語学の学習に似て、「習うより馴れろ」という方法でも覚えられます。しかし、ただ闇雲に手本を真似るだけでは上達は望めません。自然言語よりもずっと厳格な文法規則を持っているプログラミング言語では、用法に関する的確な例題を少し勉強するだけで驚くほど理解が進みます。したがって、半期 12 回程度の講義でも、ほぼ全体像を掴むことは困難ではありません。

## 本文中の記号について

本文に現れる記号の説明以下に記します。常識的なものですが、その意図を良く理解して効果的に学習を進めてください。

- **例題 n** で始まる題材は本講義の中核をなすもので必ず行ってください。
- **問題 n** は、例題を発展させた、あるいは例題を補ぎなう題材です。実力アップの効果を狙っています。
- ☞ で始まる段落は、「ポイントですよ」という意味で、C 言語の文法説明やインターネットのリンク一覧を記しています。
- ☝ は注意書きです。細かいこと (実際活字も小さいです) が気になる性格の方は読んでみてください。
- ✕ は、初心者が陥りやすい間違いを強調します。
- ex1.c<sup>☞</sup> のように表示されているファイルはダウンロードできます。講義の後半ではプログラムが複雑化し、ソースファイルも長くなります。そこで、学習するポイントに関連する部分のみ各自で追記する形式のソースファイルを用意しました。
- 以下のような囲みを用いて各回毎にまとめを書くように努力します。

### まとめ

## 課題の提出について

成績評価を左右する課題は、メールで提出してもらいます。皆さんのなかには既に自分に合ったメールを使いこなしている人もいるでしょうが、この講義では大学が提供する Web メールサービス を利用してください。

**Web メールサービス以外からは受け付けません**

ので、注意してください。

## Web メールサービスの使い方

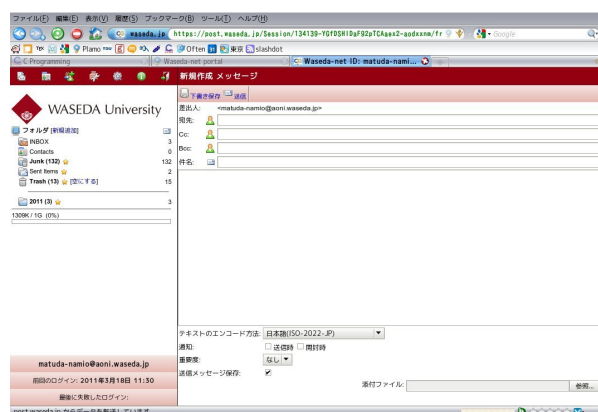


図 1 Web メールサービスのメッセージ作成画面

Web メールサービスは直観的なインターフェースを持った分かりやすいメールシステムです。IE や Firefox などといった WWW ブラウザを用いて、フォーム入力やボタン操作だけでメールを読み書きできるので、誰でもすぐに使えるようになります。したがって、解説は不要かもしれませんが、少しだけ注意しておきます。

**例題 1** Web メールサービス上の自分のメールアドレスを宛先にして、適当なメールを出しなさい。また、そのメールが受信されることを確かめなさい。

日本語入力は、**Ctrl + \** (+ は同時にキーを押すという意味です) でトグル切替えとなっています。これは、xwnmo という日本語入力クライアントの場合です。講義では Xemacs というエディタを使いますが、Xemacs は独自の日本語入力システムを持っています。したがって、日本語変換入力の方法が xwnmo とは異なる場合がありますので、注意が必要です。

## 提出時の注意点

Web メールサービスは直観で操作して間違えることはないのですが、自分でいろいろ試してください。課題提出に関して、以下に注意をまとめますので厳守してください。

- **件名に課題番号を記す**：内容を見れば判断はできませんが、必ず課題番号を記してください。
- **送信控を保存をする**：不幸にもメールが届かない事故も考えられます。したがって送信を行った証拠を残しておいてください。

## 日本語テキストファイルについて

### 漢字コード

文字をデータで表すことを符号化といいます。欧米では日常使われる文字種（数字とアルファベットと記号）が少なく、C 言語でいえば char (1 バイト=8 ビット) の範囲、すなわち 256 個で十分間に合います。実際には、ASCII コードという 7 ビット (“man ascii” としてみてください) の符号化コードが現在でも良く使われています。日本語は平仮名、片仮名だけならばなんとかなるかもしれませんが、漢字を符号化するには日常使われる範囲でも 2 バイト、大漢和辞典に収録されているように十万を越える文字を表現するには、3 バイト以上が必要になります。日本規格協会 (JIS) が日本語の符号化を定めた当時、通信手続きなどの問題から 8 ビットを使うことができないことを考慮して、7 ビットのデータ列で表現する規約を定めました。それが JIS コードです。従って、公には 7 ビット JIS を用いることが推奨されるのですが（電子メールを送信する際に使うことができる漢字コードは、公の合意では今もって 7 ビット JIS です）、コンピュータ内部では 8 ビットを用いた符号化が用いられるようになりました。Linux などの UNIX 互換 OS では、日本語拡張ユニックスコード (Enhanced Unix Code)、MS-DOS/Win や MAC ではシフト JIS (MS 漢字コードともよばれる) がデフォルトの漢字コードとなっています。

**例題 2** システムの使用言語を以下のコマンドで確かめなさい。

```
echo $LANG
```

### 漢字コードの変換

従って日本語を扱う場合には、各 OS のデフォルトの漢字コードを用いるのが安全です。UNIX 互換 OS はテキストを扱うツールが充実しており、漢字コード変換に関しては、**nkf** (ネットワーク用漢字コードフィルタ)、**lv** などの定番のツールがあります。しかしながら、標準でインストールされていない場合には、実質的な UNIX の国際標準規格である **POSIX 規格** (狭義には、**IEEE Std 1003.1** を指す) に採用されたインターフェースを備える **iconv** を利用します。



Unix 互換システムで現在 UTF-8 が採用されている場合が多いのですが、MS-Windows ではユーザー漢字コードとして Shift-JIS が使われていますから、MS-Windows のファイルが文字化けして読めない場合には、

```
iconv -f SJIS -t UTF8 filename > filename.u8
mv filename.u8 filename
```

などとして、UTF-8 漢字コードのファイルに変換します。iconv は標準出力（一般には画面）に変換結果を出力しますから、それを一端適当な名前のファイルに書き込み、それから、元のファイルに上書きするという手順が必要となります。面倒だからといって

```
ダメな例: iconv -f SJIS -t UTF8 filename
> filename
```

としないでください、中身がまったくなくなってしまいます。

**nkf** もし nkf がインストールされている場合には、iconv がない便利な機能がありますから、また Win32 版もありますからそれを使うことを奨めます。“man nkf”で概要が分かりますが、UTF-8 に変換するには

```
nkf -w filename > filename.u8
mv filename.u8 filename
```

などとし、nkf は入力ファイルの漢字コードを自動判別してくれます。従って、ファイルの漢字コードを検出することにも使えます。オプション -g（あるいは -guess）を付けて実行します。また、ファイルの上書きもサポートしていますので、便利です。すなわち、

```
nkf -w --overwrite filename
```

だけで済ませることが可能です。

🔗 **UNICODE**: 欧米の文字は 255 で足りるといいましたが、実情はそう甘くありません。例えばギリシャ文字を考えてください、ギリシャ文字はもちろん今でも使われていますから、これとアルファベットを同じコード表に載せることは難しいことです。日本語を含めて、言語別にコード表を切替えて（エスケープシーケンス）使うという方法が現在の主流ですが、そうではなく、全言語の文字を一つの大きな表（例えば 4 バイトなら  $2^{32} \sim 40$  億の文字種）に載せてしまおうという考え方は自然な発想でしょう。もちろん ascii だったら 1 バイトなのを 4 バイトで表現すれば、ファイルの大きさは 4 倍、処理も遅くなりますから、単純な実装方法では明かに非効率です。したがって、ascii は 1 バイトで済むように工夫された実装方法 (UTF8) が MS-Win の日本語の内部処理には既に使われており、漢字言語圏以外の国ではその方向で多言語化がすすんでいます。ただし、表の作成すなわちコード化の作業が、非漢字言語圏の人達によって行われ、日本などの漢字言語圏の関係者の意見をあまり聞かないうちに強行されてしまったことが禍して、中国と日本語を同時に表示

することが困難な状況になってしまいました。また、S-JIS や EUC-J は JIS を基にしているので漢字の並び順は保存されており、相互変換が容易です。ところが現在の UNICODE 表では、漢字の並び順について JIS は考慮されていません（漢字言語圏を一緒にしてしまったのですから当然です）。したがって、UTF8 に変換するには、非常に大きな変換表を参照するしか方法がなく、処理速度がかなり低下します。以上のような問題があるので中国ではこのようなコードを用いないことを政府が決定しています。いろいろ問題がありますが、基本的な発想は十分合意できるものなので、将来的には、全世界の言語が 1 つのコード表に載ることが実現するでしょう。

## 🔗 参考ウェブサイト：ユニコード

- [ウィキペディア：Unicode](#)
- [日本の Linux 情報（アーカイブ）：Unicode-Howto.txt](#)
- [IT 用語辞典 e-Words：ユニコード](#)

lv はそういった現時点での UNICODE の実装法の一つ UTF8 をも含めた、テキストビューアです。JIS, EUC-J, S-JIS, UTF8 の間で自由に漢字コードを変換することができます。“man lv”として使い方を調べてみてください。

表 1 OS に依存する漢字コードと行末コード

OS	漢字コード	行末コード
Linux	日本語 EUC (EUC-J)	LF
MS-Win	シフト JIS (S-JIS)	CR+LF
古い MAC	シフト JIS	CR

**例題 3** テキストファイルの漢字コードを使用中のシステム以外のものに変換した後、端末に表示させてみなさい。  
[ex03.txt](#)

## 行末コード

漢字コード以前にテキストファイルの行末コードが、各 OS で異なるということも問題となることがあります。表 1 に代表的な 3 つの OS の漢字コードと行末コードを示します。ラインフィード (LF) およびキャリッジリターン (CR) はプリンタの制御命令で、それぞれ改行（行送り）、（行頭への）復帰を意味します。

## 行末コードの変換

もちろん、漢字コードと同じく行末コードも OS に合わせて変換するのが安全です。unix2dos, dos2unix というその名通りのツールがありますから使ってみましょう。MS-DOS/Win から UNIX へのテキスト変換、すなわち **テキストファイルの行末コード CR+LF を UNIX の行末コード LF に変換するには**、

```
dos2unix file
```

とします、逆に UNIX から DOS へのテキスト行末コードの変換は

```
unix2dos file
```

専用ツールがない場合の対処法として、現在人気の高いスクリプト言語 perl を用いた方法を紹介し、MS-DOS/Win のテキストファイルの行末コード CR+LF を Linux の行末コード LF に変換するには、CR を無文字に変換すると考えて

```
perl -p -e "s/\r//" <dosfile >unixfile
```


逆に LF から CR + LF に変換するには以下のようにします。

```
perl -p -e "s/\n/\r\n/" <unixfile >dosfile
```

④ 以上の変換ツールはテキストファイルに限ってのことで、バイナリファイルには実行しないでください。あるファイルのあるデータが漢字コード、行末コードという意味を持つのは、テキストファイルとして扱う場合に限るからです。一般に、ファイルがどのような種類のものであるかを判断できるようにするため、例えば先頭に決まったコードを書くように定めることがあります。画像ファイルの多くはそのような規約に従うものが多いです。テキストファイルについてはそのような規約はありません。したがってあるファイルがテキストであるかどうかは、それをテキストとみただけではある程度処理を行ってみる必要があります。nkf などではそのような程度処理をして漢字コードを自動判別します。

なお、UNIX には“file”という、ファイルの種類を判別するツールがあります。これはかなり強力な /etc/magic というファイルに、評価されるファイルの種類が登録されています。実行例を以下に示します。

```
$ file ex0
ex0: ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked, not stripped
$ file ex0.c
ex0.c: C program text
$ file text00.tex
text00.tex: International language text
```

④ 実行例中の  は、**[Enter]** キーを押して入力を行ったことを意味します。

## 漢字コードと行末コードの同時変換

バージョン 1.9 以降の nkf には、漢字コードと行末コードを同時に変換するオプションができました。例えば UNIX 形式に変換するには

```
nkf --unix dosfile > unixfile
```

とします。さらに、元ファイル (MS-DOS/Win 形式) を UNIX 形式で上書きするには

```
nkf --unix --overwrite file
```

とします。iconv には改行コード変換の機能はありません。

**例題 4** 例題 3 で用いたファイルを適当に変換し、行末コードの違いを次のようなコマンドで確かめなさい (バイナリダンプと言います)。

```
od -c ファイル
```

```
hexdump -c ファイル
```

**[解]** ex03.txt (SHIFT-JIS) を、そのまま、UTF-8、EUC-JP 変換した後のそれぞれのバイナリダンプの様子を示します (紙面の都合で左側の位置情報表示を割愛しています)。

```
$ od -c ex03.txt
j  a  p  a  n  e  s  e      223 372 226 { 214 352 \r
\n
$ iconv -f SJIS -t UTF8 ex03.txt > ex03.u8
$ od -c ex03.u8
j  a  p  a  n  e  s  e      346 227 245 346 234 254 350
252 236 \r \n
$ iconv -f SJIS -t EUCJP ex03.txt > ex03.e
$ od -a ex03.e
j  a  p  a  n  e  s  e      306 374 313 334 270 354 \r
\n
$ dos2unix ex03.u8
dos2unix: converting file ex03.u8 to UNIX format ..
$ od -c ex03.u8
j  a  p  a  n  e  s  e      346 227 245 346 234 254 350
252 236 \n
```

## まとめ

- 漢字コードは現在 Unicode 化が進んでいるが、JIS, Shift-JIS, 拡張 EUC を用いた過去の遺産もあることに留意する。
- Linux と MS-DOS/Win では日本語テキストの漢字コードや行末コードが異なるので、漢字変換ツール iconv などに変換する。

# 1 はじめてのプログラミング

## 1.1 ソースの編集と実行ファイルの作成

C 言語ではソースファイルをエディターで書いて、コンパイラー（ここでは GNU の gcc を用います）でコンパイルを行い、**実行ファイル**を作ります。具体的には、foo.c という名前のソースファイルから bar という名前の実行ファイルを作成するには、以下のようにします。

```
$ gcc foo.c -o bar -lm
```

-o のあとに実行ファイル名を指定します。そうしないと a.out という名前の実行ファイルが作成されます。-lm は数学関数ライブラリを利用する場合に必要な（リンク）オプションです。この他にも、オプション -Wall および -Wstrict-prototypes を付けると、プログラムが正しく書かれているかどうかを厳しくかつ細かく警告してくれるので、C 言語の理解に役立ちます。

なお C 言語のソースファイルの拡張子は一般に小文字の .c としてください。後々学習するかもしれませんが、大文字の .C や .cc、.cpp があるソースは、c++ のものと自動的に判断されるのが普通です。

🔗 Command を逐次実行して確かめながらプログラムを作成する言語は、**インタプリタ**と呼ばれ（代表は BASIC です）、初心者に向いています。しかし、実行速度が遅いことや大規模プログラミングに向いていないなどの理由で、敬遠されがちです。

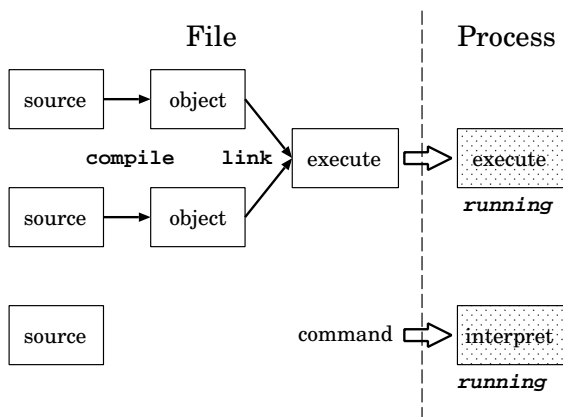


図2 コンパイル言語（上）とインタプリタ言語（下）の実行までの概要

**例題 5 コンソール出力** 『Hello Wold! の表示』は非常に有名なプログラムです。次の手順でプログラムを作成・実行しなさい。

- (i) リスト1のソースを gedit などのエディタで編集して、ex02.c という名前で保存。
- (ii) gcc でコンパイルして、hello という名前の実行ファイルを作成。

- (iii) コマンドラインから ./hello と入力して実行。

### リスト1 ex02.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int
6 main (int argc, char **argv)
7 {
8     printf ("Hello World!\n");
9     return (0);
10 }
```

なお、リストの各行頭の数字とコロンは、説明のために紙面上で付けたものです。実際には**入力しない**ように。

C 言語のソースには**関数**（と呼ばれる機能を発揮する上での命令文の集まり）の定義と実行手順が記述されます。最初の3行は、いろいろな標準関数が見えるように定義ファイルを読み込むよう指示するもので“定番”と考えてください。実行時には、main 関数が最初に動きます。その中から他の関数が次々に呼び出 (call) されます。ところで関数は次のような構文に従って定義されます。

関数名（引数のリスト）

```
{
    ...
    定義
    ...
}
```

main も関数ですから 5,6,9,10 行目が型通りの記述で、定義内容は 8 行目だけです。

さて、その内容ですが、書式付き出力 printf("...") を用いて**標準出力**（一般的には**ディスプレイ**）にデータを表示しています。基本的には、二重引用符（ダブルクォーテーション）で挟まれた部分の文字列がそのまま出力されます。しかし、画面制御文字などはもともと対応する**可視文字**がないので（そういう意味で**不可視文字**と呼ばれます）頻度の高いものについては、\ で始まる記号で表現します。例題中の \n がその例で**改行**（すなわち、左端にカーソルが復帰して行が送られる）を意味します。その他に**タブ**を意味する \t もカラム (column) をそろえる場合によく使います。なお、可視文字は \020 (8 進数) や \x0d (16 進数) のようにコードで表現することもできますが、可読性に劣るので普通使いません。これらは C 言語の printf 関数に限らず、文字に関する慣用的な規約です。もう一つは、% で始まるもので、**変換子**と呼ばれ、そこに数値や文字列が置き換わって表示されます。

第 8, 9 行目がセミコロン“;”で終了しています。これは大変重要で、命令文の終わりを示しています。



## 1.2 具体的手順

Gedit によるファイルの編集と X 上の端末エミュレータ (gnome-term や kterm, rxvt *etc.* の総称、以降 *\*term* と略します) 上での gcc の実行を具体的に説明しましょう。

(i) *\*term* 上で

```
$ gedit & ← & を必ずつけてください
```

とキー入力します。すると図のように gedit が画面に現れます。

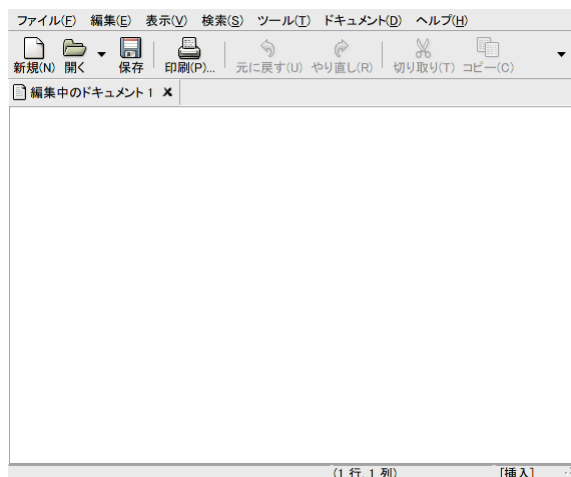


図 3 Gedit の起動画面

(ii) 編集の対象となるファイルを選択します。そのためにツールバーにある「開く」をクリックしてください、図のようにファイルの選択画面が現れます。新しいファイルを編集する場合は、ツールバーにある「新規」をクリックしてください（新規の場合には保存時に名前を付ける必要があります）。



図 4 Gedit のファイル選択画面

(iii) リスト 1 と同じ内容 (行頭の番号は入力しないこと) を

作成します。

(iv) ファイルを保存します。ツールバーの「保存」をクリックしてください。

(v) Gedit を起動した *\*term* に

```
samba02:~$
```

のようなプロンプトが最下行に表示されていれば、コマンドを受け付ける状態にありますから、C 言語のソースのコンパイル命令をキー入力します。以下、**シェルのプロンプトを単に \$ と略記します**。

```
$ gcc ex02.c -o hello -lm
```

エラーメッセージが表示されなければ成功です。

⚠ Unix は**寡黙な OS**と呼ばれます。何かの間違いがあった場合にはエラーがあったと知らせてくれますが、うまくいっている場合にはわざわざその旨を知らせてくれないのが普通だからです。

(vi) 実行ファイルが出来ているか `ls (list)` コマンドで確かめましょう。

```
$ ls hello
hello
```

この場合 `hello` があれば、単にその名前が表示されます。

(vii) `hello` は実行ファイルです。実行ファイルを実行させるには**名前をキー入力**します。ただし、ファイルの所在をはっきり示すために、カレントディレクトリを表す `./` を先頭につけてください。すなわち、

```
$ ./hello
```

と *\*term* 上でキー入力しなければなりません。

⚠ 通常 Unix では安全 (secure) であることが保証されているディレクトリにある実行ファイルしか起動しないように制限がかけられています。これは `PATH` という環境変数で設定するのが普通です。頻繁に使うプログラムと同名の危険なプログラムが含まれていても、`PATH` になければ実行されません。自分で管理していないディレクトリであってもそこに移ってしまえば、カレントとなるわけですから、カレントディレクトリ `./` を `PATH` に含めるのは非常に危険です。カレントにあるファイルは、(安全であるを) 確認して実行させる必要があります。

**問題 1** (水平) タブ `“\t”`, 垂直タブ `“\v”`, 復帰 `“\r”` を使った表示プログラムを作成して、それらの働きを確かめなさい。例えば、以下のようなプログラムが考えられます。

リスト 2 q01.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
```

```

6 {
7     printf("\tTAB\tTAB\tTAB\tTAB\n");
8     printf("\tHT\vVT\tHT\vVT\n");
9     printf("CR\rCR\rCR\rCR\r\n");
10
11     return (0);
12 }

```

**問題 2** 次の画面を出力するプログラムを考えなさい。

```

春
夏
秋
冬

```

### 1.3 四則計算結果の表示

文字列ばかりではなく、計算結果を表示するプログラムを作成してみましょう。

**例題 6** 以下に示す ex06.c をコンパイルし、実行してみなさい。

リスト 3 ex06.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     printf("7 + 3 = %d\n", 7 + 3);
8     printf("7 - 3 = %d\n", 7 - 3);
9     printf("7 * 3 = %d\n", 7 * 3);
10    printf("7 / 3 = %d\n", 7 / 3);
11    printf("7 mod 3 = %d\n", 7 % 3);
12    printf("7 + 3 = %f\n", 7. + 3);
13    printf("7 - 3 = %f\n", 7. - 3);
14    printf("7 * 3 = %e\n", 7. * 3);
15    printf("7 / 3 = %e\n", 7. / 3);
16
17    return (0);
18 }

```

整数を表示する場合には表示させる場所に “%d” という記号を置きます。実数を表示する場合は “%f” で**固定小数点数** (fixed point number), “%e” で**浮動小数点数** (floating point number) を指定したことになります。(他にありますが、それらは変数の型を学習してからにしましょう) これらを変換子といいます。

“int1 % int2” は、整数 int1 を 整数 int2 で割った余りを求める演算です。剰余と呼ばれます。7/3 の計算結果に注意してください。これは整数同士の割算ですから、小学校で習ったように、その商は 2 で、余りは 1 となります。実数であることを明確にするには 7.0 (略して 7. も可) と記述しなければなりません。

**問題 3** 「加減よりも乗除が先」という四則計算の優先順位を確かめるプログラムを考えなさい。剰余との優先順位はどうなっているかも確かめなさい。

### 3の解

解答例 q03.c<sup>①</sup> に付け加えてみなさい。

リスト 4 q03.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     printf("7 - 3 * 4 = %d\n", 7 - 3 * 4);
8     printf("(7 - 3) * 4 = %d\n", (7 - 3) * 4);
9     printf("7 + 3 % 4 = %d\n", 7 + 3 % 4);
10    printf("(7 + 3) % 4 = %d\n", (7 + 3) % 4);
11
12    return (0);
13 }

```

### まとめ

- プログラミング言語 C では、ソースを編集、以下のように C コンパイラ (この講義では gcc) でコンパイルして実行ファイルを作る。

```
gcc foo.c -o bar -lm
```

- C では関数が呼び出されて実行される。main 関数は最初に行われる最も重要な関数であり、基本的には以下のように記述する。

```

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

```

```

int main(int argc, char **argv)
{
    ...
}

```

- (命令) 文はセミコロン “;” で終端する (例外もある)。
- 書式指定付き出力関数 printf は、標準出力 (一般には画面) に文字や数値を表示するための関数である。

```

printf("文字列");
printf("文字列 + 書式指定", 変数);

```

- 文字列データの日本語の編集など、かな漢字変換を ON にしている状態では、うっかりして命令文中に**全角の空白文字**を入力しないように気をつける。

### 1CD Linux の利用

自宅のパソコンは MS-Windows である場合が多いと思います。MS-Windows 用にも**フリーで利用できる C/C++**が

ありますので、それをインストールして宿題を考えることも一案です。あるいは Linux は C 言語の学習に適した環境ですから、思い切って Linux をインストールして MS-Windows とのマルチブートも時代の先端に行く選択かもしれません。しかし、Linux のインストールに失敗すると MS-Windows もブートできなくなるのでは、と心配しているなら、1CD Linux がお勧めです。ハードディスクにインストールすることなく、ただ単に CD-ROM をいれてブートさせれば、デバイスを調べ、X の起動までこなしてくれるという、涙物の 1 枚なのです。数ある 1CD Linux の中でも、KNOPPIX (くのーぴっくす) は、産総研の有志によってすすめられた日本語化の完成度が高く、雑誌の付録にも収録されることがあり入手が容易です。まったく便利な世の中になったものです。

どうしても MS-Windows から離れたくない方には、仮想化により MS-Windows の上で Linux を起動することも可能です。最も著名な仮想化ソフトの一つである VmPlayer には無償版がありますから、それを試してみても良いでしょう。

図 5 に、Knoppix/Math 2010 (数学のアプリケーションに特化した 1DVD-ROM) を VmPlayer で起動し、自動で立ち上がった LXDE (GNOME デスクトップ環境よりも軽い) のメニューから端末と gedit を起動して、C 言語のソースを編集・コンパイル・実行した様子を示します。

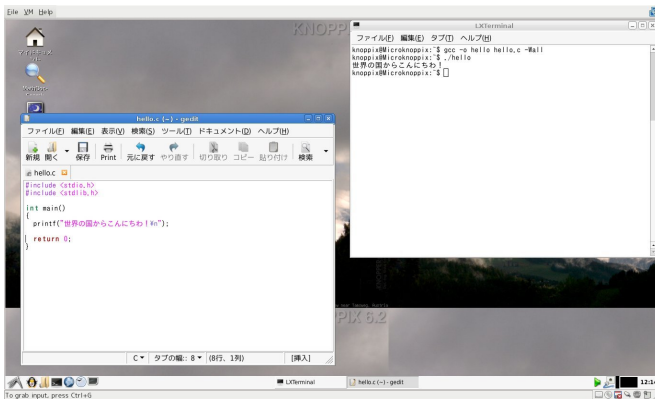


図 5 Knoppix/Math 2010 で gedit と端末を開いて作業している様子の画面例

パソコンを壊す心配が全くありませんから、是非、気楽に試してみてください。

#### 参考ウェブサイト：KNOPPIX

- 開発者 Knopper さんのページ
- 数学関連ツールを盛りこんだ KNOPPIX/Math

## 2 変数と型

C 言語では変数は宣言してから使います。宣言の一般的な書式は次のようなものです。

|| 「記憶クラス宣言子」 型指定子 識別子リスト

記憶クラスについては後述することとして、型指定子と識別子について説明します。識別子とは簡単にいえば名前のことです。単純なものはラベル名と呼ばれます。型にはあらかじめ定義された基本型と void 型、およびプログラマが新たに定義できる派生型があります。ここでは基本型と void 型を扱います。

☞ 実はこの講義の最後の方で、変数の集合を指定する型 (構造体、共用体、列挙型) の説明をしますが、その場合の識別子は **タグ** (tag) と呼ばれ、集合の形式の名前となりますから、その後ろに単純なラベル名 (実体) を指定することになります。

### 2.1 識別子

識別子すなわち名前は次の規則にしたがったものを使うことができます。

- 英数文字 (A~Z, a~z, 0~9), 下線 ( \_ )。
- 先頭に数字を用いることはできない。
- 大文字と小文字を区別する。
- 文字列の長さに関する規定はなく、処理系依存となる。一般的には、最初の 31 文字までが有効。

#### 2.1.1 予約語

C 言語では以下の語が予約語としてシステムで使われます。したがって、同じ名前の変数を用いることはできません。

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	singed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while	inline <sup>*C99</sup>	restrict <sup>*C99</sup>	_Bool <sup>*C99</sup>
		_Complex <sup>*C99</sup>	_Imaginary <sup>*C99</sup>	

☞ C 言語の規定は ANSI (American National Standards Institute) の規約 X3.159 を基にして ISO で 1990 年に採択されたものが標準とされています。その後 ISO で 1995 年と 1999 年に改訂され、特に 1999 年には新しい拡張部分が加わりました。この拡張部分については未実装なコンパイラが多いので、C99 をこのように <sup>\*C99</sup> 肩に付けて説明することになります。

☞ 予約語の他にも、常識的な名前などは定義済みの可能性があります。例えば math.h を読み込んだ場合には、初等数学関数 (sin, cos など)

は定義済みとなりますから、プログラマーが同じ名前の関数を定義できなくなります。こういった事態を避けるには、一般名称の前にプロジェクト名を付けるなどの工夫が必要となります。UNIX で有名な例は X Window System のライブラリ関数で、先頭に必ず X を付け、長い綴りの機能名（例えば XCreateSimpleWindow()）を付けて区別が付く様にしています。先頭に my（あるいは My）などの辞を付けて名前が重複しないように工夫します。

**例題 7** 次の変数名のうち、無効なものはどれですか（解は脚注<sup>1)</sup>）。

```
d, do, cpu_athlon, 386_cpu, flag-png, US$,
_arg1, INTEGER, Int,
```

2.2 基本型

2.2.1 整数型

整数型には、char, short int, int, long int, long long int の 5 種類があります。負数のない符号なしは unsinged を前置して表現しますが、正負のある符号付きは singed を省略するのが一般的です。ところで、扱える値の範囲は厳密な規定がありません。コンパイラの実装依存となりますから、正確にはヘッダファイル (/usr/include/)limit.h を参照してください。表 2 には、32bit CPU における標準の値を示します。

表 2 標準整数型と記憶サイズおよび値の範囲 (32bit CPU)

型	size	値の範囲
_Bool <sup>*C99</sup>	1	0,1
char	1	-128 ~ 127 または 0 ~ 255
unsigned char	1	0 ~ 255
signed char	1	-128~127
short	2	-32768 ~ 32767
unsigned short	2	0 ~ 65535
int	4	-2147483648 ~ 2147483647
unsigned int	4	0 ~ 4294967295
long	4	-2147483648 ~ 2147483647
unsigned long	4	0 ~ 4294967295
long long <sup>*C99</sup>	8	-9223372036854775808 ~ 9223372036854775807
unsigned long lomg <sup>*C99</sup>	8	0 ~ 18446744073709551615

<sup>1)</sup>do, 386\_cpu, flag-png, US\$

OS が 64 ビット化している現在、C の整数も 64 ビット幅を default とすることが自然です。この結果、単なる int は、OS に依存して 16, 32, 64 のいずれかのビット幅の整数を示すことになってしまいました。そこでビット幅を指定できる整数型が C<sup>99</sup> では導入されました（stdint.h 参照）。例えば int32\_t を用いて、CPU や OS に依存せず、常に 32 ビット幅の整数型を指定することができます（OS が 64bit であっても）。

**char 型** たった 255 の整数しか扱えない char 型は無意味と思われるかもしれませんが、これは整数というよりもその名前の通り文字コードを扱うための型として重要です。具体的には、char 型の変数の配列は文字列を扱う配列として用いられます。文字コード表は次のように ascii に関するオンラインマニュアルでも参照できます。

```
man ascii
```

2.2.2 浮動小数点型

実数型は、float, double, long double が定義されています。その記憶サイズと内部表現に規定はありません。よってコンパイラの実装依存となっていますが、ほとんどの場合、IEEE754-1985 の規格に従っているようです。GCC では (/usr/include/)ieee754.h を参照するとよいでしょう。表 3 に、32bit CPU の場合の記憶サイズと扱える値の範囲及び精度の桁数を示します。

表 3 実数型の記憶サイズと範囲 (32bit CPU)

型	size	値の範囲	精度
float	4	1.2E-38~3.4E+38	6
double	8	2.3E-308~1.7E+308	15
long double <sup>†</sup>	12	3.6E-4951 ~1.1e+4932	19

<sup>†</sup>GCC では、Intel のアーキテクチャーの CPU に対して、次のオプションで long double のサイズを変化させることが可能です。  
-m128bit-long-double (16bytes)  
-m96bit-long-double (12bytes, default)  
ただし、今のところ精度は変化しません。すなわち、16bytes だからといって、残念ながら精度が 30 桁にはならないということです。

**float 型について** 単精度実数 float は、過去のプログラムとの互換性を計るといった目的以外に意味のない型となっていました。なぜなら、CPU 内部で演算を実行する場合には既に double 以上の精度を使うからです。したがって、float を用いて計算が高速になることは望めません。実数計算では double を使いましょう。



**複素浮動小数点型**<sup>\*C99</sup> C 言を科学技術計算用の言語として用いる場合に大きな欠点がありました。それは複素数型がなかったのです。C++では、複素数クラスを定義して FORTRAN に遜色なく扱うことができますが、普通の C の範囲では大変でした。そこで ANSI C99 では、複素浮動小数点型を導入し、また複素数型の数学関数も採り入れました。ヘッダーファイル `complex.h` を読み込んで、変数宣言は

```
double _Complex x;
long double _Complex y=3+4i;
```

などとします。

### 2.2.3 void 型

`void` は値がない型を示します。次のように戻り値のない関数や引数がない関数などを示す場合に用いられます。

```
void func(...), double func(void)
```

また、後述しますが `void` へのポインタという使い方も重要です。

**例題 8** キーボードからの入力 円の半径を（キーボードから）入力して円の面積を（ディスプレイに）出力するプログラムを作成しなさい。

[解] リスト 5 `ex08.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     double r;
8     printf("radius = ");
9     scanf("%lf", &r);
10    printf("area = %f\n", M_PI * r * r);
11
12    return (0);
13 }
14
15
```

`ex08` という名前で実行ファイルを作成しましょう。

```
$ gcc ex08.c -o ex08 -lm
```

出来上がった実行ファイル `ex08` の起動のようすは次のようになります。

```
$ ./ex08
radius = 3
area = 28.274334
```

この例題の 9 行目で使われている数値の書式付き入力関数 `scanf("%lf",&r)` の `&r` は、変数 `r` へのポインタと呼ばれるものです。C 言語ではこのポインタを理解すること

が上級者への近道といわれます。ポインタはとても重要なのですがきちんと理解することが難しいので、今はこう書くものとして覚えてください。

**9 行目にある書式指定子 `%lf` と 10 行目の書式指定子 `%f` が違っていることに注意しましょう。** 本来 `f` は `float` の略ですから、単精度実数なのですが、`printf` ではなぜか `%f` は倍精度実数 `double` の指定となっています。scanf では `%f` は単精度実数の指定なので、`double` に対しては絶対に `%lf` (エルエフ) を使わないといけません。

11 行目の `M_PI` はもちろん  $\pi$  の値で、ヘッダーファイル (`/usr/include/`)`math.h` で定義されています。

## 2.3 書式付き出力 : `printf()`

表 4 書式付き出力の変換指定子

指定子	型	出力
d, i	int	10 進表記
o	unsigned int	8 進表記
u	unsigned int	10 進表記
x, X	unsigned int	16 進表記
e, E	double	10 進の指数表記
f, F	double	10 進の固定小数点のような表記
g, G	double	f 変換か e 変換のどちらかに変換する <sup>1</sup>
a, A <sup>*C99</sup>	double	16 進の指数表記 <sup>2</sup>
c	char, int	文字 (int は unsigned char に変換される)
s	文字列 <sup>3</sup>	文字列の先頭から終端を表す NUL (ナル) 文字 '\0' まで。
p	ポインタ	アドレスの 16 進表記
n	int *	自身の出力はない。それまでに出力された文字列数が引数に保存される。
%		文字 % 自身

<sup>1</sup> 詳細は `man 3 printf`

<sup>2</sup> C99 拡張

<sup>3</sup> すなわち `char` 配列へのポインタ


`printf()` は標準出力（一般には画面）へ書式を指定して出力する場合に多用する関数です。ex08.c の 11 行目は以下のようになっています。

```
printf("area = %f\n", M_PI*r*r);
```



基本的には二重引用符で囲まれた部分がそのまま表示されますが、変換指定子は後ろの引数の値に置き換えられて出力されるのです。

printf() 関数群のオンラインマニュアルを

man 3 printf 

で参照すると、詳しい説明を見ることができますが、変換指定 (conversion specification) の形式は以下のようになっています。

%[フラグ][フィールド幅][. 精度] 指定子

例: %12.8f %20.15g %-8.6e %d


%03d %80s %% %n %p

先頭の % 記号と指定子が必須で、それ以外はオプションです。フィールド幅は文字列の幅を明示的に指示する場合に用います。もし、実際に変換された文字列がフィールド幅より大きい場合には自動的に幅が伸びます。記号 . 以下の精度は、小数点以下の桁数を明示的に指示する場合に用います。フィールド幅とは異なり、桁数に丸められます。規定値は 6 です (単精度 float の精度)。指定子を表 4 に示します。

フラグ文字は、表示を整えるためのオプションを指定するもので、表 5 に示すように 5 つあります。それらの組み合わせもできます。

表 5 書式付き出力のフラグ文字


フラグ文字	出力
0	変換値の左側を 0 で埋める。
-	変換値を左揃えにする。0 フラグよりも優先される。
+	符号付き変換の数値の前に + か - の符号を表示する。
' ' (スペース)	符号付き変換の正值の前に空白を置く。
#	変換規則の代替。o 変換では 8 進数の先頭に 0 を付ける。X,x 変換ならば 16 進の先頭に 0x あるいは 0X を付ける。a,A,e,E,f,g,G 変換では、必ず小数点部を表示する。


**例題 9** 整数と実数を入力して表示する以下のプログラムを実行して、変換による表示の違いを確認しなさい。  
ex09u.c 


リスト 6 ex09.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
```

```
4
5 int main(void)
6 {
7     int i;
8     double x;
9
10    printf("integer i = ");
11    scanf("%d", &i);
12    printf("double x = ");
13    scanf("%lf", &x);
14    printf("\n");
15
16    printf("%d\t i = %d\n", i);
17    printf("%8d\t i = %8d\n", i);
18    printf("%010d\t i = %010d\n", i);
19
20    printf("%f\t x = %f\n", x);
21    printf("%.3f\t x = %.3f\n", x);
22    printf("%e\t x = %e\n", x);
23    printf("%12.8g\t x = %12.8g\n", x);
24
25    return (0);
26 }
```


**問題 4** 2 つの整数値を入力して、その加減乗除を表示するプログラムを考えなさい。q04.c 

**問題 5** 長方形の縦と横の長さを入力して、面積を出力するプログラムを作成しなさい。q05.c 

 変数と変換指定子の型が一致しない場合、コンパイル時に警告だけが出て実行プログラムは作成されてしまうことがあります。しかしこれは正しいプログラムが作成できたことを意味しません。一般に、実行時の出力は予期しないものになります。場合によっては、セグメンテーションフォルトを起こしてプログラムが停止します。すなわち、書式付き入出力関数; printf(), scanf() においては、変数と変換指定子の型は合致させなければなりません。

## 2.4 main 関数への引数

プログラムにデータ (あるいはパラメータ) を渡すために、前の例題では書式付き入力関数 scanf を用いました。それ以外に、main 関数に引数を渡す方法があります。プログラムを実行するには、そのプログラム名を入力しますが、それに続く文字列は \*\*argv (名前は任意ですが慣用的に argv を用います) に引数として格納され main 関数に渡されます。すなわち、

foo arg1 arg2 arg3 .... 

のように foo を実行すると、空白で区切られた文字列を一塊にして、argv[0] には 'foo', argv[1] には 'arg1', argv[2] には 'arg2', ... (以下同様)、それぞれ文字列として格納されるのです。main 関数の宣言中の int argc, char \*\*argv (argc も慣用的に用いています) はこのような目的で使われます。

この結果、引数の個数を示す `argc` は”引数+1”の値となります。この文字列を、整数や実数に変換して用いばいいのです。

**例題 10** `main` 関数に値を渡す方法で、円の半径から面積を計算して表示するプログラムを作成しなさい。

#### リスト 7 ex10.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int
6 main (int argc, char **argv)
7 {
8     double r;
9
10    r = atof (argv[1]);
11    printf ("area = %f\n", M_PI * r * r);
12
13    return (0);
14 }
```

**【解】** `ex10.c` が解答例です。9 行目の `atof()` は `ascii to float` すなわち文字から実数（ただし `double` です）への変換関数です。この他にも `int` への変換 `atoi()`、`long int` への変換 `atol()` などがあります。実行例を以下に示します。

```
$ ./ex10 3
area = 28.274334
```

ところで、このプログラムを引数なしで起動すると重大なエラーが発生した旨が表示されます（なぜか考えてみてください）。

```
$ ./ex10
Segmentation fault (core dumped)
```

このような誤動作を防ぐには、予め予想を立ててプログラムを組む必要があります。この例では、引数があることを確認して（できればそれが数値を表す文字列であることを確認して）`atof` を実行するようにすべきなのです。信頼性が要求される場合には、全く何も知らないユーザーがプログラムを使うことを想定して、種々のエラー処理を施さなければならないでしょう。しかし、ここではそこまで厳格には考えません。処理全体の見通しの良さとエラーの起こる確率を秤にかけながらエラー処理を考えます。

**問題 6** 各例題において、数字以外のメッセージを日本語にしてみなさい。

🔗 日本語を使う場合には、システムに合った日本語コードを使ってください。UNIX システムでは、

```
|| echo $LANG
```

で言語の情報を得ることができます。

**問題 7** 円の面積を表示する例題において、半径の自乗の計算に冪乗関数  $x^y$ : `pow(x,y)` を用いてみなさい。

**問題 8** `M_PI` は `math.h` で定義されています。この他にどんな定数が定義されているか調べてみなさい。

**問題 9** 3 角形の 3 辺の長さを入力して、その面積を表示するプログラムを考えなさい。3 辺の長さが  $a, b, c$  の三角形の面積  $A$  は以下の [Heron](#) の公式で与えられます。

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad \left( s = \frac{a+b+c}{2} \right)$$

`scanf` を用いる方法と `main` への引数 `**argv` を用いる方法、両方について考えなさい。

## まとめ

- 変数は以下のように宣言してから使う。

〔記憶クラス〕 型指定子 識別子

また型の種類は、基本的には整数と実数である。

- 変数が扱える数の範囲を指定するために接頭辞をつける。
- 整数では、符号なし `unsigned` と符号付き `signed` に気をつける。
- 実数は、単精度 `float` を使う意味があまりないので、通常 `double` を用いるようにする。

### 3 条件分岐

(コンピュータを用いて) ある特定の目的を達成するための処理手順を**アルゴリズム**と呼びます。そしてプログラミング言語によりアルゴリズムを具体化したものがプログラムです。アルゴリズムのなかで最も重要な要素の一つは**条件に応じて処理を変える**ということです。条件を適切に設定し対応する処理を考えることこそがアルゴリズムの本質といっても過言ではないでしょう。従って、プログラミング言語と呼ばれるからには、条件を記述するための**論理演算と条件分岐命令**が必ず備わっています。

#### 3.1 真偽値

C 言語では式は一般に値を持ちます。それを論理値として扱う場合、**0 は偽 (false) というのが大原則**です。真値については、`_Bool` 型では 1 と厳密に決まっていますが、一般的には真偽のどちらかが常に定まるように **0 以外の値を真 (true)** として扱います。真値が 1 であることを原則にしていませんから注意してください。

#### 3.2 関係演算子 (relational operator)

関係演算子は (式の) 値の大小の比較 (comprison) の結果を 1 または 0 で返す `int` 型の式を構成します。'=>' や '<=' という記号はありませんから注意しましょう。

表 6 関係演算子

演算子	意味	例	結果: 真なら 1, 偽なら 0
<	より小さい	<code>x &lt; y</code>	<code>x</code> が <code>y</code> より小さければ 1, でないなら 0
<=	以下	<code>x &lt;= y</code>	<code>x</code> が <code>y</code> 以下ならば 1, でないなら 0
>	より大きい	<code>x &gt; y</code>	<code>x</code> が <code>y</code> より大きければ 1, でないなら 0
>=	以上	<code>x &gt;= y</code>	<code>x</code> が <code>y</code> 以上ならば 1, でないなら 0
==	等しい	<code>x == y</code>	<code>x</code> と <code>y</code> が等しければ 1, でないなら 0
!=	等しくない	<code>x != y</code>	<code>x</code> と <code>y</code> が等しくなければ 1, でないなら 0

**✗ 関係演算子 == と 代入演算子 = の混用** 初心者が陥りやすい間違いは、== とすべきところを = と記述してしまうことです。問題なのは、間違えて書いてしまっても文法上の間違いとならない場合が多く、コンパイルが無事終了してしまうにも関わらず、実行時におかしな結果が生じてくるということです。**数学の記号と微妙に違うところ**がありますので注意してください。

### 3.3 論理演算子 (logical operator)

表 7 に論理演算子を示します。C 言語の論理演算子の演算対象となる (論理) 式が真であるとは、偽でないすなわち値が 0 以外であることです。真値を 1 に限定していませんから注意してください。


表 7 論理演算子

演算子	意味	例	結果: 真なら 1, 偽なら 0
&&	論理積 ( $x \cap y$ )	<code>x &amp;&amp; y</code>	<code>x</code> と <code>y</code> がともに真ならば 1, でないなら 0
	論理和 ( $x \cup y$ )	<code>x    y</code>	<code>x</code> と <code>y</code> の少なくとも一方が真ならば 1, でないなら 0
!	否定 ( $\sim x$ )	<code>!x</code>	<code>x</code> が真でないならば 1, でないなら 0

表 8 に否定、論理積、論理和、包含関係 ( $p$  ならば  $q$ ) の論理値を示します。真を  $T$  で、偽を  $F$  で表現しています。

表 8 論理演算表

$\sim p$	$p$	$q$	$p \cap q$	$p \cup q$	$p \supset q$
$F$	$T$	$T$	$T$	$T$	$T$
$F$	$T$	$F$	$F$	$T$	$F$
$T$	$F$	$T$	$F$	$T$	$T$
$T$	$F$	$F$	$F$	$F$	$T$

 **排他的論理和 (exclusive OR)** 「A または B」の意味は 2 通り考えられます。「老人または子供が先だ」ならば、これは論理和と考えられます。しかしながら、「ミカンまたはリンゴを使ったパイ」ならば両方を使うことは念頭にないでしょう。このように日常的には A と B 両方が同時には成立しないことを表現する場合が結構あります。これは排他的論理和と定義されます。

### 3.4 条件分岐命令

論理演算結果の真偽に応じて処理を替える条件分岐命令には、`if` 文と `switch` 文があります。

#### 3.4.1 if...else

構文は以下のようになっています。

|| `if (式) 文1 [else 文2]`

条件の真偽を評価して、真ならば文 1 を実行します (偽ならば何もしません)。else はオプションで、偽の場合に文 2 を実行するように指定できます。なお、複数の式を **中括弧**

{ } で括って**複合文（ブロック）**とし複雑な処理を実行させることができます。文が単独の場合には中括弧は要りませんが、処理の範囲がどこまでなのかははっきりさせるためどんな場合にも中括弧を書いた方が無難かもしれません。

**例題 11** if ... else を用いたプログラムを作成しましょう。キー入力した実数値  $x$  が 0 以外なら True, 0 ならば False と表示するものです。

リスト 8 ex11.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     double x;
8
9     printf("x = ");
10    scanf("%lf",&x);
11    printf("x = %g --> ", x);
12    if (x) {
13        printf("True\n");
14    } else {
15        printf("False\n");
16    }
17    return (0);
18 }
```

実行例は以下ようになります。0 として扱われる境界の値を探してみると面白いです。

```
$ gcc -o ex11 ex11.c -lm
$ ./ex11
数値 = 1.4
x = 1.4 --> True
$ ./ex11
x = 0.0
x = 0 --> False
$ ./ex11
x = 1e-500
x = 0 --> False
$ ./ex11
x = 1e-200
x = 1e-200 --> True
$ ./ex11
x = a
x = 0 --> False
```

このプログラムで最も重要な 12~16 行は次のように書いても同じ結果を得ることができます。

```
if (x) printf("True\n");
if (!x) printf("False\n");
```

すなわち**アルゴリズムは正しい**ことになります。しかしプログラム上は大きな違いがあります。このように書くと、2つの if 文が必ず順に 2 回評価されます。例題では、if 文の評価は 1 回だけです。したがって**効率の良いプログラム**という観点からは、例題の方が優れていることになります。さらに、この例では条件式が単純に  $x$  すなわち  $x \neq 0$  だけだったので、その否定も簡単に記述できました。しかし、

もしもっと複雑な条件だったら、その否定を間違えてしまうかもしれません。すなわち**間違いを起こし難い**という観点からも、例題の方が良いと考えられます。

**問題 10** 上記例題に手を加えて、「入力引数の実数変換値  $x$  を表示し、 $x$  が 3 以上なら True, それ以外は False を表示する」プログラムに変えなさい。入力値として 2.999...9 を与えた場合、9 をいくつにしたら 3 と判断されか確かめなさい。

**問題 11** 平面の座標点  $P$  の  $x, y$  成分を受け取り、 $P$  が点 (2,3) を中心にして半径 2 の円内にあるか否かを判定するプログラムを作成しなさい。q11.c

**例題 12** 条件が複雑な場合を練習しましょう。2つの実数  $x, y$  を受け取り、 $x-y$  平面の第何象限に属しているかを表示するプログラムを作成しなさい。ただし、 $x, y$  軸上にある場合には軸名を、また原点ならば原点にいと表示させなさい。

リスト 9 ex12.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     double x, y;
8
9     printf("x, y = ");
10    scanf("%lf, %lf", &x, &y);
11    printf("(%.2f,%.2f) --> ", x, y);
12
13    if (x > 0) {
14        if (y > 0) printf("1st quadrant\n");
15        else if (y == 0) printf("x-axis\n");
16        else printf("4th-quadrant\n");
17    }
18    else if (x == 0) {
19        if (y == 0) printf("origin\n");
20        else printf("y-axis\n");
21    }
22    else { /* x < 0 */
23        if (y > 0) printf("2nd quadrant\n");
24        else if (y == 0) printf("x-axis\n");
25        else printf("3rd-quadrant\n");
26    }
27
28    return 0;
29 }
```

**解** ex12.c が解答例です。実行例を以下に示します。

```
$ ./ex12
x, y = 1, 0
(1.00,0.00) --> x-axis
$ ./ex12
x, y = 0, 0
(0.00,0.00) --> origin
$ ./ex12
x, y = -2, -3
(-2.00,-3.00) --> 3rd quadrant
```



### 3.5 switch...case

整数型の制御式の値を複数の整数値と比較するには、if ... else でつないでいくと見通しが悪いので、switch ... case 文を用います。構文は以下のようになります。

```
switch (制御式) {
    case 値1 : 文1
                [break;]
    case 値2 : 文2
    ...
    default : 文d
}
```

セミコロン ';' ではなくコロン ':'

制御式の値と同じ値を持つ case ラベルに分岐してそれ以降の処理を実行します。default: はオプションで、あれば case に合致しない場合の処理を記述します。途中 break; があればそこで外に抜けます。

**例題 13** 文字を読み込み 'a, A' ならば alive, 'd, D' ならば dead, それ以外は other と表示するプログラムを作成しなさい。switch ... case 文を使いなさい。

リスト 10 ex13.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     char inputc;
8     printf("文字 = ");
9     scanf("%c", &inputc);
10
11     switch (inputc) {
12     case 'A': "break;" がないので次の行へ進む
13     case 'a':
14         printf("alive\n");
15         break;
16     case 'D': "break;" がないので次の行へ進む
17     case 'd':
18         printf("dead\n");
19         break;
20     default:
21         printf("other\n");
22         break;
23     }
24     return (0);
25 }
```

**[解]** ex13.c が解答例です。argv[1][0] は文字列 argv[1] の最初の 1 文字を指します（そうなる理由は文字列のところでしっかり学習します）。8 行目でその値を inputc に代入しています。もちろん scanf("%c", &c) を用いて読み込むようにしても結構です。

scanf で文字を 1 つ読み込むために "%c" と指定した場合には、空白類（スペース、タブ、改行）の無視が無効となってしまう、期待した結果を得られない場合があります。空白類の無視を有効とするために、%c の前に空白をあけて " %c" と指定してください。

実行例を以下に示します。

```
$ ./ex13
文字 = D
dead
$ ./ex13
文字 = a
alive
$ ./ex13
文字 = p
other
```

**問題 12** if ... else 文を用いて、例題と同じ出力を得るプログラムを考えなさい。どちらが見通しがよいですか？  
q12.c

**問題 13** オプション指定のための 1 文字と 2 つの実数 x, y を読み込み、文字が k ならば相加平均  $(x + y)/2$  を、文字が j ならば相乗平均  $\sqrt{xy}$ 、文字が b ならば両方を表示するプログラムを作成しなさい。q13.c

**文字定数** 単引用符で囲まれた 1 つ以上の文字から構成された定数を、文字定数 (character constant) と呼びます。文字定数は int 型 です。例えば定数 'A' の値は 10 進数で 65 (16 進数で 0x41) です。2 文字以上からなる文字定数の値は規定がありません。すなわち互換性が保証されませんから使用を避けるべきでしょう。

ところで GCC ではどうなっているか気になりますから、文字定数 'abc' の値を表示するプログラムを実行してみましょう。sizeof 演算子 は変数などのメモリ格納サイズを返します。

リスト 11 notes3-5.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(void)
6 {
7     int s = 'abcd';
8     printf("size = %d\n", sizeof(s));
9     printf("%#x\t%d\t%c\n", s, s, s);
10    return (0);
11 }
```

まず当然ながら、コンパイル時に警告が出ます。size は 4 ですから (long) int であることが判ります。結果は、文字を低位側に向けて 1byte ずつ並べて構成した整数となりました。

```
$ gcc -o charconst -Wall charconst.c
charconst.c: In function 'main':
charconst.c:7: warning: multi-character character constant
$ ./charconst
size = 4
0x616263 6382179
```

### 3.6 条件演算子

if ... else は文なので値を持つことはありません。条件分岐を行い、その分岐先の式の評価の結果を値として持つ演



算子が条件演算子です。次のように記述します。

|| 式 a ? 式 b : 式 c

a が真ならば b が評価され (c は評価されません), a が偽ならば b が評価されます (c は評価されません)。評価された方の式の値が全体の値となります。if ... else 文とは異なり, b, c に複合文を置くことはできません。なお, 条件演算子を入れ子 (多重に用いる意) にすることは可能です。条件演算子には, 項が 3 つあることから **3 項演算子**と呼ばれます。

**例題 14** 条件演算子を試すために, 次のソースを編集して実行してみなさい。

リスト 12 ex14.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     int n;
8     double x;
9
10    printf("x = ");
11    scanf("%lf", &x);
12    printf("%f --> ", x);
13    n = x ? printf("True\n") : printf("False\n");
14    printf("printed characters = %d\n", n);
15
16    return (0);
17 }
```

条件演算子を用いているので, 分岐先の式を評価した値が全体の値となります。printf() 関数が返す値は出力した文字数ですから, 真の場合には 'True' と '\n' の 5, 偽の場合には 6 となります。実行例を示します。

```
$ ./ex14
x = 12
12.000000 --> True
printed characters = 5
$ ./ex14
x = 0
0.000000 --> False
printed characters = 6
```

**問題 14** 2 つの実数を読み込み大きい方の数値を表示するプログラムを考えなさい。if ... else 文 と 条件演算子 それぞれを用いて書いてみなさい。

**問題 15** 3 つの実数を読み込み, それを 3 辺とする三角形が構成できる場合には, その面積を表示し, 三角形を構成できない場合にはその旨を表示するプログラムを考えなさい。なお, 任意の 2 数の和が残る数よりも大きい場合に, 三角形が構成可能です。

**問題 16**  $x-y$  平面上の座標  $P(x,y)$  が, 三角形 OAB の内部, 辺上, 外部のいずれにあるかを判定するプログラム

を考えなさい。ただし, O, A, B の座標はそれぞれ (0,0), (1,0), (0,2) です。

**問題 17** 方程式  $ax^2 + bx + c = 0$  の解を求めるプログラムを考えなさい。二次方程式にならない場合も考慮しなさい。q17.c

## まとめ

- 値の大小と等値を判定する関係演算子は記号  $> >= < <= == !=$  で表わされる。
- 論理否定, 論理積, 論理和は記号  $! \ \&\& \ ||$  で表される。
- 論理式の真偽値は, 偽値が 0 であることは一貫している。したがって真値は 0 以外となる。真値を 1 と限定する場合もある。
- 条件分岐には次の 3 種類がある。
  - if ... [else] 文
  - switch ... case 文
  - 条件演算子 a ? b : c
- switch ... case 文の制御式および case ラベルに含まれる値は整数型である。
- 条件演算子は条件分岐して評価された式の値を全体の値として持つ。

## 4 繰り返し (loop)

コンピュータは、人間には耐えられないあるいは人間には不可能な回数の単純な処理の繰り返しを行ってくれます。これはコンピュータを利用する大きな利点といえます。アルゴリズムの観点から言えば、条件分岐命令と無条件分岐命令があれば繰り返し命令を構成することが可能です。しかしながら、繰り返しの実行はコンピュータの大きな特徴ですから、それを活かすために複数の命令が用意されています。繰り返しはその性格上、次の4つの要素によって構成されます。

- 繰り返したい処理そのもの (ループ本体)
- 処理の継続を判定する**制御式**
- 制御式で判定にかかる変数 (制御変数) の調節 (一般には増減)
- 制御変数の初期化

### 4.1 while 文

#### 4.1.1 while

構文は以下の通りです。記述がありませんが**制御変数の初期化**を忘れないでください。制御式が真ならば文① (ループ本体+制御変数の増減) を実行し、再び制御式の評価に戻り真ならば... を繰り返します。制御式が偽になった時点でループを抜け出します。

|| while (制御式) 文①

複雑な処理を行うには文①として、**複合文** (複数の文を中括弧で括ったもの) を用いることができます。

**例題 15** 整数  $i_{\max}$  を読み込み、10 から  $i_{\max}$  までの整数  $i$  およびその自乗  $i^2$  を表示するプログラムを作成しなさい。

#### リスト 13 ex15.c

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5
6 int main(int argc, char **argv)
7 {
8     制御変数 i の初期化
9     int i = 10, imax;
10
11     printf("imax = "); 制御式
12     scanf("%d", &imax);
13     while (i <= imax) {
14         printf("i = %3d\t i**2 = %5d\n", i, i * i);
15         i++;
16     }
17     制御変数 i の増加
18     return (0);
19 }
```

[解] while 文を使った解答例を ex15.c に示します。行番号でいうと、10→11→12→10→11... を10行の制御式が真であるかぎり繰り返します。

12行目  $i++$  の  $++$  は整数  $i$  を1だけ増加させる演算子です。これを**増分演算子** (increment operator) といいます。逆に1減ずる演算子  $--$  は**減分演算子** (decrement operator) といいます。ここで、増分演算子を含めて算術演算子をまとめておきましょう。

表 9 算術演算子

演算子	意味	例	結果
*	乗算	$x * y$	$xy$
/	除算	$x / y$	$x/y$
+	加算	$x + y$	$x + y$
-	減算	$x - y$	$x - y$
%	剰余	$x \% y$	$x \bmod y$
			$x, y$ は整数型
++	増分	$++x, x++$	$x$ の値を1増す。 式の値に注意*。
--	減分	$--x, x--$	$x$ の値を1減らす。 式の値に注意*。

\* 増分演算子、減分演算子について これらの算術演算子は、オペランド (この場合にはすなわち整数型の変数) の右に置かれる場合 (後置記法 postfix notation) と左側に置かれる場合 (前置記法 prefix notation) があります。どちらもオペランドの値を1増減しますが、式の値自身は異なります。前置記法 ( $++i, --i$ ) は増減後の値 ( $i+1, i-1$ ) となりますが、後置記法 ( $i++, i--$ ) は増減前の値 ( $i$ ) です。

🔗 変数  $i$  の値を1増すには、単純代入演算子 (simple assignment operator) を用いて

```
i = i+1;
```

としても構いません。しかし、数学の等式として見てしまうと明らかに間違いのこの式をわざわざ使う必要はないでしょう。あるいは、複合代入演算子 (compound assignment operator) を用いて

```
i += 1;
```

と記述することもできます。

実行例は以下のようになります。

```

$ ./ex15
imax = 100

i = 10  i**2 = 100
i = 11  i**2 = 121
i = 12  i**2 = 144
... (中略)
i = 99  i**2 = 9801
i = 100 i**2 = 10000

$ ./ex15 9
$ ← 何も出力されない
```

10 未満の整数を渡した場合には、条件式が最初から偽ですから、ループ本体は 1 回も実行されません。

**問題18** 上記例題で、初期化  $i=10$  を忘れた場合、制御変数の増加  $i++$  を忘れた場合にどうなるか試してみなさい。ソースの該当箇所を削除したりしないでください。/\* と \*/ で囲んでコメントアウトするのが普通です。

#### 4.1.2 do...while

do ... while は while とほとんど同じですが、その特徴は、以下の構文をながめて判る様に

do 文 while (制御式); 制御式の評価が文 (ループ本体+制御変数の調節) の後にあるため、**ループ本体が最低 1 回実行される**ことにあります。

**例題16** 前の例題と同じ内容を do ... while を用いて作成しなさい。

リスト14 ex16.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     制御変数 i の初期化
8     int i = 10, imax;
9
10    printf("imax = ");
11    scanf("%d", &imax);
12    do {
13        printf("i = %3d\t i**2 = %5d\n", i, i * i);
14        i++; 制御変数 i の増加
15    } while (i <= imax); セミコロンを忘れずに
16    return (0);
17 }
```

実行例を示します。ループ本体が最低 1 回実行されますから、10 未満を引数に与えても 初期値  $i=10$  の場合の処理だけは表示されてしまいます。この例は do ... while に適してないといえましょう。

```
$ ./ex16
imax = 9
i = 10 i**2 = 100
```

## 4.2 for 文

for 文はループ本体を除く 3 つの制御に関する部分をまとめて記述できるので、すっきりしています。構文は

|| for (式1; 式2; 式3) 文

となっており、式1に変数の初期化、式2はループ継続の制御式、式3に制御変数の調節を記述します。while 文と同じく文には複合文を置く事もできます。評価の順番は、式1 (1回だけ)、式2、文、式3、式2、文、式3、... 以下繰り返します。

**例題17** while, do ... while の例題を for 文を用いて書き換えなさい。

リスト15 ex17.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     int i, imax;
8
9     printf("imax = ");
10    scanf("%d", &imax);
11    for (i = 10; i <= imax; i++) {
12        printf("i = %3d\t i**2 = %5d\n", i, i * i);
13    }
14
15    return (0);
16 }
```

**[解]** ex17.c が解答例です。11 行目に繰り返し制御に関する式がすべて記述されていて、繰り返したい処理のみをループ本体に書く事ができるので大変使いやすい繰り返し文です。実行例は省略します。

#### 4.2.1 多重ループ

for 文は制御構造が判りやすいので、入れ子にして使う場面が多いです。

**例題18** 0 ~ 127 の整数値を、8 個ずつ 16 行に渡って、10 進数と 16 進数で 100(x64) のように表示するプログラムを考えなさい。

リスト16 ex18.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     int i, j, n;
8
9     for (i = 0; i < 16; i++) {
10         for (j = 0; j < 8; j++) {
11             n = 8 * i + j;
12             printf("%3d(x%02x) ", n, n);
13         }
14         printf("\n");
15     }
16
17     return (0);
18 }
```

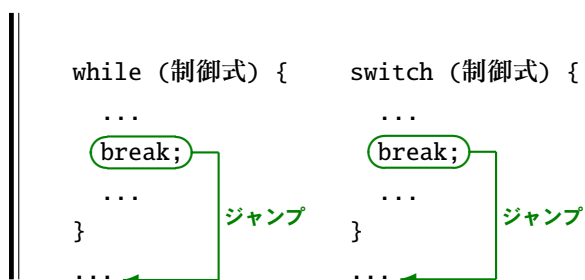
**問題19** 例題の表に、列の区切り線と上下の水平の罫線を加えて見栄えを良くするプログラムを考えなさい。 [q19.c](#)

### 4.3 無条件分岐

C 言語には制御式の判定とは関係なく、無条件に特定の場所にジャンプする命令があります。break, continue, goto, return がそのような無条件分岐命令です。break は既に switch ... case 文で登場していて switch ... case 文の外に抜ける場合に用いました。また、return (値) も最初から登場していますが、関数から抜けて、関数を呼び出した場所に戻る命令ですから無条件分岐命令ということができません。これらの制御命令を無限ループや制御式の条件分岐と組み合わせることで非常に柔軟な制御構造を構成することができます。

#### 4.3.1 break

break はループや switch ... case の外に抜ける命令です。



もちろん他のループ、do ... while 文、for 文においても同様に用いることができます。

**例題19** 文字を読み込んで表示する無限ループを形成し、q または Q を受けたら終了するプログラムを考えなさい。

リスト17 ex19.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(void)
6 {
7     char c;
8
9     while (1) {
10         printf("Input a character ");
11         scanf("%c", &c);
12         if (c == 'q' || c == 'Q') {
13             break;
14         }
15         printf("%c\n", c);
16     }
17     printf("Now quit\n");
18     return (0);
19 }
```

**解** ex19.c をコンパイルしてください。実行例は以下のようになります。

```
$ ./ex19
Input a character a
a
Input a character 1
1
Input a character Q
Now quit
$ ← 終了してシェルプロンプトに戻る
```

**別解** ex19b.c のように、break を使わずに while の制御式に継続条件 (= 終了しない条件) を記述してループを組むことも可能です。すると、外に抜けるのはループを最後まで実行して制御式の評価に戻った時点となります (なぜなら 10, 11, 12, 9 行と実行されるので)。すなわち、q, Q を入力した場合でも、表示を行う文が実行されてしまい、期待通りの結果でなくなります。

リスト18 ex19b.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(void)
6 {
7     char c;
8
9     while (c != 'q' && c != 'Q') {
10         printf("Input a character ");
11         scanf("%c", &c);
12         printf("%c\n", c);
13     }
14     printf("Now quit\n");
15     return (0);
16 }
```

なお、継続条件すなわち終了条件  $A \parallel B$  の論理否定は  $!(A \parallel B) = !A \&\& !B$  ですから、9 行目のようなやや複雑な制御式となります。

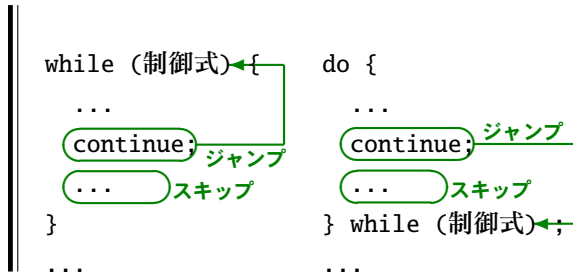
```
$ ./ex19b
Input a character Q
Q ← Q が表示されている
Now quit
$
```

**問題20** 例題を、while ループをそのままにして、switch ... case 文を用いて書き換えなさい。while ループを抜けるためのフラグを用いなければなりません。制御構造の見通しはよくなったでしょうか。 [q20.c](#)

#### 4.3.2 continue

continue はループの中だけで使うことができ、ループ本体の残り部分をスキップします。すなわち while 文で

は制御式, for では制御変数の調節を行う式にジャンプします。



**例題 20** break の例題に, さらに次のような制御を付け加えなさい。 “入力文字が c か C ならば, continue を用いてループ本体をスキップする”。

リスト 19 ex20.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(void)
6 {
7     char c;
8
9     while (1) {
10         printf("Input a character ");
11         scanf("%c", &c);
12         if (c == 'c' || c == 'C') {
13             continue;
14         } else if (c == 'q' || c == 'Q') {
15             break;
16         }
17         printf("%c\n", c);
18     }
19     printf("Now quit\n");
20     return (0);
21 }
```

[解] 解答例 ex20.c をながめてください。実行例を示すまでもないでしょう。

### 4.3.3 goto

goto は任意のラベルにジャンプするという大変強力な命令で, 多用すると制御構造が入り込んでしまう危険性がありますから, 本当に限られた場合に使うように心がけるべきといわれています。例えば, 制御が複雑になり過ぎて, 一気にそのルーチンを抜けさせたい場合などは goto を使うと最も見通しがよくなります。なお, ラベルは同じ関数内になければいけません。

**例題 21** continue の例題の if ... else 文を switch ... case 文に書き改めて見通しをよくします。すると break 命令単独では while ループ本体から抜けられなくなります。そこで一気に脱出させられるよう goto を使ってみなさい。

リスト 20 ex21.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(void)
6 {
7     char c;
8
9     while (1) {
10         printf("Input a character ");
11         scanf("%c", &c);
12         switch (c) {
13             case 'c':
14             case 'C':
15                 break;
16
17             case 'q':
18             case 'Q':
19                 printf("Now quit\n");
20                 goto QUIT;
21
22             default:
23                 printf("%c\n", c);
24                 break;
25         }
26     }
27 QUIT:
28     return (0);
29 }
```

任意位置の  
ラベルへジャンプ

[解] ex21.c を吟味してください。

### 4.3.4 return

return は, 呼出された関数を終了して, 呼び出しをかけた位置に制御を戻す命令で, 無条件分岐と分類していいでしょう。関数の中に複数の return を置く事ができます。関数のところで詳しく使い方を学習します。構文を示します。

return 式 ;

式の値が関数の型に変換されて返却値 (return value) となります。括弧を使って return (式) と書く必要はありません。現在 C 言語に関する教科書のほとんどで括弧をつけない様式となっています。しかし, 括弧をつけても間違いではありませんし, 複雑な式の場合には可読性が高まるという意見もあります。この講義では混乱を避けるため参考書の様式に合わせています。

## 4.4 実数によるループ制御

制御変数には整数以外の型の変数を用いることができます。例えば実数を用いた方が数式などと対応が良い場合もあります。しかし, ループの回数を正確に守りたいなどといった場合には, 整数をもちいるように工夫しないといけません。



**例題 22**  $1e-5$  ( $1 \times 10^{-5}$ ) を刻み幅にして 0 から 1 までという制御のつもりで書いた以下のプログラムの実行結果を吟味しなさい。

リスト 21 ex22.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     int i, lmax;
8     double dd = 1.0e-5, s, id;
9
10    printf("dd = %.15f\n", dd);
11    printf("lmax = %d\n", (int)(lmax = 1 / dd));
12    for (s = 0, i = 0; s <= 1.0; s += dd, i++) {
13        id = i * dd;
14        if (i > lmax - 3) {
15            printf("%6d, %.15f, %.15f\n", i, s, id);
16        }
17    }
18    return (0);
19 }
```

実行結果を示します。

```

$ ./ex22

dd = 0.000010000000000000
lmax = 99999
99997, 0.9999699999998084, 0.999970000000000000
99998, 0.9999799999998084, 0.999980000000000000
99999, 0.9999899999998084, 0.999990000000000000
100000, 0.9999999999998084, 1.000000000000000000
```

まず、 $1/(1e-5)$  が 100000 になっていません。同じことかもしれませんが  $1e-5$  を 100000 回足した結果が 1 になりません。したがって、1 に等しいときに終了などという制御式をかいたら旨くないのは明らかです。一方  $1e-5$  に整数  $n$  をかけたものは  $ne-5$  ( $n \times 10^{-5}$ ) となっていますから、まだましです。

**問題 21** 1 から  $n$  までの整数の 2 乗の和を計算し、公式

$$\sum_{k=1}^n k^2 = \frac{1}{6}n(n+1)(2n+1)$$

と比較するプログラムを考えなさい。 [q21.c](#)

**問題 22** 次のような、無限級数展開の公式を用いて関数  $e^x$  の近似値を求めます。

$$e^x = \sum_{k=0}^{\infty} c_k(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

キー入力値  $x$  に対して、 $c_k(x) < 1e-15$  ( $1.0 \times 10^{-15}$ ) となる項までの和を求めて近似値とし、数学ライブラリ  $\exp(x)$  の示す値 (真値と考える) と比較するプログラムを考えなさい。 [q22.c](#)

## まとめ

- 繰り返し (loop) には次の 3 種類がある。
  - while (制御式) 文
  - do 文 while (制御式);
  - for (式 1; 式 2; 式 3) 文
- break, continue, goto などの無条件分岐命令を用いて複雑なまた柔軟な制御構造を構成できる。
- 繰り返しの制御変数はなるべく整数を用いること。実数を用いると累積誤差により判定間違いを生ずる可能性がある。

## 5 配列

同じ型の多数のオブジェクトの集まりに番号をつけたものが配列 (array) です。配列は**名前**、要素の**型**と**個数**を指定して定義します (個数は省略できる場合があります)。すなわち、

```
|| 型 名前 [要素数];
```

と宣言します。例えば

```
|| double x[128];
```

は、double 型のオブジェクトを要素とした要素数 128 の配列です。配列の各要素は、番号を鉤括弧で括った番号を配列名の後ろにつけて、

```
|| x[0], x[1], x[2], x[3] ...
```

と表します。番号は**添字** (subscript)、添字を括る記号の [ ] は**添字演算子** (subscript operator) と呼ばれます。例に示したように、**添字は 0 から始まります**。すなわち、先頭から  $i$  個目の要素は添字演算  $[i-1]$  で指定されます。したがって、**要素数  $N$  で定義した配列の添字の最大値は  $N-1$  です**。プログラマは添字の範囲を越えないように注意しなければなりません。というのも、 $N$  以上の添字を指定したソースをコンパイルしてもエラーはでないのが普通だからです。もちろん、実行時には予想もできない間違いが起こり得ます。なお、**宣言時の要素数は定数**でなければなりません。すなわち、

```
int N = 5;
double x[N];
```

のような宣言はできません。ただし、ANSI C99 ではブロック有効範囲においては正の整数式を要素数に用いて宣言しても良いことになりましたので、上記のような宣言が可能です。GCC は ANSI C99 の実装がすすんでおり、このような宣言ができてしましますが、互換性を重視するならば使わないべきです。

⚠ 配列の宣言時の要素数が定数でなければならないというのは不便です。これでは予め大きさが定められない場合には、予想して大きめに確保しなければならなくなり無駄ですし、予想をはずれて範囲を越えることも起こり得ます。後に、動的な配列の定義 (メモリの確保) の方法を学習します。

### 5.1 配列要素への値の代入

配列要素は単独の変数と同じ扱いができますので、値を代入するには、**代入演算子** = の左側に置き、その右側に代入元の値 (式) を置きます。

```
x[0] = 0.12;
x[2] = 2*x[0] + x[1];
x[k] = x[i]*x[j];
```

ただし、配列そのものを代入することはできません。

```
double ax[50], ay[50];
ax = ay;  ← 配列変数への代入は不可
```

はコンパイルエラーとなります。したがって、配列のコピーを得るには、各要素を全て代入するという地道な方法しかありません。

### 5.2 配列の宣言時の初期化

単独の変数と同じく配列を宣言時に初期化することができます。各要素に対応した定数値を中括弧の中でカンマで区切って並べます (リスト)。

```
double x[3] = {0.1, 0.3, 0.5};
int kx[4] = {1, 2, 3, 4};
char str[6] = {'H', 'e', 'l', 'l', 'o'};
```

要素数よりも右側の項目数が少ない場合には、残りの部分が 0 で初期化されます。要素数よりも右側の項目数が多い場合には、GCC ではコンパイル時に警告が発せられ (エラーとまでは至らないようです) 要素数以上の項目が無視されます。char 型の配列は文字列として扱う関係上、要素数は右側の項目数よりも 1 多く宣言した方が安全です。理由は、文字列の節で明らかになります。

また、定数値リストを用いて代入を行うことはできません。すなわち以下の代入 (のつもり) はコンパイルエラーとなります。

```
double x[2];
x[3] = {0.1, 0.2};  ← 配列要素と型が違う
x = {0.1, 0.2};  ← 配列変数への代入は不可
```

#### 5.2.1 要素数を省略した初期化

配列を宣言時に要素データの並びで初期化する場合、要素数の記述を省略することができます。このとき要素数は初期化に用いられた定数値のリストの長さとなります。

```
double x[] = {0.1, 0.3, 0.5};
int kx[] = {1, 2, 3, 4, 5, 6};
char str[] = {'W', 'o', 'r', 'l', 'd'};
```

上記の例では、配列 kx の要素数は 6 です。したがって添字が 6 以上の配列要素の読み書きを行ってはなりません。

**例題 23** ランダムに発生させた 0 から 9 までの整数を要素とする、要素数 16 の配列を添字とともに表示するプログラ

ムを考えなさい。0 から RAND\_MAX の間の疑似乱数整数を生成する関数 rand() と、その初期化の関数 srand(seed) を用いなさい。

リスト 22 ex23.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>  ← 乱数の初期化のために必要
5
6 int main(int argc, char **argv)
7 {
8     int i, xi[16];
9
10    srand(time(NULL));  ← 乱数の初期化
11    for (i = 0; i < 16; i++) {  ← '< 16' が間違いない
12        xi[i] = rand() % 10;
13        printf("xi[%2d] = %d\n", i, xi[i]);
14    }
15    return (0);
16 }
```

[解] ex23.c が解答例です。12 行目で rand() により整数の乱数を得ます。さらに 剰余演算子 % によって 0 から 9 までの整数に変換しています。srand(seed) を実行しないと、同じ乱数列が生成されてしまいますので、seed に正整数値を与えて srand() を呼びます (10 行目)。その時、1970 年 1 月 1 日 00:00:00 UTC からの経秒数を得る関数 time(NULL) を使って整数値を得るのが標準的です。コマンドを起動する度、異なる値が得られ、その値で初期化できることとなりますから好都合なのです。なお、NULL はヌルポインタと呼ばれる特別なポインタです。詳細はポインタの節で学習します。

実行結果を示します。長いので途中を省略します。

```

$ ./ex23
xi[ 0] = 9
xi[ 1] = 9
xi[ 2] = 0
...
xi[15] = 2
$ ./ex23
xi[ 0] = 6  ← 前とは異なる乱数となっている
xi[ 1] = 3
xi[ 2] = 8
...
xi[15] = 5
```

**問題 23** 例題を元にして、要素の取る値の範囲や配列の要素数を変えたプログラムを作成しなさい。

**問題 24** 得られた配列の全要素の平均値 (実数) を計算して表示するプログラムを考えなさい。

### 5.3 多次元配列

行列 (matrix) などは、2 次元の配列で表現するのが自然です。M × N 行列 A は A[M][N] と宣言すれば、各要素を添字 i, j を用いて、A[i][j] で読み書きできるようになります。

要素が配列であるような配列を多次元配列と呼びます。つまり A[M][N] と宣言した多次元配列は N 個の要素を持った配列が M 個並んだ配列となります。したがって A[i] は i 番 (i+1 個目) の配列 (要素数は N) を意味します。また A[i][j] = (A[i])[j] は i 番 (i+1 個目) の配列の中の j 番 (j+1 個目) の要素ということになります。

**例題 24** 0 から  $N^2 - 1$  までの整数を要素とする N 次の正方行列を生成し、表示するプログラムを作成しなさい。ただし、各要素の位置がランダムとなるようにしなさい。

リスト 23 ex24.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5
6 #define N 4
7
8 int
9 main (int argc, char **argv)
10 {
11     int i, j, n, A[N][N];
12
13     srand (time (NULL));
14     for (i = 0; i < N; i++) {
15         for (j = 0; j < N; j++) {
16             A[i][j] = -1;
17         }
18     }
19
20     for (n = 0; n < N * N; n++) {
21         while (1) {
22             i = rand () % N;
23             j = rand () % N;
24             if (A[i][j] == -1) {
25                 A[i][j] = n;
26                 break;
27             }
28         }
29     }
30     for (i = 0; i < N; i++) {
31         for (j = 0; j < N; j++)
32             printf ("%2d ", A[i][j]);
33         printf ("\n");
34     }
35
36     return (0);
37 }
```

[解] ex24.c の 6 行目において、配列の大きさ N を

マクロ定義: #define N 4

を使って定義しました。この指令により N が 4 に置き換わってからソース全体がコンパイルにかかります。したがって、コンパイル時点では (正の) 整数定数となっています。# で始まる命令はもうすでに #include を使っています。こ

れらは**プリプロセッサ指令**と呼ばれますが、それらについては後でまとめて学習します。

アルゴリズムを概説します。各要素を -1 で初期化しておき、代入されているかどうかの判断に用います。20 行から 26 行が、整数値をランダムな位置に代入するルーチンです。代入先の要素の候補を乱数で選んで、その値が -1 なら代入を実行します。-1 でなければ既にその要素には代入が行われていますから、次の候補をまた乱数で選びます。これを 0 から  $N^2 - 1$  について繰り返します。

30 行からが、二重ループを用いて 1 行分表示したら改行することで、得られた行列をそれらしく表示するルーチンです。ex24u.c<sup>c</sup>

$N = 4$  として作成した場合の実行例を以下に示します。0 から 15 までの整数が全て揃っていますか。

```
$ ./ex24
9 0 14 3
6 13 15 10
12 4 8 11
2 5 1 7

$ ./ex24
9 5 10 0
8 2 12 7
1 6 14 15
11 4 3 13
```

**問題 25** 例題の位置を決定するアルゴリズムは、特に最後の方で明らかに無駄な繰り返しがあり感心しません。アルゴリズムの改良を考えなさい。q25.c<sup>c</sup>

## 5.4 多次元配列と 1 次元配列

1 次元配列は連続した記憶領域を占めます。 $N \times M$  の 2 次元配列も普通に宣言すれば要素数  $NM$  の 1 次元配列と記憶領域における並びが同一となります。このことを利用すると、2 次元配列を 1 次元配列としてあるいは逆に 1 次元配列を 2 次元配列として用いることができます。

**例題 25** 要素数 9 の 1 次元配列と  $3 \times 3$  の 2 次元配列を同じ定数リストで初期化して、格納領域での並びが等しいことを確認しなさい。

**[解]** ex25.c にソース例を示します。15, 20 行  $\&A[i][j]$  の記号  $\&$  は  $A[i][j]$  のアドレス (address: メモリ上の位置) を返す**アドレス演算子**と呼ばれるものです。scanf() ですでに登場していますが、詳しいことはポインタの節で学習します。変換指定子 %p は、ポインタ (アドレス演算子を作用させた変数 &x はポインタになっています。) のアドレスを 16 進数で表示させる指定です。(long) int は大きさが 4 バイトですから、その先頭番地から 4 バイト分を占めることになります。ex25.c<sup>c</sup>

### リスト 24 ex25.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     int i, j;
8     int B[3][3] = {
9         {8, 7, 6},
10        {5, 4, 3},
11        {2, 1, 0}
12    };
13    int A[9] = { 8, 7, 6, 5, 4, 3, 2, 1, 0 };
14
15    printf("One dimensional array\n");
16    for (i = 0; i < 3; i++)
17        for (j = 0; j < 3; j++)
18            printf("%d => %p\n", A[3 * i + j], &A[3 * i + j]);
19
20    printf("Two dimensional array\n");
21    for (i = 0; i < 3; i++)
22        for (j = 0; j < 3; j++)
23            printf("%d => %p\n", B[i][j], &B[i][j]);
24
25    return (0);
26
27 }
```

実行例は以下のようになります。

```
$ ./ex25
One dimensional array
8 => 0x7ffc8878b5c0
7 => 0x7ffc8878b5c4
6 => 0x7ffc8878b5c8
5 => 0x7ffc8878b5cc
4 => 0x7ffc8878b5d0
3 => 0x7ffc8878b5d4
2 => 0x7ffc8878b5d8
1 => 0x7ffc8878b5dc
0 => 0x7ffc8878b5e0
Two dimensional array
8 => 0x7ffc8878b5f0
7 => 0x7ffc8878b5f4
6 => 0x7ffc8878b5f8
5 => 0x7ffc8878b5fc
4 => 0x7ffc8878b600
3 => 0x7ffc8878b604
2 => 0x7ffc8878b608
1 => 0x7ffc8878b60c
0 => 0x7ffc8878b610
```

連続した領域内に全く同じ順番で並んでいることが判ります。記憶領域上の配置が次のようになっていることを理解しましょう。

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	7	6	5	4	3	2	1	0

B[0][0]	B[0][1]	B[0][2]	B[1][0]	B[1][1]	B[1][2]	B[2][0]	B[2][1]	B[2][2]
8	7	6	5	4	3	2	1	0

## 5.5 配列の応用

### 5.5.1 素数を求めるプログラム

参考書 p.110 に記載されていますが、素数を求めるプログラムは配列を理解する標準的な演習問題です。



**例題 26** 素数を USHRT\_MAX までの範囲で求めるプログラムを考えなさい。求められた素数を配列に順次登録してゆくこととし、整数  $n$  が素数であることの判定条件として、

$n$  の平方根以下の全ての素数で一度も割り切れることがない

を用いて、効率化しなさい。

#### リスト 25 ex26.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <limits.h>
5
6 int main(int argc, char **argv)
7 {
8     int i, n, flag;
9     int prime[USHRT_MAX] = {2,3}, ptr = 2;
10
11     for (n = 5; n < USHRT_MAX; n += 2) { ← 偶数のみ
12         flag = 1; ← default は素数であるとしておく
13         for (i = 1; prime[i]*prime[i] <= n; i++) {
14             if (n % prime[i] == 0) { ← もし割り切れたら
15                 flag = 0; ← 素数ではないので
16                 break; ← for を抜ける
17             }
18         }
19         if (flag) { ← もし素数ならば
20             prime[ptr] = n; ← 素数配列に登録
21             ptr++; ← 添え字を増加：次に移る
22         }
23     }
24
25     for (i = 0; i < ptr; i++) {
26         printf("%5d, ", prime[i]);
27     }
28     printf("\n%d\n", ptr); ← 見つかった素数の総数
29     return (0);
30 }
```

[解] ex26.c に例を示します。unsigned short int 型の最大値 USHRT\_MAX は limits.h に定義があります (32bit CPU 用の GCC では 65,535)。参考書と違って、素数であることを真 (=1) とした flag を用います。12 行目で素数であると仮定して判定のループに入り、割り切れたら素数ではないので flag を偽 (=0) にします (15 行目)。ループ終了後、19 行目 flag が真であれば素数なので、素数の配列に登録します。ex26u.c ←

**複合代入演算子** 11 行目の  $n += 2$  は複合代入演算 (compound assignment operator) と呼ばれるもので、 $n$  に加法演算  $+$  を行った結果を代入します。この場合には  $n$  に 2 が加わることになります。一般には、増分・減分演算子を除く算術演算子 (表 9 参照)  $op$  に対して  $x \ op = y$  と記述して用いることができます。以下に例を示します。

記述	結果
$x \ op = y$	$x = x \ op \ y$
$x += 1$	$x = x + 1$
$x /= a$	$x = x / a$
$i \% = 8$	$i = i \% 8$

#### 5.5.2 頻度分布への応用

配列の典型的な応用例は頻度分布の作成です。変数  $x$  が区間  $[k\Delta, (k+1)\Delta]$  の値をとる事象が起こる数を  $h[k]$  に積算して得られます。添字は正整数なので  $x$  の値 (一般には実数) からの変換に少し注意が必要です。

**例題 27** 区間  $[0,1)$  の一様疑似乱数列を与える関数 drand48() を使い、次の近似式より正規乱数を生成して、その分布 (中心が 0.5) を数値表示するプログラムを考えなさい。

$$R_N = \frac{r_1 + r_2 + \cdots + r_{12}}{12} \quad (r_i \text{ は } [0, 1) \text{ の一様乱数})$$

試行回数を大きくすると滑らかな分布が得られます。

#### リスト 26 ex27.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5
6 #define N 20
7
8 int main(int argc, char **argv)
9 {
10     int i, imax, j, xi[N] = { 0 };
11     double s;
12
13     printf("imax = ");
14     scanf("%d", &imax); ← 試行回数を取得
15     srand48(time(NULL)); ← drand48() の初期化
16     for (i = 0; i < imax; i++) {
17         s = 0;
18         for (j = 0; j < 12; j++) {
19             s += drand48();
20         }
21         s /= 12;
22         xi[(int) floor(s * N)] += 1;
23     }
24     for (i = 0; i < N; i++) {
25         printf("%.2f: %.2f] = %6d\n", (i + 0.0) / N,
26             (i + 1.0) / N, xi[i]);
27     }
28     return (0);
29 }
```

[解] ex27.c に例を示しますが、ちょっと説明が必要です。

**キャスト演算子** 22 行目の (int).. はキャスト演算子 (cast operator) と呼ばれ、右側に続く変数の型を明示的に変換する場合に用います。(int) は (実数を) を整数に変換します。実は C 言語では、暗黙の型変換という規則があっ




て、例えば、整数型の変数と実数型の変数の算術演算式があると、整数型の変数を実数値に変換してから計算を行います。もちろんこれは実数の情報を保持するための処置です。例題中では、26 行目の  $(i+0.0)/N$  は暗黙の型変換を期待したものです。これが  $i/N$  のままでは整数/整数として計算されますから、期待通りの結果は得られません。もちろん  $(double)i/N$  と明示的な型変換をする方が良いと言われていますが、ここでは敢えて暗黙の型変換を使ってみました。


暗黙の型変換に逆からうには、**実数から整数への型変換は明示的に行う** 必要があるわけです。ex27.c<sup>㉔</sup> をダウンロードしてコンパイルしてください。

説明が長くなりました。実行例は次のようになります。

```
$ ./ex27
imax = 1000000


[0.00:0.05] = 0
[0.05:0.10] = 0
[0.10:0.15] = 4
[0.15:0.20] = 83
[0.20:0.25] = 889
[0.25:0.30] = 6686
[0.30:0.35] = 28340
[0.35:0.40] = 80972
[0.40:0.45] = 159473
[0.45:0.50] = 223680
[0.50:0.55] = 223799
[0.55:0.60] = 159870
[0.60:0.65] = 80496
[0.65:0.70] = 28190
[0.70:0.75] = 6549
[0.75:0.80] = 901
[0.80:0.85] = 66
[0.85:0.90] = 2
[0.90:0.95] = 0
[0.95:1.00] = 0
```

 **drand48 の品質**: GCC では標準である drand48 は疑似乱数列発生関数としてあまり品質が高くないという批判があります。周期が  $2^{48}$  であり十分長いようにみえますが、大規模な数値計算あるいは暗号などで使用する場合には注意が必要です。また、drand48 以前からある標準関数 rand のオンラインマニュアルも読んでください。

man 3 rand 

**参考ウェブサイト：乱数**

- [Mersenne Twister Home Page](#)
- [良い乱数・悪い乱数](#)

 例題の配列  $xi[N]$  の宣言時に  $xi[N]=\{0\}$ ; と初期化しない場合には、配列の要素の初期値は不定ですから、結果が異常となってしまいます。全てを 0 で初期化する必要があります。

**問題26** 例題の数値表示に代えて、マーク \* の棒グラフを表示するプログラムを考えなさい。端末画面の 1 行の表示文字数を考慮して、マークの最大印字数を 50 として設計するとよいでしょう。q26.c<sup>㉔</sup>

表示例を以下に示します。

```
$ ./q26
imax = 300000
[0.00:0.05] =
[0.05:0.10] =
[0.10:0.15] =
[0.15:0.20] =
[0.20:0.25] =
[0.25:0.30] = *
[0.30:0.35] = *****
[0.35:0.40] = *****
[0.40:0.45] = *****
[0.45:0.50] = *****
[0.50:0.55] = *****
[0.55:0.60] = *****
[0.60:0.65] = *****
[0.65:0.70] = *****
[0.70:0.75] = *
[0.75:0.80] =
[0.80:0.85] =
[0.85:0.90] =
[0.90:0.95] =
[0.95:1.00] =
```

## まとめ

- 同じ型の多数のオブジェクトの集まりは配列と呼ばれ、その各要素は添字を用いて読み書きができる。  
宣言: `double x[50];`  
代入: `x[1] = 1.0;`
- 配列の代入は不可。コピーは各要素を全て代入しなければならない。
- 配列は、定数値リストにより初期化できる。  
`double x[2]={2e-3, 0.1};`  
`int dd[]={3, 4, 5, 9}`
- 多次元配列は配列を要素とする配列であり、添字を多重に用いて要素を読み書きできる。  
`double A[6][7];`  
`int C[2][3][3];`
- 配列の添字は非負の整数である。実数値から添字を決定する場合には、キャスト演算子を用いて明示的に変換する必要がある。
- 配列の添字の範囲はプログラマが管理して、範囲を越えた読み書きをしないように注意する。

## 6 関数

長い複雑な処理を単純な処理の組み合わせに分解することがプログラムの設計の大きな指針です。C 言語では、単純な（正確には機能が明確でちゃんと分離していることが大切）処理を関数という形の実行単位にまとめます。いままで作成してきた main 関数では、いくつか標準ライブラリ関数（library function）を利用してきましたが、それだけでは見通しのよい設計にならない場合が多く、独自の関数を作成する必要性が生じます。

### 6.1 関数宣言と関数定義

関数も宣言・定義されていなければ呼び出して使うことができません。変数は宣言がほとんどそのまま定義となりますが、関数では単なる宣言あるいは**プロトタイプ宣言**と定義を別に分けて記述することがあります。単なる宣言は、返却値の型と関数名だけを記すものです。返却値がなければ void 型を指定します。

|| 型 名前 ()

プロトタイプ宣言はその関数を呼び出して使う際に必要な仕様が記されたヘッダ部分を与えるもので、以下のように行います。

|| 型 名前 (仮引数リスト)

単なる宣言に比べると関数が受け取る仮引数の型を指定するところが違います。仮引数の名前を記してもよいですが、関数定義時の仮引数リストの名前がコンパイラに渡されるので、プロトタイプ宣言時の名前は単なるコメントに過ぎません。返却値あるいは仮引数がない場合には void 型を指定し明示します。以下に例を示します。

```
double funcz(double)
void funcv(char, double, double)
```

最初の funcz は、double 型の引数を 1 つ取り double 型の値を返す関数となります。また、funcv は、char 型を 1 つ、double 型を 2 つ引数に取り、返却値がない関数を宣言しています。

関数の定義は、完全に定義されたヘッダ部分と本体（文の集まりが中括弧囲まれブロック化されている）を記述します。すなわち

```
|| 型 名前 (完全な仮引数リスト) ← 関数ヘッダ
{                                     ← 中括弧でブロック化
    変数宣言
    ...
    文の並び
    ...
    return 式; ← 型が void なら不要
}               ← 中括弧でブロック化
```

となります。仮引数は型だけでなくその名前（関数内で有効）も指定して完全な形で与えなければなりません。

関数はファイルの任意の場所で定義できますが、関数の中で定義することはできません。

**inline 関数** <sup>\*C99</sup> 型の前に inline をつけると、コンパイラに関数の呼び出しのスピードを最適化するように指示したことになります。一般には実行速度が上がることを期待したもので、C90 ではマクロ定義などがそのような高速化の手法として用いられる場合があります。

### 6.2 関数の呼出し

関数の呼び出しは、printf, atof, atof などのライブラリ関数を今まで自然な形で使ってきましたので説明するまでもないでしょう。定数値や変数名あるいは算術式などを関数の引数（実引数）として渡し、返ってくる処理結果を受け取るというものです。もちろん型が一致していなければなりません。ところで、変数名を渡した場合、変数そのものではなく値だけが関数内部の変数に代入されて渡されます。これを**値渡し**（pass by value）といいます。したがって、**呼び出し側の変数の内容が書き換えられることはありません**。以下に、double funcz(double z) という定義の関数の呼び出しの模式を示します。

```
double funcz(double z)
{
    z を含む処理 ← 関数内部では変数名は z
}

int main(void) ← 呼び出し側
{
    ...
    funcz(1.2); ← 1.2 が z に代入される
    ...
    x = 2.0;
    funcz(x); ← x の値 2.0 が z に代入される
    funcz(x*x); ← 式 x*x の値 4.0 が z に代入される
    ...
}
```

**例題 28** 2 つの実数を引数にとり、大きいほうの値を返す関数を定義し、main 関数から呼び出してみなさい。

**[解]** ex28.c に例を示します。大小を判定する関数 bigone の 8,10 行目に return 文があり、場合に応じて返却値を変えています。実行例は以下のようになります。

```
$ ./ex28
x, y = -61, 2.4
2つの実数 -61 と 2.4 を比べて、大きいのは 2.4 です。
```

リスト 27 ex28.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 double bigone(double s, double t)
6 {
7     if (s >= t) {
8         return (s);
9     } else {
10        return (t);
11    }
12 }
13
14 int main(int argc, char **argv)
15 {
16     double x, y;
17
18     printf("x, y = ");
19     scanf("%lf, %lf", &x, &y);
20     printf("2つの実数 %g と %g を比べて, ", x, y);
21     printf("大きいのは %g です. \n", bigone(x, y));
22
23     return (0);
24 }

```

**問題27** 三角形の3辺の長さから3つの頂角を計算する関数を定義し、main関数から呼び出して結果を表示するプログラムを考えなさい。q27.c<sup>←</sup>

### 6.3 引数のない関数

引数がないあるいは返却値がない関数をいくつか作成してみましょう。

**例題29** ktermの画面をクリア（表示を一掃して左上にプロンプトを移すことです）する関数を作成し、main関数から呼び出してみなさい。ktermやxtermなどの端末エミュレータは、次の文字列を受けると画面をクリアします。“\E[H\E[2J”ここに\EはASCIIコードでESC（10進数で27、16進数で0x1b）です。

**[解]** ex29.cに例を示します。clrという関数名にしました。main関数から呼び出されると、画面をクリアします。

☞ 画面制御の情報は、コマンドinfocmpを端末エミュレータで実行すれば得られます。そこに現れるclear=XXXのXXX部分が画面消去の制御文字列です。他の制御記号の意味はman terminfoを読んでみてください。もちろん、このような低レベルの制御文字をいちいちプログラムする必要はなく、画面制御のための専用のライブラリlibcurses（Linuxではlibncurses）があり、それを利用すれば良いのです。

#### リスト28 ex29.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 void clr(void)
6 {
7     char clear[8] =
8     {27, '[', 'H', 27, '[', '2', 'J'};
9     printf("%s", clear);

```

```

10 }
11
12 int main(int argc, char **argv)
13 {
14     clr();
15     return (0);
16 }

```

**問題28** 整数 $n$ を引数にとり、記号“-”を出力して長さ $n$ の横線を描く関数を作成しなさい。q28.c<sup>←</sup>

### 6.4 アドレス渡し

大規模なデータを処理する場合には、大きな配列を関数に渡す必要が生じます。その場合に、複製するのは不経済ですし、だいいち配列への代入はできませんから配列の値渡しはできないのです。また、配列に限らず一般に関数に渡す元の変数自身の値を書き換える必要が生ずる場合もあります。単純な値渡しでは上記の要求を実現することはできません。

そこで、オブジェクト（変数や配列や関数の総称の意味）の**アドレス渡し**という手法があります。結局は**アドレスの値を渡す**のですが、この機構を用いればもっと柔軟に変数の情報を関数に伝えることができます。詳しくはポインタの節で学習することにして、ここでは概略を述べます。

関数scanf()は、値を格納する変数の前には記号&をつける仕様になっていました。この記号は**アドレス演算子**と呼ばれ、変数がメモリのどこの番地に収まっているかその先頭番地を得る演算子です。一般の変数 $v$ の値とは実はアドレス& $v$ からのデータ（ビット列）をその型の情報にあわせて解釈した結果なのです。ここで重要なのは番地とそこから始まるデータの型が判っていることです。このような理由で、**アドレスの値と型**を関数に渡せば、関数側でそのアドレスで表される変数の読み書きができることになります。

配列においては、型と配列が収まっているメモリの先頭番地が判っていれば、その要素へのアクセスが可能になります。もちろん関数においても、関数の開始アドレスが判れば、引数やreturn先などの情報が伝わるようになっていきます。表10にオブジェクトとそのアドレスを得る記法をまとめます。

表 10 オブジェクトとそのアドレスの取得

オブジェクト	宣言	アドレス
変数 $x$	$x$	& $x$
配列 $A$	$A[]$	$A$ (& $A$ も可)
関数 $f$	$f()$	$f$ (& $f$ も可)

### 6.4.1 配列を渡す

配列 `A[]` では式 `A` は配列の第 1 要素のアドレス値 (`&A[0]`) と同じ値を持っています, それを渡すことで, 関数側で配列の要素の読み書きが可能となります. 関数側の仮引数は配列で宣言すれば良く, 関数内部での配列要素の読み書きも添字演算子を用いて可能ですから, 煩わしくありません.

**例題 30** 整数の配列の最大値を求める関数を作成し, `main` 関数から呼び出すプログラムを考えなさい. 整数配列は疑似乱数で発生させることにします. [ex30u.c](#)

[解] `ex30.c` に例を示します. 整数配列の最大値を返却する関数 `maxof()` に配列名と配列の大きさを渡しています.

リスト 29 `ex30.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5
6 #define N 32
7
8 int maxof(int n[], int imax)
9 {
10     int i, max = n[0];
11
12     for (i = 0; i < imax; i++) {
13         if (n[i] >= max)
14             max = n[i];
15     }
16     return (max);
17 }
18
19 int main(int argc, char **argv)
20 {
21     int i, a[N];
22
23     srand(time(NULL));
24     for (i = 0; i < N; i++) {
25         a[i] = rand() % 10000;
26         printf("%4d ", a[i]);
27         if ((i % 10) == 9) printf("\n");
28     }
29     printf("\n 1 番大きい数値は %d です. \n", maxof(a, N));
30
31     return (0);
32 }
```

実行例を示します.

```
$ ./ex30
153 6142 2924 1420 7508 6840 8709 1537 6601 8504
4362 4478 3439 3421 4589 6881 8901 688 1037 2499
6290 131 2596 4089 5274 9955 2222 9620 3779 9655
3702 284
1 番大きい数値は 9955 です.
```

### 6.4.2 変数のアドレスを渡す

配列と異なり普通に宣言した変数 `z` のアドレスを渡すには, アドレス演算子を付けた `&z` を関数の実引数とします. また, 関数側ではアドレスを受けるように仮引数宣言をしなければなりません. それには, ポインタ変数として宣言します. すなわち, アスタリスクを前につけて

```
double *x
```

のように宣言します. また, これを「ダブルへのポインタ型変数 `x`」と読みます. `x` はアドレス値を扱う変数となります (型の情報も持っています).

**例題 31** 実数の変数のアドレスを受けて, それが指す変数の値を 1.0 に書き換える関数を作成し, `main` から呼び出すときに渡した変数の値が書き換えられることを確かめるプログラムを考えなさい.

[解] とても単純な例を `ex31.c` に示します.

リスト 30 `ex31.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 void one(double *x)
6 {
7     *x = 1.0;
8 }
9
10 int main(int argc, char **argv)
11 {
12     double z = 10.0;
13
14     printf("z = %f\n", z);
15     one(&z);
16     printf("z = %f\n", z);
17
18     return (0);
19 }
```

5 行目, 関数の引数はアドレスを受け取るので, ポインタ変数として宣言します. 7 行目, ポインタ変数が指す変数に値を代入することができます. `x` は `main` 関数中の変数 `z` のアドレスと型を持っています. すなわち, `x = &z` なのです. そこでそのアドレスに位置する変数 `z` は間接演算子をつけて `*x = *(&z) = z` で得ることができます. 15 行目, 関数 `one()` はアドレスを要求しますから, 変数 `z` のアドレス `&z` を実引数にして呼び出します.

**間接演算子 '\*'** アドレスが判っても, その内容が具体的に得られなければ意味がありません. アドレスと型 (この 2 つの内容を持つものをポインタといいます) からデータを解釈して読み書きできる状態にする演算があり, ポインタの前に記号 `*` を付けます. これは**間接演算子**と呼ばれます. すなわち, `ptr` がポインタ (オブジェクトのアドレスと型をもっている) であれば, `*ptr` は `ptr` が指すオブジェクトとなります. 普通に宣言した変数 `x` では `&x` が `x` をさす (定数) ポインタですから, `*&x = x` となります. なお, 普通に宣言した変数 `x` はポインタではありませんから `*x` は無意味です.

**問題 29** 2 つの実数値を交換する関数を作成して, その働きを確かめるプログラムを考えなさい. [q29.c](#)

**問題 30** 整数の配列の最大値と最小値を求める関数を作成し, `main` 関数から呼び出すプログラムを考えなさい. 整数配列は疑似乱数で発生させることにします. [q30.c](#)



**ヒント：**配列型の関数は定義することができませんので、結果を伝えるには、max, min をアドレス渡しすることとなるでしょう。(構造体を使えばもっと簡単に多数の値を返却できるようになります)

```
void maxminof(int a[], int imax, int *max,
int *min)
```

## 6.5 再帰関数

自分自身を呼び出す関数は**再帰的** (recursive) であると呼ばれます。階乗などは再帰的に計算できる典型的な例です。

**例題32** 階乗を求める関数を再帰関数として定義してみなさい。整数ではすぐに上限を越えるので、型は実数としなさい。

**[解]** ex32.c にソースの例を示します。

リスト31 ex32.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 double fact(double x)
6 {
7     if (x > 1) {
8         return (x * (fact(x - 1))); ← 自分自身の呼び出し
9     } else {
10        return (x);
11    }
12 }
13
14 int main(int argc, char **argv)
15 {
16     int i;
17
18     for (i = 1; i <= 25; i++) {
19         printf("fact(%2d) = %.0f\n", i, fact(i));
20     }
21
22     return (0);
23 }
```

実行例は以下の通りです。

```
$ ./ex32
fact( 1) = 1
fact( 2) = 2
fact( 3) = 6
fact( 4) = 24
fact( 5) = 120
fact( 6) = 720
fact( 7) = 5040
... (中略) ...
fact(19) = 121645100408832000
fact(20) = 2432902008176640000
fact(21) = 51090942171709440000
fact(22) = 112400072777607680000
fact(23) = 25852016738884978212864
fact(24) = 620448401733239409999872
fact(25) = 15511210043330986055303168
```

23 の階乗は明らかに間違っています。倍精度実数の有効桁は約 16 桁ですから、しかたありません。もちろん上位の 16 桁は正しいです。long double を用いると約 20 桁の有効数字となります。すると、どこまで完全に正しく計算できるでしょう。

**問題31**  $n$  を大きい自然数として Stirling の公式  $\log n! \simeq n \log n - n$  が成立することを確かめるためのプログラムを作成しなさい。本当に確かめられましたか？ [q31.c](#)

任意精度計算ツール [gp](#) を用いると次の結果を得ます。階乗の計算値が急速に大きくなるので、1000000! は計算できませんでした。

$n$	$\log(n!)$	$n \log n - n$
1000	5912	5907
10000	82109	82103
100000	1051299	1051292

**例題33** 関数  $f(x)$  の区間  $[a, b]$  における定積分は、

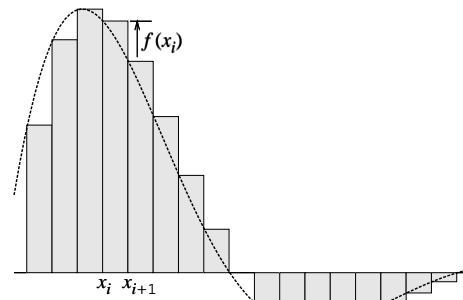


図 6 単純な求積法による数値積分の概念

図 6 ように一定幅  $x_{i+1} - x_i = h = (b - a)/n$  の細い長方形の面積の和と近似して、以下のように求めることができます。

$$\int_a^b f(x) dx = h \sum_{i=0}^{n-1} f(x_i)$$

$\cos(x)$  の区間  $[a, b]$  における定積分を上記の方法に従って数値計算する関数を考えなさい。区間の両端  $a, b$  の他に区間の分割数  $n$  を引数にするものとします。また、 $a = 0$ ,  $b = x$  として、解析解  $\sin(x)$  と比較するプログラムを考えなさい。

[解] 少し長いですが、ex33.c に例を示します。

### リスト32 ex33.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 void hline(int len)
6 {
7     for (; len > 0; len--) printf("-");
8     printf("\n");
9 }
10
11 double integ_cos(double a, double b, int n)
12 {
13     int i;
14     double sum = 0, dx = (b - a) / n;
15
16     for (i = 0; i < n; i++)
17         sum += cos(a + i * dx) * dx;
18     return (sum);
19 }
20
21 int main(int argc, char **argv)
22 {
23     int i, N;
24     double x, numeric, err;
25
26     printf("N = ");
27     scanf("%d", &N);
28     hline(45);
29     printf("x\t\tinteg[cos(x)]\terror\n");
30     hline(45);
31     for (i = 0; i <= 32; i++) {
32         x = 2 * M_PI / 32 * i;
33         numeric = integ_cos(0, x, N);
34         err = numeric / sin(x) - 1;
35         printf("%.6f\t%.6f\t%.4e\n", x, numeric, err);
36     }
37     hline(45);
38
39     return (0);
40 }

```

11～19 行、積分を行う関数 integ\_cos() は説明の必要がないでしょう。計算結果の体裁を整えるため、罫線を引く関数 line() を定義しました (5～9 行)。27 行、分割数 N をキー入力から読み取ります。34 行で相対誤差 = (計算値 - 真値) / 真値 = 計算値 / 真値 - 1 を計算しています。

実行例を示します。x = 0, π, 2π では相対誤差の分母 sin(x) = 0 ですから、相対誤差が桁違いに大きくなっています。ex33u.c

```

$ ./ex33
N = 10000
-----
x                integ[cos(x)]      error
-----
0.000000         +0.000000         +nan
0.196350         +0.195091         +9.6690e-07
0.392699         +0.382685         +3.9055e-06
... (中略) ...
2.945243         +0.195382         +1.4952e-03
3.141593         +0.000314         +2.5654e+12
3.337942         -0.194760         -1.6945e-03
3.534292         -0.382343         -8.8841e-04
... (中略) ...
5.890486         -0.382661         -5.8613e-05
6.086836         -0.195084         -3.0006e-05
6.283185         -0.000000         +7.1605e-01
-----

```

**問題32** 例題の方法を少し変えて、長方形の高さを  $x_i$  で

はなく  $x_i + h/2$  の位置で与えることにしてみなさい。より良い近似値が得られましたか。q32.c

**問題33** 関数  $f(x)$  の微分の定義

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

に従い、十分小さい  $h$  を与えて関数  $x^3$  の微分値を数値計算する関数を作成しなさい。また、解析解  $\frac{dx^3}{dx} = 3x^2$  と比較するプログラムを考えなさい。

## 6.6 標準数学関数

C 言語の標準数学ライブラリに加えて GNU の GCC のライブラリには多くの数学関数が含まれています。GNU 独自の関数あるいは C99 の標準指定から取り入れられた関数にはどのようなものがあるかは、調べないと判りません。使っている GCC のバージョンの数学関数については、/usr/include/math.h および /usr/include/bits/mathcalls.h を眺めればおおよそが判ります。それぞれの関数の仕様は“man 関数名”で調べてください。ただし、実装されていてもオンラインマニュアルの整備が追いついてない場合もあります。

C99 で追加された全ての関数を使う場合には、\_\_USE\_ISOC99 を、GNU 独自の拡張関数を使うなら \_\_USE\_GNU をそれぞれセットします。すなわち

```

#define __USE_ISOC99 1
#define __USE_GNU 1
#include <math.h>

```

とします。

表 11 int 型の数学関数

C 言語の関数名	数学上の関数
rand()	0 から RAND_MAX までの疑似一様乱数
srand(seed)	rand() を初期化する。seed には unsignedint を。型は void。
abs(n)	n
div(n,m)	n/m を計算して、商の整数部と余りを div_t という構造体の、quot, rem という int 型メンバーに格納する。
random()	rand() にほぼ同じ、BSD4.3 由来。
srandom(seed)	random() を初期化する関数、BSD4.3 由来。

表 12 double 型の数学関数

C 言語の関数名	数学上の関数
<code>sin(x), cos(x), tan(x)</code>	$\sin(x), \cos(x), \tan(x)$
<code>asin(x), acos(x)</code>	$\arccos(x), \arcsin(x)$
<code>atan(x), atan2(y, x)</code>	$\arctan(x), \arctan(y/x)$
<code>sinh(x), cosh(x), tanh(x)</code>	$\sinh(x), \cosh(x), \tanh(x)$
<code>pow(x, y), sqrt(x)</code>	$x^y, \sqrt{x}$
<code>exp(x)</code>	$e^x$
<code>frexp(x, *n)</code>	正規化分数 $y$ , 整数 $n$ $2^n y = x \quad (\frac{1}{2} \leq y < 1)$
<code>ldexp(x, n)</code>	$2^n x \quad (n: \text{整数})$
<code>log(x), log10(x)</code>	$\log_e(x), \log_{10}(x)$
<code>ceil(x), floor(x)</code>	$\lceil x \rceil, \lfloor x \rfloor$
<code>fabs(x)</code>	$ x $
<code>fmod(x, y)</code>	$x$ を $y$ で割った余り
<code>modf(x, *y)</code>	$x$ の小数部, $*y$ には $x$ の整数部実数

## まとめ

- 良いプログラムでは、処理を分割する適切な関数が設計されている。
- 関数は返却値の型、仮引数リストの宣言、定義本体で構成される。
- 関数は**値渡し**を行うので、呼び出し側の変数の値が書き換えられることはない。
- 配列やポインタ変数を引数として**アドレス渡し**を行うと、関数内部で呼び出し側のオブジェクトの値を書き換えることが可能になる。
- 自分自身を呼び出す関数を**再帰関数**という。

表 13 double 型の数学関数：GCC で普通に使えるもの

C 言語の関数名	数学上の関数
<code>rint(x)*C99</code>	最も近い整数値(実数)
<code>copysign(x, y)*C99</code>	絶対値が $x$ に等しく符号は $y$ に同じ
<code>drem(x, y)</code>	$x$ を $y$ で割った余り。 ( <code>remainder(x)</code> を使うべき)
<code>remainder(x, y)*C99</code>	$x$ を $y$ で割った余り
<code>fma(x, y, z)*C99</code>	$xy + z$
<code>drand48(), srand48()</code>	区間 $[0.0, 1.0)$ の疑似一様乱数とその初期化
<code>j0(x), j1(x), jn(n, x)</code>	第 1 種ベッセル関数。 0 次, 1 次, $n$ 次
<code>y0(x), y1(x), yn(n, x)</code>	第 2 種ベッセル関数。 0 次, 1 次, $n$ 次
<code>asinh(x)*C99, acosh(x)*C99</code>	$\operatorname{arsinh}(x), \operatorname{arcosh}(x)$
<code>atanh(x)*C99</code>	$\operatorname{artanh}(x)$
<code>cbrt(x)*C99, hypot(x, y)*C99</code>	$\sqrt[3]{x}, (x^2 + y^2)^{1/2}$
<code>exp2(x)*C99, expm1(x)*C99</code>	$2^x, e^x - 1$
<code>log1p(x)*C99, log2(x)*C99</code>	$\log(x + 1), \log_2(x)$
<code>logb(x)*C99</code>	$x$ の内部表現の指数部分
<code>lgamma(x)*C99</code>	$\log  \Gamma(x) $
<code>erf(x)*C99, erfc(x)*C99</code>	$\operatorname{Erf}(x) \equiv \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt,$ $\operatorname{Erfc}(x) \equiv 1 - \operatorname{Erf}(x)$

## 7 文字列

C 言語においては、数値について `int` や `double` のような基本型が定義されているのに比べて、文字列についての基本型が定義されていません。人間とのインターフェースとして最も重要と思われる文字列の扱いは、非常に簡素であり、低水準な関数のみを提供しています。C 言語はコンパクトな言語ですから、必要最小限の機能に限定した結果このような仕様になったのかもしれません。あるいはコンピュータの用途が、数値の計算に限られていた時代の名残なのかもしれません。もちろん、これらの低水準関数は必要十分な基本的関数であり、それらを用いて高水準の関数を組み上げていくところにプログラマの楽しみがあるともいえます。

### 7.1 文字列：char 型の配列

文字列は `char` 型の配列です。他の型の配列と異なる点は、**ナル文字 `'\0'` を終了の印としていること**です。文字列関数はナル文字をみつけると自動的にそこで処理を終えるという仕様となっているものが多いのです。したがって、配列の大きさを意識しないで扱うことができます。しかし、この便利な約束に頼りすぎるとバグを潜ませてしまうことがあります。すなわち、もし `'\0'` がなんらかの理由で消失した場合（これは往々にしてあるのです）には、文字列の終端を見つけれずにプログラムが暴走する危険性も生じるのです。

宣言と初期化については他の型の配列と同様です。例えば

```
char abc[4];
char message[5] = {'G','o','o','d','\0'};
char greeting[] = "Hello!" ;
```

となります。ナル文字で終端するの忘れないようにします。このことから判るように**文字列の長さを  $N$  とすると、それを納める `char` 型の配列のサイズは  $N + 1$  でなければなりません**。なお、最後の例の文字列リテラル（二重引用符で囲まれた文字列）は、見えませんが実は最後にナル文字が付加されています。したがって、もちろん `greeting` のサイズは  $6(\text{Hello!}) + 1(\text{'\0'}) = 7$  です。

要素は添字演算子を用いて表し読み書きします。上の例では `char` 型の配列 `abc` に要素を代入するには

```
a[0]='a';
a[1]='b';
a[2]='c';
a[3]='\0';
```

などします。最後にナル文字で終端することを忘れてはいけません。配列要素に文字 1 つずつ格納されている概念を下に頭示します。

a[0]	a[1]	a[2]	a[3]
a	b	c	'\0'

**例題 34** 文字列を宣言、初期化して表示するプログラムを実行してみなさい。

**解** `ex34.c` を編集してコンパイルしてください。

#### リスト 33 `ex34.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)
5 {
6     int i, len;
7     char str[] = "A quick brown fox jumps"
8     " over the lazy dog";
9
10    printf("str = '%s'\n", str);
11    len = sizeof(str) - 1;
12
13    for (i = 0; i < len; i++) {
14        printf("[%c]", str[i]);
15    }
16    printf("\n");
17
18    str[10] = '\0';
19    printf("str = '%s'\n", str);
20    str[10] = ' ';
21    printf("str = '%s'\n", str);
22    str[len] = '+';
23    printf("str = '%s'\n", str);
24
25    return (0);
26 }
```

7,8 行の初期化の文字列はページに収めるために 2 つにわけました。しかし、C 言語では、**連続した文字列リテラルは結合されます**から、このように記述しても、実際には 1 つの文字列リテラルとなります。10 行の `printf` 内の変換 `%s` は文字列（`char` 配列名）を指定します。11 行、`char` 配列の大きさを `sizeof` 演算子で取得しています。`char` が 1 バイトであるために、`-1 (''\0')` の分すれば文字列の長さとなります。13~15 行は、要素を 1 個ずつ `[]` で括って表示しています。18 行、中途に終端文字を代入すると、`printf` で `%s` 変換指定した際に文字列表示がそこで終了してしまいます。22 行では文字列の終端文字を消しています。これは少し危険な操作です。なぜならこうすると、`printf` はナル文字が現れるまで表示を続行しますから、どんな文字が現れるかは予測不能で、一般に画面が乱れますし、重傷の場合にはキー入力などが受け付けられなくなります。実行例を示します。

```
$ ./ex34
str = 'A quick brown fox jumps over a lazy dog.'
[A][ ][q][u][i][c][k][ ][b][r][o][w][n][ ]... 略
str = 'A quick br'
str = 'A quick br wn fox jumps over a lazy dog.'
@(' = 'A quick br wn fox jumps over a lazy dog.+
```

🔗 “A quick brown ...” は pangram と呼ばれ、日本語では「いろはにほへと...」に相当し、文字を（原則）1 回ずつ使って意味のある文章をつくる



言葉遊びです。

参考ウェブサイト：言葉遊び

- [パーフェクト・パングラム・デスク](#)
- [パズル遊びへの招待・オンライン版](#)

## 7.2 文字列の操作

数値の操作や数学関数に比べて文字列の操作は日常意識することがないので、どのように考えたらよいか目処すらたたないのではないかと思います。そこで、標準の文字列操作関数ライブラリを眺めながら、**文字列の操作**という事自体についても学んでいきましょう。

### 7.2.1 文字列の長さ

手始めは、配列サイズと関係の深い文字列の長さの取得です。終端文字 '\0' が必ずあるという前提にたてば、その 1 つ前までの要素数が文字列の長さということになります。すなわち、文字列の長さを `int length` に収めるには、次のような手続きで概ね良いことになります。

```
i = 0;
while (A[i] != '\0'){
    i++;
}
length = i;
```

文字列の長さを調べる関数は `strlen()` です。オンラインマニュアルでその仕様を調べてみた様子を以下に示します。

\$ man strlen		
STRLEN(3)	Linux Programmer's Manual	STRLEN(3)
名前	strlen - 文字列の長さを計算する	
書式	#include <string.h>  size_t strlen(const char *s);	
説明	strlen() 関数は文字列 s の長さを計算する。このとき、終端文字 '\0' は計算に含まれない。	
返り値	strlen() 関数は s の中の文字数を返す。	
準拠	SVID 3, POSIX, BSD 4.3, ISO 9899	
関連項目	string(3)	
April 12, 1993		1

`size_t` はそのコンパイラの実装によりますが、一般には `unsigned int` となる場合が多いようです。さらに関連項目の `string` を `man` で調べると、文字列操作関数全体の情報が得られます。

```
strcasecmp, strcat, strchr, strcmp,
strcoll, strcpy, strcspn, strdup,
strfry, strlen, strncat, strncmp,
strncpy, strncasecmp, strpbrk, strrchr,
strsep, strspn, strstr, strtok,
strxfrm, index, rindex
```

これを資料にして、次節以降で重要な操作の概念とその関数を説明します。

### 7.2.2 文字列の複写 (copy, duplicate)

文字列を複写するのに近道はありません。配列のコピーは全要素について複写元 (source) から複写先 (destination) に各々代入するしかありません。文字列を格納するための配列の大きさが `dest` に確保されていることを確認して、概ね以下のような処理を行うことになります。

```
char dest[128] = {'\0'};
char src[] = "source strings";
...
i = 0;
while ( src[i] != '\0' ) {
    dest[i] = src[i];
    i++;
}
dest[i] = '\0';
...
```

これと同じ処理を行う関数として、文字列操作関数ライブラリでは `strcpy()`、`strncpy()` という名前の関数が用意されています。man `strcpy` より関数の仕様がわかります。

```
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest,
               const char *src, size_t n);
```

引数はポインタとなっていますが、配列 `A[]` はその名前 `A` がポインタなので素直に名前を渡します。`dest` の大きさが `src` を収める程十分に確保されているかどうかは、文字列操作関数側でチェックできませんので、プログラマが管理しなければなりません。

### 7.2.3 文字列の比較 (compare)

文字列の比較とはどういうことでしょうか。等値ということの意味は自明でしょう。大小関係があるのでしょうか？一般には **アルファベット順** (日本語なら「あいうえお順」) を基準と考えるのが自然でしょう。“man `ascii`” で `ascii` コード表をみてください。大文字の `A` が一番小さく、小文字の `z` が一番大きい数値に割り当てられています。した

がって、char 型の変数を数値で比較した大小関係で整理すれば、小さい方から [A-Z][a-z] の順に並ぶこととなります。すなわち文字列 s1 と s2 を（数値で）比較するルーチンは概略

```
i = 0;
while (s1[i] == s2[i]) {
    if (s1[i] == '\0') break;
    i++;
}
result = (int)s1[i] - (int)s2[i];
```

のようになります。結果は result に反映されます（result が 0 より小さい、0, 0 より大きいがそれぞれ s1 が s2 より小さい、等しい、大きいという判定）。

**文字の分類** 文字列の順序が ascii コードでは大小比較と整合していましたが、そうなる保証を C 言語は規定していません。ただし、char コードの割り当ての実装にかかわらず、文字の分類を調べる関数や、大文字と小文字の変換関数を標準関数として定義しています。これらの関数を使うには ctype.h を include します。

表 14 文字分類テストの関数

関数	文字クラス
isalpha(c)	アルファベット
islower(c)	小文字
isupper(c)	大文字
isdigit(c)	10 進数
isxdigit(c)	16 進数
isalnum(c)	文字あるいは 10 進数
isprint(c)	表示可能文字（スペースも含む）
isgraph(c)	表示可能文字（スペースは除く）
isspace(c)	空白文字
ispunct(c)	記号 (!"#\$%&'()*+,-./:;<=>?@[\]^_`{ }~)
iscntrl(c)	制御文字
inblank(c)*C99	スペースあるいは水平タブ
isascii(c)*	7bit ASCII

返却値の型は int で true/false を返す。引数 c の型は char ではなく int。

\* BSD や SVID の拡張で GCC にも実装されている。

表 15 に大文字/小文字変換関数を示します。返却値は変換後の文字で、変換できなければ変換前の値（記号など）を返します。

**例題 35** 標準ライブラリの strlen(), strcpy(), strcmp() と自作の mystrlen(), mystrcpy(), mystrcmp() を比較してみましょう。

表 15 大文字/小文字の変換関数

関数	変換
tolower(c)	大文字から小文字
toupper(c)	小文字から大文字

[解] ex35u.c をダウンロードして編集・コンパイルしてください。

#### リスト 34 ex35.c

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int mystrlen(char a[])
5 {
6     int i = 0;
7
8     while (a[i] != '\0') {
9         i++;
10    }
11    return i;
12 }
13
14 int mystrcmp(char s1[], char s2[])
15 {
16     int i = 0;
17
18     while (s1[i] == s2[i]) {
19         if (s1[i] == '\0')
20             break;
21         i++;
22    }
23    return ((int) s1[i] - (int) s2[i]);
24 }
25
26 void mystrcpy(char dest[], char src[])
27 {
28     int i = 0;
29
30     while (src[i] != '\0') {
31         dest[i] = src[i];
32         i++;
33    }
34    dest[i] = '\0';
35 }
36
37 int main(void)
38 {
39     char src[] = "Dog as a devil deified,"
40     " lived as a god";
41     char cmp[] = "Dog as a devil deified,";
42     char dest[80] =
43     {'\0'};
44
45     printf("%d, %d\n", mystrlen(cmp), strlen(cmp));
46     printf("%s\n", dest);
47     mystrcpy(dest, src);
48     printf("%s\n", dest);
49
50     printf("%d, %d\n", mystrcmp(cmp, src),
51           strcmp(cmp, src));
52
53     return (0);
54 }
```

特に説明することはありません。my で始まる自作文字列操作関数を定義して、対応する標準関数と結果を比較しています。もちろん同じ結果を得ます。

```
$ ./ex35
23, 23
''
'Dog as a devil deified, lived as a god'
-32, -32  ← ナル文字 0(10) とスペース 32(10) との差
```

**問題34** strcmp() は辞書には向きません。なぜなら辞書は大文字小文字を区別しないからです。もちろん標準ライブラリには、既に大文字小文字を区別しない比較のために strcasecmp() があります。これと同様の機能を持つ mystrcasecmp() を考えてみなさい。q34.c

#### 7.2.4 文字列の連結 (concatenate)

文字列の基本型または文字列変数があって、それらへの代入と足し算があれば、文字列を扱うプログラムはかなりすっきりと書けそうです。実際、

```
$a = "文字列の"  ← C 言語では文字列変数はない
$b = "足し算"   ← また文字配列単位の代入もできない
$c = $a+$b      ← C 言語にはない演算
```

というような演算は、現在多くのスクリプト言語では定義されている、基本的な演算です。これが C 言語にはありません。標準文字列関数ライブラリの strcat() を使うことになります。cat はもちろん concatenate に由来しています。書式は

```
char *strcat(char *dest, const char *src);
```

となっており、文字列 dest の後ろに文字列 src を連結します。具体的な処理は dest の終端文字 '\0' を削除して、そこから src の内容を書き込むというものです。dest は src を連結してできる長い文字列が収まる大きさを事前に確保されていなくてはなりません。例によって、コンパイラは大きさに関するチェックをしてくれません。

**問題35** 文字列を連結する関数 mystrcat() を作成しなさい。それを標準文字列操作関数 strcat() と比較しなさい。q35.c

#### 7.2.5 文字列中の文字の検索

対象文字列中に、ある 1 文字が含まれているかどうかその位置を得る機能は文字列検索の基本です。標準文字列操作関数には、strchr, strrchr, index, rindex と複数用意されています。1 文字ではなく文字セットに属する 1 文字や文字列が含まれる位置を示す関数、strpbrk, strstr など標準関数として用意されています。

### 7.3 文字の入出力

いままでキーボードから文字列を入力する関数として scanf() を用いました。しかし、このような基本的と思われる事柄を扱う関数にしては、引数がポインタという仕様が初心者には違和感があります。そこで原始的ながら、判り易い 1 文字入出力関数 getchar(), putchar() を使ってみましょう。

**例題36** 1 文字入出力関数 getchar(), putchar() を用いて、標準入力（キーボード）から標準出力（画面）に文字をコピーするプログラムを作成しなさい。ただし、制御文字 BEL='\a' を受けたら終了するようにします。

[解] ex36.c に例を示します。

#### リスト35 ex36.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define QUIT '\a' /* ascii BEL */
5
6 int main(void)
7 {
8     int ch;
9
10    system("stty -icanon -echo");
11    while ((ch = getchar()) != QUIT) {
12        putchar(ch);
13        putchar('-');
14    }
15    system("stty sane");
16    printf("\n");
17    return (0);
18 }
```

9 行、getchar() は標準入力（一般にはキーボード）から 1 文字を unsigned char として読み込み、int にキャストして返します。その値を int 型変数 ch で受け取り、10 行の putchar(ch) で標準出力（一般には画面）に出力します。制御文字 BEL='\a' (kterm では Ctrl+G) が入力されたら while を抜けて終了します。実行例を示します。

```
$ ./ex36
abcdefGHIJKLM!"#$%&'()
  ← を入力したので改行した
^G  ← 制御文字 BEL の入力で終了
$
```

**問題36** 入力文字を大文字に変換して出力するプログラムを考えなさい。ただし、英数字以外の記号は全て ? に変換しなさい。q36.c

### 7.4 ライン入出力

文字単位ではなく、行単位（改行コード '\n' あるいは EOF (end of file) までの文字列）の入出力を行う原始的な関数を使ってみましょう。標準入出力に対しての gets(),

puts() と入出力先を指定できる fgets(), fputs() があります。gets() は、入力バッファの大きさを越える危険性があるので使わないこと。代わりに fgets() を使いなさいと man には書いてあります。

fgets() は、指定した入力行から文字列を読み込みますが、その大きさを制限することができて、バッファ溢れが起こる心配はありません。書式は以下のようになっています。

```
char *fgets(char *s, int size, FILE *stream);
```

読み込まれる文字数は高々 size - 1 に制限されます。入力ファイル (FILE \*stream) は今のところ stdin, すなわち標準入力を指定することにします。

**例題37** fgets() を用いて入力した行を逆さに表示するプログラムを考えなさい。入力文字数の最大値は 80 としなさい。

[解] ex37.c に fgets() を用いた例を示します。リスト36 ex37.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define BUFSIZE 81
6
7 int main(void)
8 {
9     int len, i;
10    char buf[BUFSIZE];
11
12    while (fgets(buf, BUFSIZE, stdin) != NULL) {
13        len = strlen(buf);
14        for (i = len - 2; i >= 0; i--) {
15            putchar(buf[i]);
16        }
17        putchar('\n');
18    }
19
20    return (0);
21 }
```

入力を stdin と記述すると標準入力から読み込むことになります。fgets() は、'\n' もバッファに読み込みます。したがって 14 行では改行文字をスキップするために (len-1)-1 から書き出しています。実行例を示します。

```
$ ./ex37
Dog as a devil deified, lived as a god
dog a sa devil ,deified lived a sa goD
Madam I'm adam
mada m'I madaM
$ ← [Ctrl] + [d] を入力して終了
```

終了は [Ctrl] + [d] あるいは [Ctrl] + [c] をキー入力してください。

**問題37** 前問に手を加えて (大文字小文字の区別をなくす, 句読点および空白を無視する等) 英語の回文を確かめるプログラムを考えなさい。q37.c

## まとめ

- C 言語に文字列型はない。文字列とは、char 型の配列であり、ナル文字 '\0' で終端する。
- 多くの文字列操作関数は、ナル文字 '\0' で終端していることを前提に作成されている。
- 文字列のために用意した char 型配列の大きさについては、プログラマが管理し、その範囲を越えた書き込みが発生しないように十分注意しなければならない。



## 8 ポインタ

C 言語のポインタはその習得が難しいことで有名で、ポインタを理解するための書籍も発行されているくらいです。素直に考えれば、そうまでして覚える価値があるということです。ポインタを活用するとコンパクトなプログラムを作成することができ、C 言語の特徴が際立ちます。もっと重要な点はポインタでなければ実現できない事柄もあるということです。したがって、入門段階においてもポインタは学習すべき必須項目なのです。

誤解を恐れずにいえば、**ポインタとは型とアドレスを表し、オブジェクトをアドレスを頼りにアクセスする場合に用いられるもの**といえます。いままで、後で詳しく説明するからと弁解しながらポインタを使ってきました。表 16 に整理します。これについて説明していきます。

表 16 オブジェクトとアドレス定数、ポインタ変数

オブジェクト	宣言	アドレス定数	定数ポインタ	定数ポインタの間接参照値
変数 x	x	&x	&x	x
配列 a	a[]	&a	&a	a のアドレス
関数 f	f()	&f	&f	f のアドレス
配列 a	a[]	a (=&a[0])	a (=&a[0])	a[0]
ポインタ型のオブジェクト	宣言	アドレス定数	ポインタ変数	ポインタ変数の間接参照値
変数 p	*p	&p	p	ポインタ先
配列 A	(*A)[]	&A	A	ポインタ先 (アドレス)
関数 F	(*F)()	&F	F	ポインタ先 (アドレス)

### 8.1 定数ポインタ

上段の単純におよび配列や関数として宣言したオブジェクトはアドレス演算子&を前置するとポインタを生成できます。これらの**オブジェクトのアドレスは変更できません**ので**アドレス定数**あるいは**定数ポインタ**と呼ばれます。すなわち、つぎのような代入演算はできません。

```
double x, y, a[10];

&x = &y;

&x = &a[1];

&a[1] = &x;

a = &x;
```

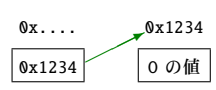
ポインタに**式中で間接演算子\***を前置すると、ポインタが指しているアドレスにあるデータ列を型通りに解釈して

値を取り出します。したがって定数ポインタ &x では \*(&x) = x となります。ただし、配列と関数は型に関係なく、自身のアドレス値が取り出されます。

### 8.2 ポインタ変数

アドレス演算子&によって生成される定数ポインタは定数ですからその値を変更することは許されません。それに対し、**宣言時に\*を前置したポインタ変数 P はアドレスを値にとります**。つまり代入演算子の左に置かれ、**値が変更可能**です。その値（代入された値）があるオブジェクト O のアドレスである場合に、間接演算子\*を前置した \*P は O を表します。というわけで、\*P は（同じ型の）オブジェクトの別名（alias）となりえるのです。

P:                      O:        \*P = \*0x1234  
                                      = 0x1234 にあるデータ  
                                      = 0 の値



\*address = address にあるデータ

#### 8.2.1 ポインタ、配列、関数の優先順位

基本型に\*, [], () を付けて宣言すると新しい型が無限に派生します。この3つの演算子には優先順位があり、\*は(), [] よりも優先順位が低いです。したがって、配列や関数へのポインタは名前との結び付きを優先するために()で括って(\*A)[], (\*f)()と書かなければなりません。もし、\*A[], \*f()と記述するとポインタを要素とする配列、ポインタを返す関数と解釈されます。これは全く別物です。

### 8.3 ポインタと配列名

さて混乱するのは配列です。**配列は（例外があります）式中ではその名前が先頭要素へのポインタと解釈されます**。したがって、配列名 a に間接演算子を作用させた \*a は a[0] となります。もちろん型は要素の型です。配列については別途説明します。

配列の定数ポインタ &a は a と同じ値を持ちます。実際、間接演算子を作用させると \*&a と \*a は等値です。しかし要素ではなく本来の意味通り配列全体を指していますので、型が異なるということになります。

☞ 実は、関数名も式中ではポインタと解釈されます。f(); と宣言された場合には、f は自身のアドレスを表します。ところが、アドレス演算子を前置した &f も同じ結果を得ますので混乱しません。

### 8.3.1 配列名が先頭要素へのポインタでない場合

配列名がその先頭要素へのポインタでない場合があります。

- sizeof 演算子のオペランドとなった場合。この場合には文字どおり配列の大きさが返ります。すなわち、例えば `char a[10];` と定義したならば `char` は 1 バイトですから `sizeof(a)` は 10 となります。
- アドレス演算子のオペランドとなった場合。これを例外というのか、本来の意味に立ち返るといふべきか判りませんが、配列全体と解釈されます。

## 8.4 ポインタ変数の初期化

関数内部で初期化せずに宣言した変数は値が不定となっています。もちろんポインタ変数も値=ポイント先（アドレス）が不定です。ポイント先がきちんと設定されていない場合には、アクセスにより大変な事態を招く可能性が高いです。OS が、許可されない記憶領域をアクセスしたと認識して Segmentation Fault で落としてくれるなら有難いくらいで、表面上何事もなかったように動いていて、その実オブジェクトを破壊しているなどという事になったら最悪です。

例えば、次のような使い方をしてはいけません。

```
double *px;          または double *px = 1.0;
*px = 1.0;
```

まず左側の場合。これでは、ポイント先が不定ですから、`*px = 1.0;` を行うと、一体どこに書き込んでいるのかわかりません（`px` の値をみればアドレスは判りますが、それは一般にはプログラムで認識できない場所です）。`px` 自身のアドレス `&px` に書き込むのではなく `px` がさす先の実体が必要なのです。次に右側の場合。これは 2 重の間違いをおかしています。一つは左側と同じくポイント先がないこと。もう一つは、代入すべき値がアドレスでなければならないのに、実数を渡していることです。宣言時の `*` は間接演算子ではありません。したがって、必ず次のようにします。

```
double rx;  ← 実体 rx をまず確保する
double *px;
px = &rx;  ← px のポイント先に rx を設定する
px = 1.0;  ← *px(=rx) への書き込み
```

あるいは、初期化も同時に行って

```
double rx;  ← 実体 rx をまず確保する
double *px = &rx;  ← 初期化しながら宣言
px = 1.0;  ← *px(=rx) への書き込み
```

ポイント先がしっかり確保されたオブジェクトとなるようにプログラミングしましょう。

**例題 38** 表 16 を確かめてみましょう。すなわち、(i) 単純に宣言した基本型の変数、配列、関数への定数ポインタの内容、(ii) ポインタ宣言した、基本型の変数、配列、関数への同じ型のオブジェクトのアドレス値代入と、それによって生まれる alias の機能を確認してください。

**[解]** `ex38.c` をダウンロードしてコンパイルしてください。

リスト 37 `ex38.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 double f(double x)
6 {
7     return (x * 100.0);
8 }
9
10 int main(int argc, char **argv)
11 {
12     double x = 1, a[3] = { 10, 20, 30 }, f();
13     double *px, (*pa)[3], (*pf) ();
14
15     printf("&x = %p, *(&x) = %f\n", &x, *(&x));
16     printf("&a = %p, *(&a) = %p\n", &a, *(&a));
17     printf("&f = %p, *(&f) = %p\n", &f, *(&f));
18     printf("a = %p, *a = a[0] = %f\n", a, a[0]);
19     px = &x;
20     pa = &a;
21     pf = &f;
22     printf("px = %p, *px = %f\n", px, *px);
23     printf("pa = %p, *pa = %p\n", pa, *pa);
24     printf("pf = %p, *pf = %p\n", pf, *pf);
25
26     printf("(*pf)(3.1) = %f\n", (*pf)(3.1));
27     printf("(*pa)[1] = %f\n", (*pa)[1]);
28
29     return (0);
30 }
```

繰り返しますが、ポインタ宣言した変数 `px`, `pa`, `pf` に普通に宣言したオブジェクト `x`, `a`, `f` のアドレス値を代入すると、`*px = x`, `(*pa) = a`, `(*f) = f` となっていることを確認してください。実行例を示します。

```
$ ./ex38
&x = 0xbffff9a4, *(&x) = 1.000000
&a = 0xbffff98c, *(&a) = 0xbffff98c
&f = 0x80483f0, *(&f) = 0x80483f0
a = 0xbffff98c, *a = a[0] = 10.000000
px = 0xbffff9a4, *px = 1.000000
pa = 0xbffff98c, *pa = 0xbffff98c
pf = 0x80483f0, *pf = 0x80483f0
(*pf)(3.1) = 310.000000
(*pa)[1] = 20.000000
```

### 8.4.1 関数へのアドレス渡し

C 言語では関数の引数が値渡しであることから、相互干渉のない独立した安全な関数を構成できるという特徴があります。このように入力と出力が完全に分離できているこ

とは大きな長所ですが、残念ながら出力たる返却値がやや単調で、複数の値を返すには構造体（次回のテーマです）に頼るしかありません。そこで、Fortran のように、ある変数を入出力に用いるために**アドレスの値を渡す**方法がとられ、ポインタ変数の活躍となります。第7回「関数」のところでとり上げましたが、大変重要な事柄ですから再度学習しましょう。

**例題 39** 整数配列を引数にとり、その最大値と最小値および平均値を返す関数を考えなさい。乱数を用いて4桁の整数を  $N$  個発生させて、この関数を呼び出して機能を確認するプログラムを作成しなさい。個数  $N$  は実行時に与えるようにしなくてもいいです。プログラム内で#defineによりマクロ定義する程度で結構。

[解] 例を ex39.c に示します。ex39u.c<sup>④</sup> をダウンロードして完成させなさい。

#### リスト 38 ex39.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define N 32
6
7 void stat(int a[], int n, int *max,
8          int *min, double *av)
9 {
10     int i, sum = 0;
11
12     *max = *min = a[0];
13     for (i = 0; i < n; i++) {
14         if (a[i] >= *max) *max = a[i];
15         if (a[i] <= *min) *min = a[i];
16         sum += a[i];
17     }
18     *av = (double) sum / n;
19 }
20
21 int main(int argc, char **argv)
22 {
23     int i, a[N], max, min;
24     double average;
25
26     srand(time(NULL));
27     for (i = 0; i < N; i++) {
28         a[i] = rand() % 10000;
29         printf("%4d, ", a[i]);
30         if (i % 8 == 7) printf("\n");
31     }
32
33     stat(a, N, &max, &min, &average);
34     printf("最大値は %4d, ", max);
35     printf("最小値は %4d です. \n", min);
36     printf("平均値は %.2f です. \n", average);
37
38     return (0);
39 }

```

説明しなくても読める事と思います。実行例を示します。本題とは関係ありませんが、発生させた整数を綺麗に表示するためにちょっと工夫してあります。

```

$ ./ex39
3431, 5071, 7164, 9095, 1030, 737, 3476, 6708,
440, 8053, 8050, 5000, 4735, 262, 3360, 1014,
8256, 3403, 5090, 3094, 2355, 3828, 4471, 2290,
9320, 9938, 9885, 6550, 4653, 207, 1865, 8085,
最大値は 9938, 最小値は 207 です.
平均値は 4716.12 です.

```

## 8.5 ポインタの演算と配列の添字演算子

ポインタはアドレスを値に持っていて、それに対していくつかの演算が行えます。まずは整数を足し算できます。すなわち、ポインタ  $p$  に対して

$p+1, p+2, p+3, \dots$

とすることができ、値（指しているアドレス）が増加します。この時に**ポインタの型のサイズを単位としていること**に注意してください。すなわち、char 型へのポインタなら1バイト、long int 型ならば4バイト、double 型ならば8バイトを単位に指しているアドレスが増加します。そして得られた結果もポインタとなっていますから、間接演算子で型にあてはめた値を取り出す事ができます。

$*(p+1), *(p+2), *(p+3), \dots$

さて、配列  $a[]$  の名前  $a$  が先頭要素  $a[0]$  へのポインタであるということから、 $a+i$  は  $a[i]$  へのポインタ  $\&a[i]$  となります。更に間接演算子を作用させれば

$a+i = \&a[i] \Rightarrow *(a+i) = *(\&a[i]) = a[i]$

が成立します。このように配列の添字演算はポインタおよびその整数演算に置き換えることが可能です。つまり、

$a[i] \equiv (*(a+i))$

となります（括弧を多用しているのはどのような場合でも等値となることを考慮した記述としたからです）。

④ 実際、GCC では全く同じ機械語を生成しています。ところで CPU が直接理解する機械語は数値ですから人間には大変読み難いので、通常一対一対応させた可読性の高いアセンブリ言語が開発現場では用いられます。C 言語のコンパイラはこの最終コードと呼べるアセンブリ言語に翻訳されたソースを生成させることができます。それにはオプション  $-S$  を指定します。例えば次の2つのソース

```

#include <stdio.h>
int main(void){ int a[5]; a[1]; return (0);}

#include <stdio.h>
int main(void){ int a[5]; *(a+1); return (0);}

```

をそれぞれ、 $-S$  付きでコンパイルすると拡張子  $s$  がついた  $****.s$  というファイルが生成されます。それを見比べて、中身が同じであることを確認してみてください。

**例題 40** 配列の添字演算とポインタに整数を加えて間接演算を作用させた結果が同じであることを確認しなさい。

[解] 例を `ex40.c` に示します。

リスト 39 `ex40.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     int i;
8     char ca[] = "xyz";
9     double xa[] = { 1, 3, 5 };
10
11     for (i = 0; i < 3; i++) {
12         printf("(%p): ca[%d] = %c, *(ca+%d) = %c\n",
13             ca + i, i, ca[i], i, *(ca + i));
14     }
15     for (i = 0; i < 3; i++) {
16         printf("(%p): xa[%d] = %.3f, *(xa+%d) = %.3f\n",
17             xa + i, i, xa[i], i, *(xa + i));
18     }
19
20     return (0);
21 }
```

素直に比較しているだけなので説明しなくてもいいでしょう。実行例は以下ようになります。もちろんアドレス値は、各自の実行環境によります。

```
$ ./ex40
(0xbffff9a4): ca[0] = x, *(ca+0) = x
(0xbffff9a5): ca[1] = y, *(ca+1) = y
(0xbffff9a6): ca[2] = z, *(ca+2) = z
(0xbffff98c): xa[0] = 1.000, *(xa+0) = 1.000
(0xbffff994): xa[1] = 3.000, *(xa+1) = 3.000
(0xbffff99c): xa[2] = 5.000, *(xa+2) = 5.000
```

### 8.5.1 ポインタの演算

文字列を扱う関数では伝統的に `char` 配列の添字演算を用いた記述ではなく、文字配列のポインタを用いた記述が好まれます。ポインタの整数演算の方が速度的に有利な時代があったというのも大きな理由です。例えば、文字列の長さを与える関数 `strlen()` は以下のように書いたりします。

```
int strlen(char *s)
{
    char *p = s;
    while (*p != '\0') p++;
    return p - s;
}
```

この例では、配列の添字演算に換えてポインタの増分演算 `p++` が使われています。その他にポインタ同士の引き算も行っています。ここで改めてポインタに対して可能な演算をまとめておきます。

- 同じ型のポインタ変数への代入、比較。定数ポインタへの代入は不可。
- ポインタ変数への整数の加減算。ただし、増減の単位(バイト数)はポインタの型に応じて決まります。

- 同じ型同士のポインタの差。意味があるかないかはプログラム次第です。
- 前項で同じ配列に属する要素へのポインタ `p1`, `p2` 同士ならば、その差 `p1 - p2` は間にある配列の要素数となります。

## 8.6 ポインタの配列

文字列のデータは一般にその長さが異なります。したがって、要素の大きさが固定の 2 次元配列では、データの最大長さを予測してサイズを確保しなければなりません。これでは非効率になるであろうことは明らかです。文字列へのポインタを配列にすることでこの問題は解決します。データベースなど、文字列を扱うデータ構造を必要とする場合にはポインタの配列や、ポインタを含む構造体(次週のテーマです)が多用されます。また、行列計算においても三角行列などに 2 次元配列を単純にあてはめたら、ほぼ半分が無駄であることも明らかでしょう。この場合にもポインタを活用すれば、要素数を任意にとれますから無駄が解消されます。

このようにポインタの配列は大変有用なオブジェクトです。ここでは、宣言時のリストによる初期化を使って大きさの異なる要素を持たせた `char` 配列と普通に宣言した 2 次元 `char` 配列との比較を行う例題を実行してみましょう。

**例題 41** `char` へのポインタの配列を文字列リストで初期化し、更にそのデータを 2 次元 `char` 配列にコピーして、それぞれのアドレス構造を表示するプログラムを作成しなさい。

[解] `ex41.c` をダウンロードしてコンパイルし、確かめなさい。

リスト 40 `ex41.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5
6 int main(int argc, char **argv)
7 {
8     char *ptr[] = { "A", "brown", "fox", "jumps", NULL };
9     char str[5][8];
10    int i;
11
12    for (i = 0; i < 4; i++) {
13        strcpy(str[i], ptr[i]);
14        printf("%p, %p: %s\n", ptr[i], str[i], str[i]);
15    }
16
17    return (0);
18 }
```

`char` へのポインタ配列 `ptr` は初期化時に代入された、(可変長) 文字列を要素としますから、アドレスの間隔は(文字列長さ+1)です。ところが、2 次元配列 `str` は長さ 8 の `char` 配列の配列ですから、常にアドレスが 8 ずつ変化してしまい、非効率的です。



実行例を示します。

```
$ ./ex41
0x804851c, 0xbf810094:      A
0x804851e, 0xbf81009c:    brown
0x8048524, 0xbf8100a4:     fox
0x8048528, 0xbf8100ac:    jumps
```

(文字列の長さ+1) ずつ変化      8 ずつ変化

### まとめ

- ポインタとは型とアドレスをもち、オブジェクトを指し示すものである。
- アドレス演算子により、オブジェクトのアドレスが得られる。
- アドレス定数あるいは定数ポインタへの代入はできない。
- ポインタ変数には同じ型のオブジェクトのアドレス値が代入でき、そのオブジェクトの別名となる。
- 配列名は、式中ではほとんどの場合先頭要素への定数ポインタとなる。
- a が配列の先頭要素へのポインタならば、整数 i に対して以下の関係が成り立つ。

$$a[i] = *(a+i)$$

## 9 構造体

構造体は、いろいろな型のデータをまとめて、新たな 1 つの型として定義する場合に用いられます。同じ型しか扱えない配列と違って**大変柔軟なデータ構造を構成することが可能です**。ただし、自分と同じ構造体をメンバにすることは不可で、同じ構造体へのポインタをメンバにすることのみできます。また、構造体は**全体としての代入が可能です**。したがって、(構造体で定義された) **複数の返却値を持つ関数を設計することができます**。すなわちポインタのアドレス値渡しといった手法を用いなくても、入出力が分離した安全な関数が作成できるのです。

### 9.1 構造体の宣言、初期化

構造体はキーワード `struct` を用いて以下のように宣言します。

```
struct [タグ名] {
    メンバの宣言リスト
} [オブジェクトのリスト];
```

タグ名とオブジェクトは省略可能ですが、同じデータ構造のオブジェクトを宣言することがあるでしょうからタグ名を必ず宣言するとよいでしょう。例えば

```
struct member {
    char name[40];
    unsigned short int age;
    double height;
} p1, p2;
```

ならば、3 つのメンバ `name`, `age`, `height` を持つデータ構造に**タグ名**を `member` と付け、`struct member` 型のオブジェクト `p1`, `p2` を宣言したことになります。この宣言があれば、同じデータ構造のオブジェクトを別の場所で

```
struct member  suzuki, satoh, oishi;
struct member  club[100], *who, func();
```

のように、`struct member` 型の配列や関数あるいはポインタ型のオブジェクトを宣言することができます。

宣言時に初期化することも可能です。

```
struct member matuda =
    {"Namio Matuda", 46, 166};
```

### 9.2 構造体の代入

構造体は初期化時以外にも代入が可能です。すなわち、既に `matuda` が値を持っていたとすると

```
club[0] = matuda;
```

により、matuda の各メンバの値が club[0] のメンバにそれぞれ一括して代入されます。ただし、初期化時のようなリストによる代入はできません。

```
club[1] = {"Namio Matuda", 46, 166};
```

✗ 宣言時以外は不可

## 9.3 メンバへの個別アクセス

### 9.3.1 単純に宣言した場合

構造体のあるメンバにアクセスするには、構造体変数名の後ろに、ドット演算子“.”を挟んでメンバ名を記述します。

|| 構造体変数名.メンバ

例えば、以下のように、個別にメンバの値を取り出したり、メンバへの値の代入ができます。

```
strcpy(matuda.name, "mazda");
matuda.age = 50;
club[1].age = matuda.age;
scanf("%lf", &club[1].height);
```

### 9.3.2 ポインタ型の場合

ポインタ変数として宣言された場合には、

|| (\*構造体へのポインタ名).メンバ

とすればよいのですが、煩わしいので別の記法があります。それはアロー演算子“->”を挟んでの記述です。

|| 構造体へのポインタ名->メンバ

例えば struct member \*who と宣言された場合には以下のように記述します。

```
strcpy(who->name, "Someone Else");
who->height = 203.2;
printf("%.2f\n", who->height);
```

**例題 42** 金属元素のデータ（原子番号、元素記号、名前、融点  $T_m$ 、電気伝導率  $\sigma$ 、熱伝導率  $\kappa$ ）ファイル element.dat を講義のページから前もって入手しておいてください。まず適当な構造体を定義しなさい。次に原子番号、元素記号、名前、融点を構造体に読み込む関数、構造体のメンバを表示する関数を作成しなさい。またそれらをデモするプログラムを考えなさい。なお、fgets(buffer, size, stdin) を用いて標準入力から読み込む設計とし、シェルのリダイレクト機能を活用することにします。すなわち、

ある command の標準入力から file のデータを読み込ませるには、以下のように実行することで実現します。

```
$ command <file ← 入力リダイレクト
```

この機能により、ファイルへの読み書きに関する切替えをプログラム側で設定する必要がなくなります。これは、Unix 系の OS の伝統的な手法です。

[解] 少し長いですが ex42.c に例を示します。

リスト 41 ex42.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 struct iccd {
6     int number;
7     char symbol[3];
8     char name[20];
9     double mt;
10    double ec;
11    double tc;
12 };
13
14 void print(struct iccd p)
15 {
16     printf("%2d %s\t%-8s\t%8.3f %8.2g %6.1f\n",
17           p.number, p.symbol, p.name, p.mt, p.ec, p.tc);
18 }
19
20
21 struct iccd getdat(char buf[])
22 {
23     struct iccd p;
24     sscanf(buf, "%d%s%lf%lf%lf", &p.number,
25           p.symbol, p.name, &p.mt, &p.ec, &p.tc);
26     return (p);
27 }
28
29 int main(void)
30 {
31     int i = 0;
32     char buf[80];
33     struct iccd element[100];
34
35     while (fgets(buf, 80, stdin) != NULL) {
36         element[i] = getdat(buf);
37         print(element[i]);
38         i++;
39     }
40     return (0);
41 }
```

22～28 行は構造体の返却値を持つ関数の定義です。関数の内部で同じ型の構造体を宣言し、そこにデータを文字列からの書式指定付き読み込み関数 sscanf() で読み込みます。構造体が return される際にメンバの値が一括コピーされ、呼び出し側に渡されます。37 行は返却値（構造体全体）を代入しています。36 行 fgets により、最大入力長さ指定して 1 行毎に char 配列 buf にデータ文字列を読み込み、getdat() に buf を渡しています。14～20 行、構造体のメンバの表示関数 print() については特に説明する必要がないでしょう。ex42u.c<sup>④</sup>

実行例を示します。

```
$ ./ex42 < element.dat
```

```
3 Li    Lithium      453.690  1.2e+07  82.0
4 Be    Beryllium    1551.000  3.7e+07  220.0
11 Na   Sodium       370.960  2.3e+07  125.0
12 Mg   Magnesium    921.950  2.5e+07  153.0
13 Al   Aluminium    933.520  4e+07    235.0
19 K    Potassium     336.800  1.6e+07  109.0
... (中略) ...
81 Tl   Thallium      576.650  6.7e+06  47.0
82 Pb   Lead           600.652  5.2e+06  35.0
83 Bi   Bismuth        544.450  1e+06    11.0
```

**問題38** 例題の構造体メンバの表示関数 `print()` の引数を構造体へのポインタ変数にして書き換えてみなさい。 [q38.c](#)

**問題39** 融点の値が 1000 K 以上のものを表示するプログラムを考えなさい。ただし、いったん全データを読み込んでから処理する手順としなさい。 [q39.c](#)

## 9.4 typedef

構造体の宣言はキーワード `struct` を含めると長くて煩わしくなってきます。そこで、型に名前をつけるために `typedef` を使います。例えば

```
typedef unsigned short int USHRT;
typedef struct {double x, y, z;} POINT3;
```

と型宣言すれば以降、`USHRT` を `unsigned short int` 型宣言の代わりに、3つの実数メンバを持つ構造体の型の宣言に `POINT3` を用いることができます。

```
USHRT i, j[100], f(...);
POINT3 point, *pPoint
```

## 9.5 3次元ベクトルへの応用

構造体を用いた典型的な数値計算上の応用例は、3次元空間上の計算です。次のような3次元ベクトル（デカルト座標系）の構造体を定義することは、ごく自然な発想です。

```
typedef struct {double x, y, z;} Vec3;
```

すると、`Vec3` で宣言された2つのベクトルの和は、

```
Vec3 vadd(Vec3 v1, Vec3 v2)
{
    Vec3 v;
    v.x = v1.x + v2.x;
    v.y = v1.y + v2.y;
    v.z = v1.z + v2.z;
    return (v);
}
```

と定義できます。この調子で内積や外積を関数定義したくなるでしょう。頻繁に使う関数を再利用するには、ライブラリにまとめるというのが1つの手です。標準ライブラリはその例です。ライブラリを作成するのはそれなりの手間がかかりますから、ここではヘッダーファイルにまとめて `#include` することにしましょう。この方法はあまり分量が多くない場合には良く行われます。以下に示すように、3次元ベクトルを操作する関数の定義を `vector.h` として保存します。 [vector.h](#)

### リスト42 vector.h

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 #define deg2rad(deg) ((deg)*M_PI/180)
6 #define rad2deg(rad) ((rad)*180/M_PI)
7
8 typedef struct {
9     double x, y, z;
10 } Vec3;
11 typedef struct {
12     double r, t, p;
13 } Pol3;
14
15 Vec3 vadd(Vec3 a, Vec3 b)
16 {
17     Vec3 v;
18     v.x = a.x + b.x;
19     v.y = a.y + b.y;
20     v.z = a.z + b.z;
21     return (v);
22 }
23
24 Vec3 vsub(Vec3 a, Vec3 b)
25 {
26     Vec3 v;
27     v.x = a.x - b.x;
28     v.y = a.y - b.y;
29     v.z = a.z - b.z;
30     return (v);
31 }
32
33 double viprod(Vec3 a, Vec3 b)
34 {
35     return a.x * b.x + a.y * b.y + a.z * b.z;
36 }
37
38 Vec3 voprod(Vec3 a, Vec3 b)
39 {
40     Vec3 c;
41     c.x = a.y * b.z - a.z * b.y;
42     c.y = a.z * b.x - a.x * b.z;
43     c.z = a.x * b.y - a.y * b.x;
44     return (c);
45 }
46
47 double vabs(Vec3 a)
48 {
49     return sqrt(viprod(a, a));
50 }
51
52 double varg(Vec3 a, Vec3 b)
53 {
54     return acos(viprod(a, b) / (vabs(a) * vabs(b)));
55 }
56
57 Vec3 pol3tovec3(Pol3 pv)
58 {
59     Vec3 v;
60     v.x = pv.r * sin(pv.t) * cos(pv.p);
61     v.y = pv.r * sin(pv.t) * sin(pv.p);
62     v.z = pv.r * cos(pv.t);
63     return (v);
64 }
```

関数の説明を少し。voprod() はベクトル外積  $\vec{c} = \vec{a} \times \vec{b}$  を得る関数です。ベクトル外積は

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

$$|\vec{a} \times \vec{b}| = |\vec{a}| |\vec{b}| \sin \theta$$

となりますから、それを忠実に表しています。ここに  $\theta$  はベクトル  $\vec{a}$ ,  $\vec{b}$  のなす角度です。明らかに、 $\vec{a}$ ,  $\vec{b}$  を辺とする三角形の面積は  $\frac{1}{2} |\vec{a} \times \vec{b}|$  で計算されます。

pol3tovec3() は極座標系  $(r, \theta, \phi)$  からデカルト座標系  $(x, y, z)$  への変換です。Pol3 型は Vec3 型と構造は同じですが、メンバの名前が異なります。

**例題43** 1 辺の長さ 10 の立方体内にランダムに整数値を座標とする 3 点 A,B,C を生成し、三角形 ABC の面積を求めるプログラムを作成しなさい。ただし、辺の長さ  $a, b, c$  より求めた値と、ベクトル外積から求めた値を比較しなさい。

[解] ex43.c に例を示します。ex43u.c<sup>Ⓔ</sup> をダウンロードして完成させなさい。

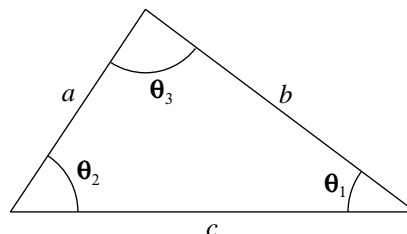
#### リスト43 ex43.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <time.h>
5 #include "vector.h"
6
7 int main(int argc, char **argv)
8 {
9     int i;
10    double s1, s2, a, b, c, s;
11    Vec3 v[3], va, vb, vc;
12
13    srand(time(NULL));
14    for (i = 0; i < 3; i++) {
15        v[i].x = rand() % 10;
16        v[i].y = rand() % 10;
17        v[i].z = rand() % 10;
18        printf(
19            "v[%d].x = %f, v[%d].y = %f, v[%d].z = %f\n",
20            i, v[i].x, i, v[i].y, i, v[i].z
21        );
22    }
23    va = vsub(v[1], v[0]);
24    vb = vsub(v[2], v[1]);
25    vc = vsub(v[0], v[2]);
26    a = vabs(va);
27    b = vabs(vb);
28    c = vabs(vc);
29    s = (a + b + c) / 2;
30
31    s1 = vabs(voprod(va, vb)) / 2;
32    s2 = sqrt(s * (s - a) * (s - b) * (s - c));
33    printf("面積1 = %f, 面積2 = %f\n", s1, s2);
34
35    return (0);
36 }
```

**問題40** 三角形における以下の正弦定理を  $xy$  平面内で数値的に確かめるプログラムを考えなさい。

$$\frac{a}{\sin \theta_1} = \frac{b}{\sin \theta_2} = \frac{c}{\sin \theta_3}$$

ここに、 $\theta_1, \theta_2, \theta_3$  は、辺  $a, b, c$  とそれぞれ向き合う頂角です。q40.c<sup>Ⓔ</sup>



## 9.6 構造体の応用例：struct tm

Unix は内部に時計を持っており、世界共通時 (UTC) 1970 年 1 月 1 日 00:00:00 からの経過秒数を基にして、種々の時間管理を行っています。今まで、time(NULL) を用いてこの経過秒数を取得し、乱数の初期化に利用しました。経過秒数を引数にして、時刻を計算する関数が何種類あります。“man time”で調べると、struct tm なる構造体が定義されていることが判るでしょう。

```
struct tm
{
    int    tm_sec;           /* 秒 */
    int    tm_min;          /* 分 */
    int    tm_hour;         /* 時間 */
    int    tm_mday;         /* 日 */
    int    tm_mon;          /* 月 */
    int    tm_year;         /* 年 */
    int    tm_wday;         /* 曜日 */
    int    tm_yday;         /* 年内通算日 */
    int    tm_isdst;        /* 夏時間 */
};
```

経過秒数 (long int) からこの構造体を得るための変換関数には gmtime(), localtime() があります。すなわち、

```
time_t now;
struct tm *nowtm;
...
now = time(NULL);
nowtm = localtime(&now)
```

とすれば、地方時の時間要素をメンバにする構造体へのポインタ nowtime が得られます。したがって、時刻の時間単位部分は nowtm->hour で取り出すことができます。

**例題44** 起動時の時間に関する構造体 struct tm を time(), localtime() を用いて取得し、日付と時刻を表示するプログラムを考えなさい。

[解] ex44.c<sup>Ⓔ</sup> をコンパイルして確かめてください。

#### リスト44 ex44.c



```

1 #include <stdio.h>
2 #include <time.h>
3
4 int main(void)
5 {
6     time_t now;
7     struct tm *nowtm;
8
9     now = time(NULL);
10    nowtm = localtime(&now);
11
12    printf("只今の時刻は %d年%02d月%02d日"
13           " %02d:%02d:%02d です.\n",
14           nowtm->tm_year + 1900, nowtm->tm_mon + 1,
15           nowtm->tm_mday, nowtm->tm_hour,
16           nowtm->tm_min, nowtm->tm_sec);
17    printf("%s", ctime(&now));
18
19    return (0);
20 }

```

年数は 1900 年からの経過年数ですから 1900 を足します。月も 0 から始まっているので+1 しなければなりません。カレンダー文字列（改行コード付き）に変換する関数 `ctime()` もついでに使ってみました。実行例は必要ないでしょう。

🕒 時間に関しては、時刻も重要でしょうが、待時間などの設定も欲しい機能の一つです。Unix では、ユーザーが簡単に使える、待時間の設定の関数 `usleep()` があります（`<unistd.h>` が必要）。また BSD 由来の秒よりも細かい時間の取得関数 `ftime()` も存在します（`<sys/timeb.h>` が必要）。簡単なプログラムでこの機能を確認してみてください。notes9-6.c

リスト 45 notes9-6.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <time.h>
5 #include <sys/timeb.h>
6
7 int main(int argc, char **argv)
8 {
9     struct tm *nowtm;
10    struct timeb nowb;
11    int interval = 300000;
12
13    if (argc > 1)
14        interval = atoi(argv[1]);
15    while (1) {
16        ftime(&nowb);
17        nowtm = localtime(&nowb.time);
18        printf("%02d:%02d %03d\n", nowtm->tm_min,
19               nowtm->tm_sec, nowb.millitm);
20        usleep(interval);
21    }
22    return (0);
23 }

```

実行例は以下のようになります。引数に与えた間隔（単位  $\mu\text{s}$ ）（指定なしでは既定値 0.3 s）に時刻を表示します。終了は `Ctrl`+`C` です。

```

$ ./a.out
19:15 282
19:15 584
19:15 894
...

```

## 10 プリプロセッサ命令

C 言語ではソースを実際にコンパイラにかける前に、コメント文を除去したり、文字列の置換を行うなどの前処理

（preprocess）を実行します。この機能は移植性（portability）を保つ意味でも重要なものです。前処理を行う **プリプロセッサ** に対する**指令**（directive）は、形式が定まっています。行頭が # で始まり、プリプロセッサ指令のみで構成されていなければなりません（コメントは記述可）。この節では、いままで使ってきた `#include` と `#define` の使い方について詳しく説明します。他にも `#if` `#elif` `#else` `#endif` や `#ifdef`, `#ifndef` 等々高度な指令がありますが、省略します。

### 10.1 #include

指定したファイルの内容をその位置に読み込ませる指令です。講義の最初からおまじないとして使ってきました。

```

#include <stdio.h>
#include <math.h>
#include "vector.h"

```

アングル <> で囲まれている場合には、プリプロセッサは予め指定されているディレクトリのみ探索します。また、二重引用符 "" で囲まれた場合には、現在の作業ディレクトリを探索します。

標準関数のプロトタイプ宣言や重要な定数の定義などを汎用的に使うために、なくてはならない指令です。

### 10.2 #define

マクロ（文字列の置換）定義に用います。以下のような構文をとります。

```
|| #define 名前 置換後の文字列
```

例えば、単純なものは

```
#define N 128
```

と定数を論理上のシンボル名で用いたい場合などに利用します。これは**シンボル定数**と呼ばれます。変数名と同じく、1つのまとまりとして認識されますから、`N+1` は `128+1` と置換されますが、`NN` とあるからといって `128128` には置換されません。複雑なものでは

```
#define deg2rad(a) ((a)*M_Pi/180)
```

のような**関数形式**のものがあります。マクロ定義は、関数呼び出しのオーバーヘッドがないので高速な動作が期待できますから、標準関数の中には実際には関数定義ではなくマクロ定義されているものもあります。また、引数の型宣言が不要なので型に依存しない汎用の処理を記述できます。このようにマクロはたいへん重宝な指令です。

しかし、定義が不十分ですと思わぬ置換結果が生じたり、**副作用**がありますから注意しなければなりません。例えば、

```
#define deg2rad(a) a*M_Pi/180
```

と定義したとします。するとソース中に `deg2rad(x+1)` や `1/deg2rad(x)` と記述された場合には、忠実に置き換えられた結果

```
deg2rad(x+1) --> x+1*M_PI/180
1/deg2rad(x) --> 1/x*M_PI/180
```

となり、もちろん、意図とは全く違った式となります。このように、**括弧を付けることを忘れてはいけません**。もう一つ有名な副作用は増分演算子を使った場合に生じます。例えば

```
#define sqr(a) ((a)*(a))
```

と定義すると、ソース中の `sqr(x++)` は

```
sqr(x++) --> ((x++)*(x++))
```

となり、`x++` が 2 回実行されてしまいます。関数形式マクロの定義は、最初のうちは定評のあるものを模倣するだけに留めるのが安全でしょう。

## 11 ファイル入出力

入出力機能について C 言語の規定はありません。しかし、実際には入出力はプログラムにつきものです。C 言語では入出力の言語仕様ではなく、**入出力標準関数の仕様を厳密に定義しました**。外部（周辺デバイス）との入出力は適当なバッファをおいたデータストリーム（stream）を通じて行くと想定されています。オープンに成功したファイルには新しいストリームとそれに関する情報を持った **FILE 構造体** とが与えられます。FILE 構造体には、バッファのアドレスや、残っているバイト数、ファイルの終端に達したかどうかなどの状態に関するフラグなどが含まれます。

そして何もしなくても標準入力（stdin）、標準出力（stdout）、標準エラー（stderr）はデフォルトで必ずオープンして確保され、通常 stdin はキーボードに、stdout と stderr は画面にそれぞれ接続されます。

標準以外のファイル、例えばハードディスクに作成されるファイルとデータをやりとりするためには、まずファイル名と読み書きのモードを引数として `fopen()` 関数を呼び、FILE 構造体へのポインタ（**ファイルポインタ**と呼ばれます）を取得する必要があります。入出力関数は、このファイルポインタをファイルの識別に用います。

### 11.1 FILE 構造体へのポインタ取得：fopen()

表 17 fopen() 関数の mode

mode	意味
"r"	読み取り。ファイルが存在しない場合には失敗する（NULL が返る）。
"r+"	読み取り+書き込み。ファイルが存在しない場合には失敗する（NULL が返る）。
"w"	書き込み。ファイルが存在する場合には上書き。ファイルが存在しない場合には新たに作成する。
"w+"	書き込み+読み込み。ファイルが存在する場合には上書き。ファイルが存在しない場合には新たに作成する。
"a"	書き込み。ファイルが存在する場合には追加。ファイルが存在しない場合には新たに作成する。
"a+"	書き込み+読み込み。ファイルが存在する場合には追加。ファイルが存在しない場合には新たに作成する。

r, r+, w, w+, a, a+ の後に、t（テキスト）あるいは b（バイナリ）を指定することも可能です。ただし UNIX 系 OS ではこれは意味を持たないので単に無視されます。

ファイルへのアクセスの第一歩は、FILE 構造体へのポインタを取得することにあります。ファイル構造体へのポインタ変数を宣言し、`fopen()` により得られた有効なポインタを代入します。

```
FILE *fp;
fp = fopen(name, mode);
```

#### まとめ

- 任意の型のデータをまとめて 1 つの型として定義するには、構造体を用いる。構造体はキーワード `struct` で宣言を開始する。

```
struct [タグ名] {
    メンバの宣言リスト
} [オブジェクト];
```

- 構造体は、配列とは異なり、全体としての代入ができる。これは、関数の引数あるいは返却値の場合にも適用される。したがって、複数の値を返すことができる関数を容易に作成することが可能である。
- `typedef` 宣言により、複雑な型に別名を与えることができる。

```
typedef 複雑な型宣言 別名
```

ここに name は char 配列へのポインタすなわちファイル名です。また mode は表 17 に示される文字列へのポインタで、ファイルへのアクセス方式を指定します。オープンに成功すればファイルポインタが得られ、失敗すると NULL が返ってきます。ファイルのアクセスの失敗は既存データの消失などの重大な障害を引き起こす可能性がありますから、必ずエラーチェックを行うようにします。

### 11.2 ファイルのクローズ: fclose()

オープンしたファイルへの操作が終了したら、fclose(fp) でクローズするのがお作法です。fp はオープンしたファイルポインタです。クローズする際には、バッファリングされていたデータが fflush() 関数により書き込まれますから、安全に終了するためには必ず fclose しましょう。

### 11.3 文字および文字列の入出力関数

表 18 に整理した入出力関数はオープンに成功したファイルポインタ fp を引数にとって、文字や文字列を読み書きします。

表 18 文字および文字列入出力関数

関数名	機能	エラー
int fgetc(fp)	1 文字読み込み	EOF
int getc(fp)	1 文字読み込み	EOF
int getchar()	getc(stdin)	EOF
int ungetc(c, fp)	1 文字 (c) の書戻し	EOF
char *fgets(s, n, fp)	1 行読み込み	NULL
char *gets(s)	1 行読み込み	NULL
int fputc(c, fp)	1 文字 (c) 書込み	EOF
int putc(c, fp)	1 文字 (c) 書込み	EOF
int putchar(c)	putc(c, stdout)	EOF
int fputs(s, fp)	1 行書込み (改行なし)	EOF
int puts(s)	stdout に 1 行書込み (改行あり)	EOF

fp はファイル構造体へのポインタ、c は int 型、s は文字配列へのポインタ、n は int 型

### 11.4 出力バッファの掃き出し: fflush()

ストリームは通常バッファリングされていますから、fputc() などを用いて出力処理を行ったつもりでも、あるサイズに達するまであるいは改行コードを受けるまで、実際にはファイルに書き込まれない場合があります。fflush(fp) はバッファに残っている出力データをその時点でファイルに掃き出す関数です。

**例題 45** 前回用意したデータファイル "element.dat" を読み込みモードでオープンし、1 文字ずつ読んで画面表示するプログラムを作成しなさい。

リスト 46 ex45.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     int c;
9     char infile[] = "element.dat";
10    FILE *fp;
11
12    if ((fp = fopen(infile, "r")) == NULL) {
13        printf("Can't open %s\n", infile);
14        exit(0);
15    }
16    while ((c = fgetc(fp)) != EOF) {
17        fputc(c, stdout);
18        fflush(stdout);
19        usleep(100000);
20    }
21    fclose(fp);
22
23    return (0);
24 }
```

[解] ex45.c が解答例です。12～15 行、fopen() 関数を用いて、名前 "element.dat" のファイルを読み込みモード "r" でオープンすることを試み、失敗ならばその旨を表示して中断します。17～21 行、fgetc(fp) で 1 文字を読み込み、fputc(c, stdout) で標準出力に書き出しています。もちろん、fgetc() の代わりに getc(), fputc() の代わりに putc() を用いてもかまいません。19,20 行は、1 文字ずつの読み書きを強調するために、fflush() で 1 文字を強制的に吐き出し、usleep() で待時間を設けてゆっくりと表示させるようにしています。表示がゆっくりすぎて終了まで我慢できないならば、**Ctrl** + **C** で中断できます。

### 11.5 内部表現のままのデータ保存

printf() 系関数は講義の最初の頃から使ってきました。その主な機能は、数値を人間にとって可読な文字列に変換することです。しかし、そのためにバイト数は増加するのが普通です。例えば、どんなに大きな数についても double 型は一律 8 バイト (64 ビット) ですが、例えば 1.2345678903456e-10 は 20 文字ありますから、20 バイト、すなわち 8 バイトに比べて 2 倍強の容量が必要です。ファイルの場合も同様で、一端文字列に変換してから保存するとそのサイズが 2 倍以上になることは当然の結果です。そこで、科学技術分野での実験データのように、形式が整った膨大なデータなどは内部形式のまま保存することが行われます。このような**バイナリ形式のデータ**はもちろんデータ構造が予め判っていなければ読む事ができません。そこで、相互に利用するためにはバイナリデータ構造の共通の規約が必要になります。また自分でバイナリデータを保存形式にした場合でも、ファイルの始まりの部分にデータ構

造の情報をテキストとして記録しておき、いわゆる自己記述 (Self-Discribing) となるようにするなどの工夫が必要となります。

### 🔗 参考ウェブサイト: 著名なバイナリデータフォーマット

- [HDF \(Hierarchical Data Format\)](#)
- [NetCDF \(network Common Data Form\)](#)

## 11.6 内部表現のままの入出力関数: fread(), fwrite()

fread(), fwrite() は数値を内部表現のままファイルに読み書きする関数です。仕様は以下のようになっています。

```
size_t fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream)
size_t fwrite(void *ptr, size_t size,
             size_t nmemb, FILE *stream)
```

fread() は、ストリーム stream から size の大きさを持つオブジェクトを nmemb 個読み込み、ポインタ ptr で指定されたアドレスを先頭にして格納します。fwrite() は、ポインタ ptr で指定されたアドレスから始まる、size の大きさを持つオブジェクトを nmemb 個読み込み、ストリーム stream へ書き込みます。オブジェクトへのポインタが void となっていてその型が不明ですから、明示的にサイズ (型) を与えなければならない点に注意してください。

**例題 46** 桁数の長い実数を 2 つのファイルに形式を変えて (テキストとバイナリ) 保存するプログラムを作成しなさい。また、作成されたファイルの大きさを比較しなさい。

**[解]** ex46.c に解答例を示します。22 行において、double 型の変数 x の文字列変換結果をファイルポインタ fpc すなわち名前 "char.dat" のファイルに書き出しています。23 行では、fcb すなわち名前 "bin.dat" のファイル に対し変数 x を double 型の内部 (バイナリ) 形式で 1 つずつ書き出しています。ex46u.c

実行後に出来上がった、2 つのファイルの大きさを比較したのが以下の画面です。

```
$ ./ex46
$ ls -l char.dat bin.dat

-rw-r--r-- ... 8192  6月 28日 23:36 bin.dat
-rw-r--r-- ... 20394 6月 28日 23:36 char.dat
```

bin.dat は当然ながら char.dat よりもサイズが小さくなっており、保存効率がよいといえます。ただし、less などのツールで内容を確認することはできません。char.dat はサイズは大きいのですが、内容が確認できます。

### リスト 47 ex46.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
```

```
4
5 int main(void)
6 {
7     double x;
8     char filec[] = "char.dat";
9     char fileb[] = "bin.dat";
10    FILE *fpc, *fcb;
11
12    if ((fpc = fopen(filec, "w+")) == NULL) {
13        printf("Can't open %s\n", filec);
14        exit(0);
15    }
16    if ((fcb = fopen(fileb, "w+")) == NULL) {
17        printf("Can't open %s\n", fileb);
18        exit(0);
19    }
20    for (x = 0; x < 1024; x++) {
21        fprintf(fpc, "%.15f\n", x);
22        fwrite(&x, sizeof(double), 1, fcb);
23    }
24    fclose(fpc);
25    fclose(fcb);
26
27    return (0);
28 }
```

**問題 41** 例題で作成した 2 つのファイルを読み込んで表示するプログラムを考えなさい。

**41の解** q42.c が解答例です。fprintf() に対応するのは fscanf(), fwrite() には fread() が対応します。q42.c

### リスト 48 q42.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main(void)
6 {
7     int i;
8     double xc, xb;
9     char filec[] = "char.dat";
10    char fileb[] = "bin.dat";
11    FILE *fpc, *fcb;
12
13    if ((fpc = fopen(filec, "r")) == NULL) {
14        printf("Can't open %s\n", filec);
15        exit(0);
16    }
17    if ((fcb = fopen(fileb, "r")) == NULL) {
18        printf("Can't open %s\n", fileb);
19        exit(0);
20    }
21    for (i = 0; i < 1024; i++) {
22        fscanf(fpc, "%lf\n", &xc);
23        fread(&xb, sizeof(double), 1, fcb);
24        printf("xc = %4.1f, \txb = %4.1f\n", xc, xb);
25    }
26    fclose(fpc);
27    fclose(fcb);
28
29    return (0);
30 }
```

## 12 プロセス制御と system()

プログラムがメモリにロードされ実行されている状態は **プロセス** と呼ばれます。プロセスへのメモリの割り当てや実行スケジュールは OS が管理しており、ユーザーが関与



できることはほとんどありません。しかし、プログラム中で他のプロセスを起動することを OS に要求することは可能です。いくつかの関数の中で、最も簡便な関数 `system()` を使ってみましょう。

`system()` はシステムのコマンドインタプリタ（例えば Unix 系ならば `/bin/sh -c`）を起動して、文字列で与えられたコマンドを実行します。すなわち

```
|| system("コマンド")
```

と記述するだけで、“コマンド”が実行されます。完成したいろいろなコマンドを呼び出せるので大変便利ですが、セキュリティ上の問題が発生する余地がありますので、少なくとも**絶対パス名**でコマンドを指定するなど、十分注意を払う必要があります。

**例題 47** `system()` 関数を用いて、カレンダーを表示するコマンド `cal` を呼び出してみなさい。

リスト 49 ex47.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     system("/usr/bin/cal");
7     return (0);
8 }
```

実行結果を以下に示します。

```
$ ./ex47
      6 月 2003
日 月 火 水 木 金 土
1  2  3  4  5  6  7
8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

## 13 複素数の計算 C99

科学技術計算に関して C 言語の弱点は複素数でした。C++ でないと複素数の計算を違和感なく記述することはできなかったのです。ところが、C99 の拡張では複素数型が導入されたいへん簡単に扱えるようになりました。GCC でも実装が進んでいますので触ってみましょう。

### 13.1 complex.h と tgmath.h

複素数を扱うには上記見出しにある 2 つのヘッダーファイルを `#include` します。complex.h にはその名の通り複素数型に関する定義がまとめられています。tgmath.h には**型汎用マクロ**が定義されていて、異なる型（double と complex）の数学関数を同じ名前呼び出すことができるようにしています。例えば、正弦関数は `double sin(double)` と `complex csin(complex)` があるのですが、このような使い分けは面倒ですから、どちらにも `sin()` で呼び出せるように統合名を定義しているのです。

**例題 48** 複素数の四則計算、数学関数の呼び出しを以下のプログラムで確かめなさい。

**[解]** [ex48.c](#) をダウンロードして確かめてください。

リスト 50 ex48.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <complex.h>
5 #include <tgmath.h>
6
7 #define NCprintf(a) (printf("#a " = ""), Cprintf(a))
8
9 void Cprintf(complex double z)
10 {
11     printf("%f %fi\n", creal(z), cimag(z));
12 }
13
14 int main(void)
15 {
16     double complex z = 1 + 2i, c;
17
18     NCprintf(z);
19     NCprintf(z * z);
20     NCprintf(I);
21     NCprintf(z * I);
22     NCprintf(z + (4 + 2i));
23     NCprintf(conj(z));
24     printf("cabs(z) = %f\n", cabs(z));
25     printf("carg(z) = %f\n", (double) carg(z));
26     NCprintf(sin(z));
27     NCprintf(exp(z));
28     c = 3 + 4i;
29     NCprintf(c);
30
31     return (0);
32 }
```

`complex double` 型で宣言したオブジェクトの実部と虚部はそれぞれ `creal()` と `cimag()` で取得することができます。ただし、gcc.2.95.3 では、式中で同時に使うことが

できないようです。純虚数単位がマクロ定数 `I` で定義されています。また、 $3 + 4i$  のように自然な記述が可能です。26 行の偏角は複素数に特有ですから `cargs()` しか存在しません。29 行、初期化時以外でも代入可能です。

7 行目でプリプロセッサ指令 `#a` を用いています。#演算子は、**文字列化演算子**(stringizing operator) と呼ばれ、引数を引用符で囲んで文字列リテラルに置換します。したがってこの場合には、

```
printf(#a " = "),
→ printf("a" " = "),
→ printf("a = "),
```

のようにソースが変換されます。最後の変換は、**連続した文字列リテラルは連結される**という規則に従ったものです。

## まとめ

- ファイルへの読み書きは `FILE` 構造体へのポインタ (ファイルポインタ) が指す、ストリームを通じて行われる。ファイルポインタを取得する関数は `fopen()` である。
- ファイルへのデータの保存形式はテキスト (文字列) が普通であるが、`fread()`、`fwrite()` を用いて整数や実数の内部表現形式のままバイナリで保存することも可能である。バイナリはテキストに比べて、サイズが小さくなるのが一般的であり有利な点である。しかし、保存時のデータ構造が予め判っていないと後に読むことができないので工夫が必要である。

## 14 アルゴリズムの学習

### 14.1 整列 (sorting)

数値ならば大きさの順、文字列ならば辞書順 (あるいは逆順) に並べ換える作業を整列といいます。整列は応用が広く、古くから様々なアルゴリズムが考え出されてきました。大規模なデータを高速に整列するためには、配列や構造体あるいはポインタを用いた工夫を凝らす必要があります。C 言語 (あるいは遍くプログラミング言語の) の文法のおさらいには格好の題材です。

さて以下に `sort` のいろいろなアルゴリズムを試してみますが、共通部分がありますから、それを `"sort.h"` にまとめておきます。具体的には、

- 要素数  $N$  のランダムな `int` 型数値データ配列をメモリを確保したのち初期化して、その先頭要素へのポインタを返却する関数：
- ```
int *initdata(int n)
```
- `int` 型配列の先頭および末尾を表示するための関数：
- ```
void printdata(int *p, int n)
```
- `int` 型配列の  $i, j$  要素を交換する関数：
- ```
swapdata(int *p, int i, int j)
```

を定義しています。

#### リスト 51 sort.h

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4 #include <time.h>
5
6 static int *initdata(int n)
7 {
8     int i, *p;
9
10    p = malloc(n * sizeof(int));
11    srand(time(NULL));
12    for (i = 0; i < n; i++) {
13        *(p + i) = rand();
14    }
15    return p;
16 }
17
18 static void printdata(int *p, int n)
19 {
20     int i;
21
22     for (i = 0; i < 4; i++) {
23         printf("%d, ", *(p + i));
24     }
25     printf("...\n...");
26     for (i = n - 4; i < n; i++) {
27         printf(", %d", *(p + i));
28     }
29     printf("\n");
30 }
31
32 static void swapdata(int *p, int i, int j)
33 {
34     int temp;
35
36     temp = *(p + i);
37     *(p + i) = *(p + j);
38     *(p + j) = temp;
```

39 }

**問題42** 直接選択ソートを用いて、昇順（小さい数値からの並び）ではなく、降順（大きい数値からの並び）となるように、プログラムしてみなさい。

### 14.1.1 単純選択ソート

最も単純なアルゴリズムですが、効率はあまり良くありません。配列の整列されていない範囲から最小値を抜き出し、配列の前の方（あるいは後ろの方どちらか）にそのデータを移動して整列するというものです。

- (i) 配列の  $n-1$  以下の要素は整列が終わっているものとして、 $n$  から  $N-1$  までの要素の最小値を探します。最小値を持つ要素と  $n$  要素を交換します。
- (ii) 操作の対象となる範囲を  $n+1$  以上として同じことを繰り返します。操作を行う範囲がなくなったら終了です。

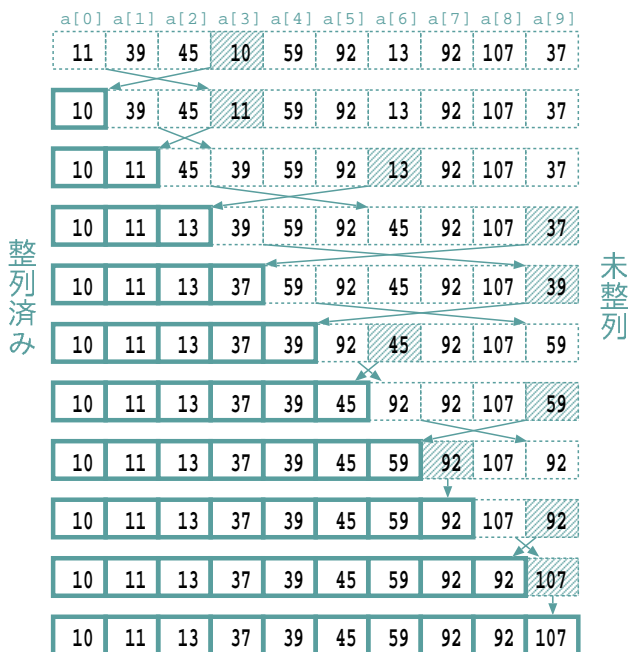


図7 単純選択による整列の過程：未整列領域の最小値（斜線）を持つ要素を領域の一番左端の要素と交換し、整列済み領域を拡大していく。

単純選択ソートでは、比較回数は  $N^2$  に比例し、交換回数は  $N$  に比例します。したがって、計算量のオーダーは  $O(N^2)$  となります。

コンパイルおよび実行画面の例を示します。time コマンドを用いて実行に要する時間（すなわち速度）を計測します。少し多めのデータ数を与えてください。

```
$ gcc -Wall -o sortSelection sortSelection.c
$ time ./sortSelection 10000
619200, 721164, 735131, 942145, ...
..., 2146695888, 2146703754, 2146832300, 2146884147

real    0m0.537s
user    0m0.530s
sys     0m0.000s
```

### 14.1.2 バブルソート

直観的なアルゴリズムで理解しやすいですが、効率はあまり良くありません。バブルソートでも、単純選択ソートと同様に配列の前の方（あるいは後ろの方どちらか）にデータを移動して整列します。

- (i) 配列の添字  $n-1$  以下の要素は整列が終わっているものとし、添字の大きい方から順番に  $j-1, j$  要素を比較し、 $j$  要素の方が大きければ交換します。これを  $j-1=n$  ( $j=n+1$ ) まで実行します。すると小さい数値は交換されて  $n$  要素のところまで移動してきます。
- (ii) 操作の対象となる範囲を  $n+1$  以上として同じことを繰り返します。操作を行う範囲がなくなったら終了です。

単純選択ソートよりも発想が面白く整ったアルゴリズムでソースも僅かに短くて済みます。しかしながら、比較・交換の回数は共に  $N^2$  に比例しますから、計算量は  $O(N^2)$  のオーダーで、しかも単純選択よりも遅くなります。面白いくれど役には立たないという例でしょうか。

💡 バブルソートは隣同士という最小距離間の交換なので、配列全体における順位の情報を取り込むのに手順がかかり過ぎるのです。そこで交換の距離を大きくする、例えば3つ離れた位置との交換を行うようにすると整列が加速されます。このように、改良されたバブルソートはそこそこの速度を持っています。

#### リスト52 sortBubble.c

```
1 #include "sort.h"
2
3 int main(int argc, char **argv)
4 {
5     int N, *data, i, j;
6
7     N = atoi(argv[1]);
8     data = initdata(N);
9
10    for (i = 0; i < N; i++) {
11        for (j = N - 1; j > i; j--) {
12            if (data[j] < data[j - 1]) {
13                swapdata(data, j, j - 1);
14            }
15        }
16    }
17
18    printdata(data, N);
19    free(data);
20
21    return (0);
22 }
```

実行例を示します。確かに直接選択ソートよりも遅くなっています。

```
$ time ./sortBubble 10000 🐼
44343, 247448, 362169, 444820, ...
..., 2146776142, 2146910421, 2147160332, 2147288822

real    0m2.022s
user    0m2.020s
sys     0m0.000s
```

**問題43** バブルソートを用いて、降順（大きい数値からの並び）ではなく、昇順（小さい数値からの並び）となるように、プログラムしてみなさい。

```
$ time ./sortQuick 100000
220314, 511016, 1119653, 1191386, ...
..., 2144871435, 2146221120, 2146889277, 2146893297

real    0m0.017s
user    0m0.020s
sys     0m0.000s

$ time ./sortQuick 1000000
1397, 3293, 6812, 7265, ...
..., 2147472445, 2147477459, 2147479480, 2147482463

real    0m1.606s
user    0m1.510s
sys     0m0.090s
```

### 14.1.3 クイックソート

クイックソートは整列対象の領域を狭めていき効率を高めています。すなわち、まずある値（ピボットと呼ばれます）よりも小さい値と大きい値に配列全体を分けます。すると**ピボットの順位が判り配列全体の中での位置が正しく決定されます**。同じことを分割された領域に適用していき、整列対象の領域の要素数が全て1となったら終了するというものです。ピボットの選び方はいろいろありますが、ここでは領域の先頭の値を用いる方法で説明します。

- (i) 配列のある領域の先頭の位置を left 末尾を right として、left 値と各値を left+1 の位置から順に比較していきます。ここで小さい値を集める末尾の位置を j とします。初め j=left です。位置 i の値が left 値より小さい場合には、j+1 の位置と交換します。そして再び交換されることがないように j++します。i=right になるまで繰り返します。これで位置 left から j の領域は、left 値よりも大きくない値で占められることになります。すなわちこの領域において left 値が最大値となっています。
- (ii) 位置 j の値と left 値を交換します。これで、j 位置を堺として、先頭側には left 値よりも小さな値、末尾側には大きな値が集められました。
- (iii) left 位置から j-1 位置までの領域、j+1 位置から right 位置までの領域について、上記の手続きを繰り返します。
- (iv) 最初 left=0, right=N-1 とし、分割されていった領域の要素数が全て 1 になったら終了します。

図 8 に整列が進行していく過程を示しましたが、それでもやや難解かもしれません。しかし、計算量のオーダーは  $O(N \log N)$  でソートのアルゴリズムとしては最良といわれています。

sortQuick.c では、再帰関数 quicksort() を定義しました。実行例を以下に示します。前の 2 つに比べて劇的に速くなっていることが判ります。こうまで違うと、アルゴリズムの重要性を認めざるを得ないでしょう。

| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9] |
|------|------|------|------|------|------|------|------|------|------|
| 109  | 108  | 41   | 112  | 90   | 0    | 34   | 44   | 1    | 27   |
| 109  | 108  | 41   | 90   | 0    | 34   | 44   | 1    | 27   | 112  |
| 27   | 108  | 41   | 90   | 0    | 34   | 44   | 1    | 109  | 112  |
| 27   | 108  | 41   | 90   | 0    | 34   | 44   | 1    | 109  | 112  |
| 27   | 0    | 1    | 90   | 108  | 34   | 44   | 41   | 109  | 112  |
| 1    | 0    | 27   | 90   | 108  | 34   | 44   | 41   | 109  | 112  |
| 1    | 0    | 27   | 90   | 108  | 34   | 44   | 41   | 109  | 112  |
| 1    | 0    | 27   | 90   | 108  | 34   | 44   | 41   | 109  | 112  |
| 0    | 1    | 27   | 90   | 108  | 34   | 44   | 41   | 109  | 112  |
| 0    | 1    | 27   | 90   | 108  | 34   | 44   | 41   | 109  | 112  |
| 0    | 1    | 27   | 41   | 34   | 44   | 90   | 108  | 109  | 112  |
| 0    | 1    | 27   | 41   | 34   | 44   | 90   | 108  | 109  | 112  |
| 0    | 1    | 27   | 41   | 34   | 44   | 90   | 108  | 109  | 112  |
| 0    | 1    | 27   | 34   | 41   | 44   | 90   | 108  | 109  | 112  |

**図 8 クイックソートの過程：対象領域（青銅色実線部分）の先頭値（斜線）よりも小さい値を先頭側に集めて（網掛け），その末尾と先頭要素を交換する．交換された末尾位置をピボット（ベタ塗）として，先頭側にはピボット値よりも小さい値，末尾側には大きい値が集められている．すなわち，ピボットの配列全体における順位（正しい整列位置）が決定される．ピボットの両側に分割された領域についても同様の手順を行いピボットを次々決定していく．整列を終えた部分は枠で囲っている．**

リスト53 sortQuick.c

```
1 #include "sort.h"
2
3 static int quicksort(int *p, int left, int right)
4 {
5     int i, j;
```



```

6   if (left >= right)
7       return (0);
8
9   j = left;
10  for (i = left + 1; i <= right; i++) {
11      if (*(p + left) > *(p + i)) {
12          j++;
13          swapdata(p, j, i);
14      }
15  }
16  swapdata(p, left, j);
17  quicksort(p, left, j - 1);
18  quicksort(p, j + 1, right);
19
20  return (0);
21 }
22
23
24
25 int main(int argc, char **argv)
26 {
27     int N, *data;
28
29     N = atoi(argv[1]);
30     data = initdata(N);
31
32     quicksort(data, 0, N - 1);
33
34     printdata(data, N);
35     free(data);
36
37     return (0);
38 }

```

⚠ 実は標準ライブラリにクイックソートを用いた整列関数 `qsort()` があります。いろいろな型に対応するために引数がやや判りずらくなっていますが、試しに使ってみましょう。 [sortStdQuick.c](#) を取り寄せてコンパイルしてみてください。実行画させると以下のようになって、`sortQuick.c` より少し遅いようです。

```

$ time ./sortStdQuick 1000000

2794, 3869, 4180, 6095, ...
..., 2147478153, 2147478363, 2147479263, 2147481865

real    0m2.548s
user    0m2.380s
sys      0m0.150s

```

⚠ 標準関数 `qsort()` を使って、構造体の配列の整列を行うプログラムを作成してみましょう。構造体元素の章で、データ “element.dat” を読み込むプログラムを作成しました、そこに追加します。構造体の配列をそのまま `qsort()` に渡すことでも実現できますが、それですと**構造体の一括代入**が起こり速度が遅くなります。そこで、構造体へのポインタ配列を用いることにしましょう。 [sortStdQStruct.c](#)

#### リスト 54 sortStdQStruct.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  typedef struct ads {
6      int number;           /* 原子番号 */
7      char symbol[3];       /* 元素記号 */
8      char name[20];        /* 元素名 */
9      double mt;            /* 融点 */
10     double ec;             /* 電気伝導率 */
11     double tc;             /* 熱伝導率 */
12 } Ads;
13
14 void print(Ads * p)
15 {
16     printf("%2d %s\t%-8s\t%8.3f %8.2g %6.1f\n",

```

```

17     p->number, p->symbol, p->name, p->mt, p->ec,
18     p->tc);
19 }
20
21 Ads getdat(char buf[])
22 {
23     Ads p;
24     sscanf(buf, "%d%s%slf%lf%lf", &p.number,
25         p.symbol, p.name, &p.mt, &p.ec, &p.tc);
26     return (p);
27 }
28
29 static int compmt(Ads ** s1, Ads ** s2)
30 {
31     if ((*s1)->mt == (*s2)->mt) {
32         return (0);
33     } else {
34         return (((*s1)->mt > (*s2)->mt) ? 1 : -1);
35     }
36 }
37
38 int main(void)
39 {
40     int i = 0, imax;
41     char buf[80];
42     Ads element[100];
43     Ads *e[100];
44
45     while (fgets(buf, 80, stdin) != NULL) {
46         element[i] = getdat(buf);
47         e[i] = &element[i];
48         i++;
49     }
50     imax = i;
51
52     qsort(e, imax, sizeof(struct iccd *), compmt);
53
54     for (i = 0; i < imax; i++) {
55         print(e[i]);
56     }
57     printf("\n");
58     for (i = 0; i < imax; i++) {
59         print(&element[i]);
60     }
61
62     return (0);
63 }

```

30 行目からの `compmt` 関数は元素データの構造体 `Ads` の融点を値とするメンバー `mt` を比較します。構造体のポインタ配列へのポインタを引数としている所に注目してください。59~61 行は、元の構造体配列 `element` 自体の並べ替えは行われてないことを確認しています。実行画面を以下のようになります。

```

$ sortStdQStruct < element.dat

80 Hg    Mercury    234.280    1e+06    8.0
19 K     Potassium  336.800    1.6e+07  109.0
11 Na    Sodium     370.960    2.3e+07  125.0
...(中略)...
73 Ta    Tantalum   3269.000   8.3e+06  57.0
74 W     Tungsten    3683.000   2e+07    170.0
                               mt (融点) についての昇順整列
 3 Li    Lithium     453.690   1.2e+07  82.0
 4 Be    Beryllium  1551.000   3.7e+07  220.0
11 Na    Sodium     370.960   2.3e+07  125.0
...(中略)...
82 Pb    Lead        600.652   5.2e+06  35.0
83 Bi    Bismuth     544.450   1e+06    11.0

```

## 付録

### A 文法の補足

#### A.1 演算子

##### A.1.1 優先度について

ANSI C で規定された優先順位の表が参考書 p.177 にありますが、これと若干異なる 15 段階に分類した表を掲載している書籍も多く混乱します。バイブルと呼ばれる

B.W. カーニハン, D.M. リッチー著 石田晴久訳：  
プログラミング言語 C 第 2 版 (共立出版, 1989)

の p.65 にも 15 段階に分類した表が載っています。ただし、この書籍には参照マニュアルが付録しており、p.244 から演算子の優先順位の説明があつて、それをみると 19 段階に分類されていて、ますます混乱です。


しかしながら、どちらの表に従っても結果が変わることはないの心配することはありません。私個人としては、分類の段階の少ない方がすっきりしていてよいと思うので、ここに 15 段階に分類された表 19 を示します。

表 19 演算子の優先度

| 優先度            | 演算子                                  | 結合規則 | 備考                   |
|----------------|--------------------------------------|------|----------------------|
| 1              | () [] -> .                           | 左から右 |                      |
| 2 <sup>†</sup> | ! ~ ++ -- + -<br>(type) * & sizeof   | 右から左 | 否定, 数値の符号<br>メモリアクセス |
| 3              | * / %                                | 左から右 | 乗除, 剰余               |
| 4              | + -                                  | 左から右 | 加減                   |
| 5              | >> <<                                | 左から右 | ビットシフト               |
| 6              | < <= > >=                            | 左から右 |                      |
| 7              | == !=                                | 左から右 | 関係                   |
| 8              | &                                    | 左から右 | ビット AND              |
| 9              | ^                                    | 左から右 | ビット XOR              |
| 10             |                                      | 左から右 | ビット OR               |
| 11             | &&                                   | 左から右 | 論理 AND               |
| 12             |                                      | 左から右 | 論理 OR                |
| 13             | ?:                                   | 右から左 | 条件                   |
| 14             | = += -= *= /= %=<br>&= ^=  = <<= >>= | 右から左 | 代入                   |
| 15             | ,                                    | 左から右 | カンマ                  |

<sup>†</sup> 優先度 2 の演算子は全て単項演算子です。

間違い易い事柄と、それを確かめるプログラムを例示しますのでよく吟味してください。以下の説明において第  $n$  群とは、上記の表に基づく優先度  $n$  である演算子の群を指します。

 曖昧な場合には () で括って優先順位を明示すると間違いないです。

**右結合** 第 2 群の演算子は単項演算子です。それらが右結合であるとは、オペランド  $x$  に対して 2 つの演算子が作用した場合

```
op1 op2 x = op1(op2 x)
op1 x op2 = op1(x op2)
```

と、右側から結びついていくと解釈します。実は、2 番目の構文は  $op2$  が後置の  $++$ ,  $--$  である場合に限ります。したがって後置  $+$ ,  $--$  は自動的に優先順位が他の第 2 群の演算子より高いことになります。

第 1 群の演算子は変数の後ろに記述されますから後置です。そこで、ANSI C では後置の  $++$ ,  $--$  を第 1 群に格上げし第 1' 群としたのです。

**増分(減分)演算子  $++$ ,  $--$**  は前置と後置があつて結構厄介です。

- 右結合である
- 後置は式全体の評価の後にオペランドの値 (式値ではない) を変化させる (代入が行われる)。
- 増分した値が代入可能な、つまり変数のような式にしか作用しない。したがって定数ポインタや式の値に作用することはない。

という規則で解釈すれば理解できます。

まずは、第 3 群の演算子と一緒にの場合です。素直に考えれば納得できるのですが不思議と間違えてしまいます。

#### リスト 55 precedence1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x = 2, y = 5;
6
7     printf("x=%f, y=%f\n", x, y);
8     printf("Value of x*y++ is %f.\n", x * y++);
9     printf("x=%f, y=%f\n", x, y);
10    printf("Value of x*++y is %f.\n", x * ++y);
11    printf("x=%f, y=%f\n", x, y);
12
13    /* 文法が間違っているの、コンパイル時にエラーとなる。
14       printf("Return value of (x*y)++ is %f.\n", (x*y)++);
15       printf("x=%f, y=%f\n", x, y);
16       */
17
18    return 0;
19 }
```

```
$ gcc -pedantic -ansi -Wall -o precedence1
precedence1.c
$ ./precedence1
x=2.000000, y=5.000000
Value of x*y++ is 10.000000.
x=2.000000, y=6.000000
Value of x*++y is 14.000000.
x=2.000000, y=7.000000
```

`++` は `*` よりも優先度が高いので `y++` や `++y` と結合されます。 `(x*y)++` とはなりません。ただし後置の場合には `y` の値が `+1` されるのは式全体の評価が終わった後です。

ポインタと一緒になった場合は大変間違い易いので注意しましょう。

#### リスト56 precedence2.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x = 1, *px;
6
7     px = &x;
8     printf("px=%p, *px=%f, x=%f\n", px, *px, x);
9     printf("Return value of *px++ is %f.\n", *px++);
10    printf("px=%p, *px=%f, x=%f\n", px, *px, x);
11
12    return 0;
13 }
```

```
$ ./precedence2
px=0xbffff974, *px=1.000000, x=1.000000
Return value of *px++ is 1.000000.
px=0xbffff97c, *px=2.399139, x=1.000000
```

ポインタに対する間接演算子 `*` と `++` はともに第2群に属していますから優先度は等しいです。解釈は結合規則に従いますから、右側にある `++` が優先されます。すなわちポインタ変数の値の増分 `px++` が実行されることになります。その結果 `px` の参照先が `x` ではなく、`&x+8` となります。もちろん間接参照値 `*px` は不定です。また、後置 `++` が実行されるのは式全体の評価が終了してからですから、9行の時点では、ポインタ変数 `px` は未だ `x` を参照しています。このような使い方は、参考書の11-3節 (p.260~) に示されているように、文字列 (あるいは配列) を扱う関数のソースでよくみかけます。

**キャスト演算子 (type)** は第2群に属しています。他の第2群の演算子より優先度が低いとしなくてもよいでしょう。

#### リスト57 precedence3.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x = 1.0 / 3, y;
6     y = x;
7     printf("x=%f, y=%f\n", x, y);
8     printf("++(int)x=%d, (int)++y=%d\n",
9           ++(int) x, (int) ++y );
10    printf("x=%f, y=%f\n", x, y);
11
12    return 0;
13 }
```

```
$ gcc -o precedence precedence.c
$ ./precedence
x=0.333333, y=0.333333
++(int)x=1, (int)++y=1
x=1.000000, y=1.333333
```

`++(int)x` は `x` がまず整数値 `0` と変換され、後に `+1` されます。一方 `(int)++y` はまず `y` が実数のまま `+1` されています。

ANSI の規定ではキャスト演算子は右辺値 (代入ができない) を生成するので (また C99 でも左辺値を生成する実装を認めていないので), `++`, `--` 演算子を作用させることはできません。この例がうまく動作するのは gcc の拡張機能によるものです。

**--- はどうなる?** 本屋で立ち読みした C 言語の某入門書に、

```
----x=-( -(-x) ) ) )
```

という記述をみつけました。このように親切に解釈してくれると嬉しい? ののですが、そうは問屋がよろしません。 `-` が3個以上の場合、優先順位の高い演算子を (多分左側から) 抽出しますから、

```
---x => -- -x, ----x => -- --x
```

と解釈し、`-x` や `--x` には代入ができませんので、文法エラーとなってしまいます。では、次の式はどうなるでしょうか?

```
x---y
```

`precedence4.c` をコンパイルして実行する前に、実行結果を予想してください。

#### リスト58 precedence4.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int x = 10, y = 6;
6
7     printf("x = %d, \ty = %d, \t", x, y);
8     printf("x---y = %d\n", x-- - y);
9     printf("x = %d, \ty = %d, \t", x, y);
10    printf("x-(-y) = %d\n", x - (-y));
11    printf("x = %d, \ty = %d, \t", x, y);
12    printf("x++-y = %d\n", x++ - y);
13
14    return 0;
15 }
```

## B プログラムの図式化

プログラムの処理の流れを図面に表して整理することは、プログラムを改良する上で大変役に立つといわれています。もっとも、大規模なプログラムではその図面自身も大きく複雑になってしまい解読が大変ですし、逆に A4 の紙1枚に収まる程度の小さなプログラムならば、わざわざ図面にしなくとも頭の中で考えれば済んでしまいます。という訳で、なかなかそのような図面を書いてからプログラ

ムに取り掛かるというような癖はつきにくいものです。プログラムを書きはじめる前に使うのではなく、ある程度作成が進んだ段階で構造を眺めて反省する位の使い方も得るものがあると思いますので、一度位は描いてみるのもよいでしょう。

このような図式化には、PF（プログラムフローチャート）、N-S 図、PAD があります。これらのうち PF は歴史が古く JIS 規格があり国内標準と呼べるものです。また PAD は ISO 規格になっています。

表 20 構造化プログラミングの三要素、接続・選択・反復の PF、PAD における記法

| 文  | PF | PAD |
|----|----|-----|
| 接続 |    |     |
| 選択 |    |     |
| 反復 |    |     |

表 20 に構造化プログラミングの三要素、接続・選択・反復だけをとり上げて記法を示しました。反復については、継続条件の判定が前判定 (while) と後判定 (do...while) の 2 通りを分けて記述します。この他に多岐選択や無条件分岐 (PAD の場合) の記法も定められていますが、詳細は省きます。

## B.1 フローチャート

フローチャートは、簡単にいえば手続きを記号（処理）と結線（順序）を用いて図示するための記述法です。実は JIS の規格では数十の記号が用意されて、正確な JIS 規格のフローチャートを書くことは難しいのですが、処理を全て四角形の記号で書くことにすれば、簡単に作成できてしまいます。

誰にでもすぐ書けるという特徴はプログラミング言語では BASIC (Beginner's All Purpose Symbolic Instruction code) になぞらえることができます。結線を goto 文に対比させると BASIC が持つ単純な制御構造の記述に見事に合致しているといえます。構造化プログラミングの立場からは BASIC は批判されましたが、初心者には敷居が低いので馴染やすいという点は評価されるべきで、フローチャートもアルゴリズムを図示して整理するというプログラム開発作業の入門には適しています。

## B.2 PAD

さて、フローチャートは、そのあまりの汎用性のゆえ構造化プログラミングの記述には不向きであるという批判があります。そこで構造を記述するのに適した PAD (Problem Analysis Diagram) を簡単に紹介します。PAD には pad2ps という C、C++、Java、Bourne Shell、C shell、awk などのソースを PostScript や TeX の図面に変換してくれるツールが入手できるという利点もあります。

PAD は、二村良彦氏（早稲田大学 理工学部情報学科）により構造化プログラミングを図式化するために考案されました。構造化プログラミングとは、二村氏の早稲田大学情報学科における講義「アルゴリズムとデータ構造 1」によると、以下のように説明されています。

- 接続、選択および反復の 3 つの基本構造のみを用いてプログラムを作成する。
  - オランダの E.W. [ダイクストラ教授](#)により 1968 年頃提唱された。
- ... (以下略)

また PAD はプログラムの流れ（アルゴリズム）以外にもデータ構造をも記述できるので、**アルゴリズムとデータ構造**というプログラミングにとって最も重要な概念の 1 つを図式化でき、プログラム開発の効率化にとって極めて有用なツールであると考えられます。

### B.2.1 pad2ps

PAD 本来の目的はプログラム設計にあるのですが、逆にでき上がったソースファイルの構造を PAD に変換するツールが pad2ps です。同時に install される pad2tex は L<sup>A</sup>T<sub>E</sub>X のソースを作成します (pad.sty が必要)。図 9 に出来上がりの例を示します。この例では、構造化プログラミングの目の敵 goto が使ってあって、その出入り口が丸印で記されています。

### 参考ウェブサイト：PAD

- [C プログラミング入門 付録 D. PAD](#)
- [プログラムの PAD 自動描画ソフト pad2ps](#)



```
int main(void)
```

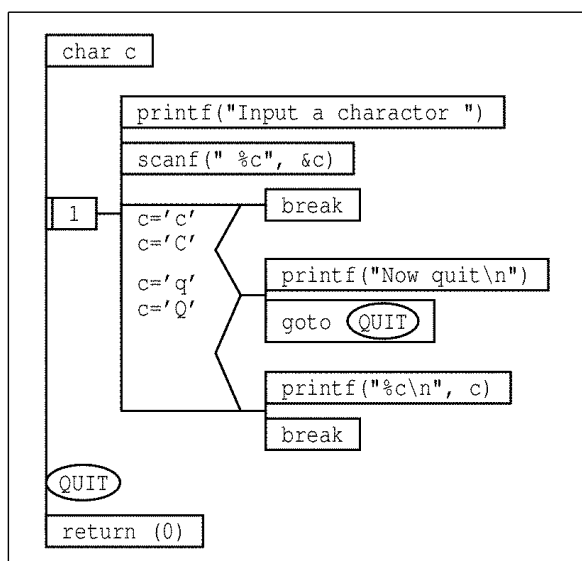


図 9 pad2eps を用いて作成した本文 ex21.c の PAD

## C LINUX および UNIX

Linux は 1991 年ヘルシンキ大学の学生時代に Linus B.Torvalds 氏が一から開発した UNIX 互換 OS です。UNIX の標準規格である POSIX 仕様をほぼ満足していて、移植性も高く intel のチップ以外にも、68k 系や Alpha さらには SPARC にも移植されています。最も大きな特徴は**ソースが公開されていて誰でも自由に改変することができる**という点です。このようなソフトウェアは**オープンソース**と呼ばれ、ソフトウェア開発の大きな流れとなりつつあります。もちろん、その上で動くアプリケーションも**ソース公開**でありながら十分な機能を持つものが多く、コンピュータを学習する人にとっては宝の山とみることができましょう。また、最近では、小規模サーバーや**組み込みの分野**でも市場を形成しつつあり、実用的な OS として認知されつつあります。

この章で述べることは、特に断らない限り Linux 固有のものではなく PC-UNIX (FreeBSD など) や UNIX (Solaris, AIX, HP-UX, SGI) 全般に対しても成り立ちます。なぜなら、UNIX はソース互換性が高く Linux で動くものは他の UNIX でもほぼ動作するからです。Linux を使うことによって、これからのネットワーク社会を担う安定で堅牢な OS として期待の高い UNIX について十分な経験を積むことが可能です。

### ☞ 参考ウェブサイト：Linux とオープンソース関連

- [日本 Linux 協議会](#)
- [Linux Journal](#)
- [GNU の母体 Free Software Foundation](#)
- [Ring Sever Project](#)

## C.1 Login/Logout

### C.1.1 login

電源の ON/OFF は端末室のスタッフに任せ、電源スイッチは勝手に押さないでください。授業開始時には既に Linux が起動して X Window System 上でログインダイアログが現れていますから、ユーザー名 (user name) と UNIX 用のパスワードを入力してください。

正規のユーザーと認められるとログインできて、ユーザーの初期画面が現れます。初期にどのような X クライアント (client:X 上のアプリケーションをクライアントと呼びます) を起動するかは GUI から入った場合には `$HOME/.xsession` で設定します。(`$HOME` については後で説明します。)

### C.1.2 logout

自分の作業 (session) を終わったら、必ずログアウトしてください。現在のデスクトップ環境 (**GNOME** といいます) では、MS-Windows と同様にツールバーの左端の“足”のアイコンをクリックして、メインメニューでログアウトを選択します。

### C.1.3 シェル

ユーザーはシェルと呼ばれる基本的なアプリケーションを通じて、OS に作業を命じます。シェルには、sh (最も初期から存在してシェル)、csh (BSD で開発された C 言語に似た文法を持つシェル) という 2 大潮流があり、どちらを使うことも可能です。また、両者の長所を採り入れた ksh や zsh などといったシェルも開発されています。

しかし、Linux では一般に sh の開発者の名前 **Stephen Bourne** を born (生まれ変わりの意) と洒落た bash (**b**ourne **a**gain **s**hell) が既定のシェルです。本文では、特に断らない限り bash を使っている場合の実行例であることに注意してください。

## C.2 ディレクトリとファイル

### C.2.1 木構造とリンク

現在、どのような OS でも外部記憶装置にディスクを用いることが常識です。プログラムのソースやテキストや実行バイナリは **ファイル (file)** という単位で管理されます。また、関連性のあるファイルをまとめて置く場所を **ディレクトリ (directory)** あるいは **フォルダー (folder)** と呼ぶこともご存知でしょう。ディレクトリの下にはディレクトリを作ることが可能で、全体は階層的な構造をなします。その形状を指して**木構造**と呼びます。UNIX では木構造の

トップ、すなわち枝分れの元はルート（根：root）と呼ばれ“/”で示されます。tree コマンドで木構造を表示させてみましょう。ディレクトリのみを表示するオプション -d と全ての名前（ドット“.”で始まる名前など）を表示するオプション -a 指定して実行します。

```
$ tree -ad /
/
|-- bin
|-- boot
|   '-- grub
|-- dev
|   |-- fd -> ../proc/self/fd
|   |-- inet
|   |-- input
|   |-- lost+found
|   |-- pts
|   '-- shm
|-- etc
|   |-- .diskless [error opening dir]
|   |-- CORBA
|   |   '-- servers
|   |-- FreeWnn
|   |   |-- ko_KR
|   |   |   '-- rk
.....
```

一目瞭然ですが、記号“->”については説明が必要でしょう。これは**リンク (link)** と呼ばれる機構で、ディレクトリやファイルを別名で参照できるようにします。例えば、一つのデータ（ファイルでもディレクトリでもよい）をいろいろなプロジェクトで共通に使いたい場合には、プロジェクト毎のディレクトリに逐一コピーしていたのでは大変です。その場合には、実体からリンクを張った仮想的なファイルやディレクトリを作成します。名前は自由ですから、プロジェクトに即した論理的なものを付けることも可能です。これで、仮想ファイルへの読み書きが、実際には実体への読み書きとなります。

ln [-s] リンク元（実体） リンク先（仮想）

このように、UNIX では木構造の枝と枝を跨いだつながりも自由に形成できます。リンクを多用すると、場合によっては蜘蛛の巣のようになってしまい全体の構造が分り難くなりますから要注意です。

④ UNIX の特徴の一つに、ファイル化の徹底があります。キーボードやプリンター、シリアルなどの周辺デバイスはテキストファイルなどとは全く実態が異なっていますが、UNIX ではこれらを仮想化してファイルとして扱えるようにしています。/dev 以下にデバイスファイルがまとめられていますから見てみるといいでしょう。例えば、プリンタとの通信はデバイスファイル /dev/lp\* への読み書きで行うことが可能です。

## C.2.2 カレント、ホーム

現在中にいて作業しているディレクトリは**カレント (current)** と呼ばれ“./”で表されます。また、ユーザーはログインすると一般にはホームディレクトリ

/home\*/ユーザー名

がカレントになるよう設定されています。次のコマンドで自分のホーム（シェル上で HOME という変数に記憶されています）とカレントを確かめてください。pwd (print working directory) は専用のツールです。

```
$ echo $HOME
/home/naio730
$ pwd
/home/naio730
```

なおホームを“~/”で表すこともできます。

## C.2.3 絶対パス、相対パス

ディレクトリやファイルの位置（パス：path）を指定するには2通りの方法があります。ルートを原点にして

```
/home/matuda/.profile,
/usr/local/share/texmf/tex/latex209
```

のように指定する絶対パス指定と、カレントを中心にして

```
./file,    ../../directory/file
```

と指定する相対パス指定です。“./”は親 (parent) ディレクトリ、すなわち階層が一つ上のディレクトリを示します。絶対パスは曖昧さのない指定ですが階層が深いと入力が面倒です。階層の深いところでは、相対パス指定が便利です。が、カレントを把握していないと間違ってしまう。旨く使い分けましょう。

## C.3 UNIX の基本コマンド

現在流行のウィンドウシステムは操作が直観的な優れたヒューマンインターフェースといえましょう。しかしながら、マウスを使ってボタンクリックすることのできることに限りがあります。その端的な例は、文章の作成です。文字列の入力は今だにキーボードを使っています。また、コンピュータは人間の作成したプログラムに従って忠実に作業を実行していく機械です。このプログラムはボタンの組合せでは表現できないでしょう。人間の思考はやはり言葉、すなわち文字列によって成り立っているのです。歴史的に言うと、UNIX は数値計算しか利用されていなかったコンピュータをもっと汎用の、例えば文章処理システムとして利用するための OS として考えられました。文字

列やその塊であるファイルの操作は UNIX の中核をなすものです。操作 (=思考) を細分化していった結果、UNIX では実に多くの単純なコマンドが生みだされました。それは “ls /usr/bin /bin” を実行すると理解できます。数百ものコマンドの存在に圧倒されるかもしれません。中には一生使わないものもあるでしょう。以下では、その中でも必須のコマンドを紹介します。このコマンドを理解することが、コンピュータを使って仕事を行う（自分の思考をコンピュータに肩代りさせるといことです）過程を理解することに通じると信じて、十分に習熟してください。

### C.3.1 コマンドの書式

コマンド (command) は、引数 (argument) としてオプション (option) やファイル名 (filename) を従える場合がありますから、[...] を付けて（あってもなくても良い意味）それらを表します。従って、一般的な書式は

```
command [-options] [filename]
```

となります。このようにコマンドど引数による命令文を**コマンドライン**と呼びます。なお、特別（しかしながら頻繁に顔を出します）なファイル名 “-” は、標準入力（後述）を示します。

### C.3.2 ディレクトリに関するコマンド

**ディレクトリ内容の一覧表示：ls** ディレクトリにどのようなファイルがあるのか表示させるコマンドです。いろいろなオプションがありますが、まず -F により、通常のファイルとディレクトリが区別されます。次の例では

```
$ ls -F
Mail/  Wnn7/  Work/  tree
....
```

ディレクトリの名前の後ろにはスラッシュ “/” がついていて通常のファイルとは区別されています。またオプション -l により詳細な情報が表示されます。

```
$ ls -l LJP
合計 1128
drwx----- 5 namio730 extrat    4096  3月 28 22:53 Mail
drwx-----t 2 namio730 extrat    4096  3月 28 22:55 Wnn7
drwxr-xr-x  2 namio730 extrat    4096  4月  4 22:22 Work
-rw-r--r--  1 namio730 extrat   131072 4月 12 13:15 tree
...
```

最初の欄はファイルの**許可属性** (permission) と呼ばれ、ファイルの秘密保護のために重要な役割を果たします。2 番目はリンク数、3、4 番目はファイルの所有者と所属グループ名、そして、ファイルの大きさ、最終更新日時、ファイル名となっています。

**ファイルの許可属性 (file permission)** 最初の一文字を除いて話をしますと、基本的には rwx が 3 回繰り返され、時折 “-” に変化してます。r(read) は読み込み可能、w (write) は書き込み可能、x (execute) は実行可能の属性を示します。“-” はその属性の否定を示します。3 回の繰り返しは、ユーザー本人 (user)、グループ (group)、他人 (other) に対応しています。したがって、

```
-rw-r--r-- : 本人は読み書き、グループ所属のユーザー
             と他人（誰でも）が読み込み可能
-rw----- : 本人は読み書き可能、他は読み書き不可能
```

の意味になります。後者の場合には他人は全く手だてできない秘密文書となります。UNIX では同時に複数のユーザーがログインすることを想定していますから、このような保護機構が考えられています。

ドットで始まるファイルは通常 ls では表示されません。全て (all) 一覧する “-a” オプションを付けると表示されます。

```
$ ls -aF
./      .bash_history .bashrc .history .mh_profile Mail/  Work/
../     .bash_profile .emacs  .kkcrc  .ssh/    Wnn7/  tree
....
```

この例では今いる (current) ディレクトリ “./” と一つ上のディレクトリ “../” が表示されました。

ファイルの所在や属性を把握することは UNIX のファイルシステム全体を理解する上で極めて重要な事柄ですから、頻繁に “ls” を実行してみましょう。

**カレントディレクトリの表示：pwd** プロジェクトに応じてサブディレクトリを何層も作って作業を行っていると、現在どのディレクトリに居るのか分からなくなる時があります。そんな場合 pwd (print working directory) でカレントディレクトリを確かめます。

**ディレクトリの作成と削除：mkdir, rmdir** ディレクトリを作成するには mkdir、逆に削除するには rmdir を用います。なお、rmdir でディレクトリを削除するには、ディレクトリが空である必要があります。したがって、rmdirではなくファイルの削除コマンド rm を用いて

```
rm -rf ディレクトリ (-r:再帰的, -f:強制的)
```

としてディレクトリ以下全てのファイルを消去する方が近道かもしれません。また、深いディレクトリを（途中のディレクトリがない場合）いっぺんに作成するには、オプション -p を指定します。次の実行結果を比べてください。

```
$ mkdir sdir/ssdir
mkdir: cannot make directory 'sdir/ssdir': No such file ...
$ mkdir -p sdir/ssdir
$
```

初めのコマンドでは `sdir/` がないので `sdir/ssdir` は作成できないと拒否されました。2 番目のコマンドはすんなり終わっています。“`mkdir -p`”と同じ働きをするコマンド `mkdirhier` (make a directory hierarchy) も良く使われるコマンドです。

**ディレクトリ間の移動: `cd`** 関連性のあるファイルを一つのディレクトリに集約するというのは自然な発想です。すると、かなりの数のディレクトリができあがって、ある作業をするために頻繁にディレクトリ間を移動することになります。`cd` は、絶対パスあるいは相対パスで指定されたディレクトリに (カレントを) 移動するためのコマンドです。

🔗 **ディレクトリスタックの操作: `pushd`, `popd`** 同じディレクトリ間を何度も行き来する場合には、`cd` でいちいち行き先を指定しなくても、`pushd` と `popd` を使って簡単に済ませることができます。これは単独のコマンドではなく、シェルの機能です。仕掛を簡単に説明します。シェルはディレクトリスタックという、ディレクトリのパスを保存する記憶場所を持っています。中身が現在どうなっているかは `dirs` で表示できます。`pushd` `newdir` とすることで、`newdir` をスタックの先頭に積みみます。`pushd +n` はスタックを循環させて左から `n+1` 番目の要素を先頭に持ってきます。単に `pushd` とすると最初と次の要素を入れ替えます。逆に `pod +n` は、スタックの左から `n+1` 番目の要素を取り除きます。引数が無い場合は先頭の要素を除きます。こうして、スタックの要素を入れ換えて先頭の要素 (ディレクトリ) が決まりますが、そこにカレントがあるという仕組みです。論より実習、次の実行例で納得してください。

```
eris061:~$ dirs
~  ← スタックにはカレントしかありません
eris061:~$ pushd /tmp
/tmp ~  ← 先頭に /tmp を積みました
eris061:tmp$ pushd /usr/local
/usr/local /tmp ~  ← 更に /usr/local を積みました
eris061:/usr/local$ pushd +2
~ /usr/local /tmp  ← 左から (2+1) 番目を先頭 (0 番目) に持ってきました
eris061:~$ popd
/usr/local /tmp  ← 先頭を取り除きました
eris061:/usr/local$
```

### C.3.3 ファイルに関するコマンド

**ファイルの内容の表示: `less`, `cat`** ファイルの内容を表示させるコマンドです。`cat` (concatenate) は古くからあるコマンドで、ファイルの最後まで一期に表示して (画面が流れて) しまいますから長いファイルでは最後しか見ることができません。それに対して `less` は逆スクロールが可能ですからじっくりと眺めることができます。`less` は表示途中で対話的な文字検索なども可能となっており非常に多機能です。

`less` を終了するコマンドは `q` です (他にもあります)。`[ESC]` や `[CTRL] + [C]` ではありません。ヘルプが `h` で立ち上がりますから一度眺めてみてください (図 10)。

図 10 less の help 画面

🔗 **`tac` と `rev`** `cat` はファイルの先頭から順に表示するコマンドですが、`tac` はファイルの末尾から表示するコマンドです。`rev` は、ファイルの先頭から表示しますが、行内の文字の並びを逆に表示します。

**ファイルの先頭や末尾の表示: `head`, `tail`** ファイルの先頭の部分を表示するコマンドが `head` で、末尾を表示するコマンドが `tail` です。何も指定しないと 10 行表示します。オ



ブション“-n 数字”で表示する行数を指定できます。単に“-数字”でも同じです。沢山のファイルの先頭の方だけを読みたい（例えばレポートだったら名前の確認をする場合など）場合などに役立ちます。

**ファイルの複写：cp** ファイルを複写するには cp を使います。複写先にファイル名を指定すると、その名前で複写されます。複写先がディレクトリならば、ディレクトリの下に複写元と同じ名前のファイルができます。例えば、ファイル名を copied, ディレクトリ sdir とすると

```
cp copied sdir (あるいは sdir/)
```

により sdir/copied が作成されます。また

```
cp copied sdir/copiedcopy
```

とすれば、sdir/copiedcopy という名前で内容が copied と同じファイルができます。複写先がディレクトリならば複写元に複数のファイルを指定することも可能です。例えば

- cp copied1 copied2 copied3 sdir/
- cp {copied1,copied2,copied3} sdir/
- cp copied\* sdir/

いずれの方法でも sdir の下に 3 つのファイルが複写されます（第 3 の方法では他のファイルも複写されてしまう危険性があります）。

**ファイルの移動：mv** ファイルの移動コマンド mv は、cp コマンドと似ていますが、移動元が消去されてしまう点が大きく異なります。同一ディレクトリで名前を変えて移動する、すなわち、ファイルの名前変更（MS-DOS では rename）コマンドとして使うことができます。したがって

- cp movesrc movedst; rm movesrc
- mv movesrc movedst

は同じ結果となります。

**ファイルの消去：rm** ファイルを消去するコマンドです。MS-DOS では delete でファイルを消去してもエントリ（名前）が無くなるだけで、内容が残っている場合もありました。しかし、UNIX 系では rm 実行後に内容を救い出すことはほぼ絶望的です。従って、**rm を実行するときには慎重に行ってください。**とくに慣れてきて、ワイルドカード“\*”を用いて一辺に沢山のファイルを消す時などは危険です。例えば、名前の最後に ~ が付いているファイル（mule が作成するので溜ってきます）を消去するには“rm ~”ですが間違って“rm \* ~”と空白を入れてしまったら大変です。違いは“ls ~”と“ls \* ~”の結果を比較すれば分ります。事故を未然に防ぐには


```
alias rm='rm -i'
```

などと**別名定義**をし、必ず実行の是非を問うようにしておくとい良いでしょう。

**ファイルの検索：find** ファイルの検索には find を使います。基本的な書式は

```
find ディレクトリ -name 'ファイル'
```

となります。ファイル名にはシェルのパターンマッチングを用いることができます。検索の条件も細かく設定でき、また、検索して見つかった場合のアクションを定めることもできるので、大変汎用性の高いコマンドです。が、その分オプションの数が膨大にあって、全容を掴むのが難しいコマンドの一つです。

 **locate** GNU にはファイル名のデータベースから文字列を高速検索するコマンド locate があります。find よりも機能も書式も単純で locate strings のように検索する文字列を指定するだけです。データベースを更新するコマンドは updatedb です。updatedb の実行は多少時間が掛かりますから、一般には cron によって早朝自動的に更新されるように設定されていることと思います。また、データベースファイルの名前は一般には locatedb の筈ですから、どこにあるのか locate で探してみてください。

### C.3.4 テキスト整形

元来テキスト処理のために UNIX は開発されたという経緯があるので、テキスト（文字列の塊）の処理は得意です。知っていると必ず役に立つ時が来ると思われるいくつかのコマンドを紹介します。

**fold** テキストの 1 行あたりの文字数を揃えます。2 バイト文字である日本語を扱う場合には 1 バイト文字である英語よりもかなり処理が複雑となります。

**expand** 行の最初に空白を入れて字下げ（インデント：indent）をすると、文章がみやすくなる場合があります。プログラミング言語のソースファイルなどはその典型です。空白（普通 4 や 8）に替えてタブストップを使うことも可能です。しかし、タブ文字を空白と解釈しない、あるいは空白 4 文字と解釈して欲しいのに空白 8 文字と扱われてしまうなどの理由によって、レイアウトが全く狂ってしまう場合もあります。タブを正しく空白文字に置き換えてしまえば、このような食い違いは生じません。expand はその目的で使います。

テストファイルを awk で作成して、確かめてみましょう。タブ文字は C 言語や awk では“\t”で表されます。

```
$ awk 'BEGIN{print"\t\t\t\t\t\t"}' > tabstop.txt
$ cat tabstop.txt
1      2      3      4      5      6
$ expand -4 tabstop.txt
1      2      3      4      5      6
$ expand -12 tabstop.txt
1      2      3      4      5      6
```

**column** 入力データをカラムを揃えて表示するコマンドです。例えば次のような column.dat という名前のデータがあったとき

```
1.234,2e-6,45678,q
1.0,3.345e-6,67,a
0.01,3.345e-10,-67,q
```

これを表形式にして表示するには、表 (table) オプション `-t` と区切り文字指定オプション `-s` を使った実行結果は

```
$ column -t s, column.dat
1.234 2e-6      45678 q
1.0    3.345e-6  67    a
0.01   3.345e-10 -67   q
```

となります。

**paste** 複数のファイルを行単位でつなぐコマンドです。同じ構造のデータファイルを1つにまとめるといった用途に便利です。

以上のように UNIX にはテキスト整形のためのコマンドが数多くあります。これら、いわば単機能のコマンドを、次節で述べるようなパイプやリダイレクションを使って組み合わせ、目的とする処理を行うことは UNIX ならではの醍醐味です。もっと多機能な文書整形コマンドもあって、`roff (man)` や `prn` などがその例です。無論あくまでも、ダイアログ (対話処理) などを起動せずに、

```
command textfile
```

で流れ作業的に処理できるところに意味があります。なぜなら、ダイアログがあると人間はそれにつき合わなければならないのですから。

## C.4 リダイレクトとパイプ: > >> |

大抵のコマンドは、実行時にデータやパラメータの入力やメッセージの出力を標準入力、標準出力・標準エラーを通じて行うように設計されています。接続先は、入力がキーボード、出力が CRT となっていますが (制御端末と呼ばれます)、これらを他のファイル切り換える機能をシェルが提供してくれます。例えば、ディレクトリの一覧の結果、すなわち標準出力の接続先を CRT ではなく、テキストファイル `dirs.txt` に切り替えるには

```
ls > dirs.txt
```

とします。また、標準入力を切り換えるには

```
ls < dirs.txt
```

の様にします。“< infile” “> outfile” を同時に指定することもできます。すなわち、

```
ls < dirs.txt > dupdirs.txt あるいは ls >
dupdirs.txt < dirs.txt
```

として、`dirs.txt` から読み込んで `dupdirs.txt` に書き込みます。**I/O リダイレクション**と呼ばれるこの操作は大変便利です。このおかげで、プログラムを作成する場合には出力先別に考える必要がなく、標準出力 (stdout) を出力にすれば済みます。ところで実行中にエラーが生じて停止した場合は、一般にエラー情報が出力されます。この情報はリダイレクトされると困ります、CRT に表示されないからです。そこで、標準エラー (stderr) の登場です。エラー情報を標準エラーに出すように設計すればいいのです。まあ、しかしエラー出力も記録したいなどという場合もあります。その目的には、“&” または “>&” が使われます。標準エラーだけリダイレクトしたい場合には番号を使って、“2> ” とします。別々に記録を取りたいなら

```
command 1> stdlog 2> errorlog
```

とします。簡単なプログラムを作成して実験してみましょう。

```
#include <stdio.h>
int main()
{
    fprintf(stdout,"stdout\n");
    fprintf(stderr,"stderr\n");
    return (0);
}
```

という内容の C 言語のソースファイル `iored.c` を作成してください。続いて

```
gcc -o iored iored.c
```

とコンパイルして実行ファイル `iored` が出来上がります。このプログラムは、標準出力に stdout、標準エラーに stderr という文字列をそれぞれ出力します。次のようにいろいろ試して遊んでみましょう。カレントにあるファイルを実行する場合には、明示的に `./iored` と指定すると確実です。**実行ファイルの所在を検索するディレクトリパスにカレントを含めない設定が UNIX では安全とされているからです。**

```
$ ./iored
stdout
stderr
$ ./iored 2>errlog 1>errlog  ← 画面には何も出力がない
$ head stdlog errlog  ← ファイル stdout,stderr の内容を表示
==> stdlog <==
stdout
==> errlog <==
stderr
$ ./iored 2>errlog &>stdlog
$ head stdlog stderr  ← ファイル stdout,stderr の内容を表示
```

入出力の切替に関するシェルのもう一つの重要な機能はパイプと呼ばれるものです。あるコマンドの標準出力を他のコマンドの標準入力に接続します。例えば `ls` の結果を整理してファイルに番号を付けるには、

```
$ ls > dirlist; cat -n dirlist; rm dirlist
```

のように、一端テンポラリファイル `dirlist` を作成する方法が考えられます。ところがパイプを用いれば

```
$ ls | cat -n
1  7x14maru.bdf
2  7x14maru.pcf.Z
3  7x14rkmaru.bdf
4  7x14rkmr.pcf.Z
5  Book/
6  Calendar/
....
```

で済ますことができます。このように単機能のコマンドの入出力(=文字列の流れ)をパイプで接続して処理を達成する方法は UNIX の中心的な考え方の一つです。

**tee** リダイレクトで出力先を変更しても、数としては一つしかないので実行結果を画面に表示しながらファイルにも記録したいなどという芸当はできません。tee は名前の通り、標準入力から読み込んだ内容を、標準出力とファイルに分岐して(T字を想像しましょう)書き出すコマンドです。例えば、telnet でリモートログインした先のホストで作業した結果などを記録に残すには、

```
$ telnet neko | tee neko.log
```

などのように tee にパイプで渡しますと、neko.log に記録が残ります。

## C.5 知っていると便利なコマンド

### C.5.1 indent

プログラミング言語のソースは適当に字下げして、構造(とくに入れ子構造)を明確にしないと読みにくいです。indent はその名の通り、入れ子構造を解釈して自動的に字下げをしてソースファイルを整形してくれます。ここでは、

そういういかにも開発環境の豊富な UNIX らしいコマンドがあることだけ覚えておいてください。

### C.5.2 grep

ファイルの中からある文字列に一致した行を表示するコマンドです。例えば、

```
$ grep PI /usr/include/math.h
#ifndef M_PI
#define M_PI      3.14159265358979323846      /* pi */
(中略)
#ifndef PI
#define PI M_PI
#endif
#define PI2 (M_PI + M_PI)
```

のように、`/usr/include` (C 言語のヘッダーファイルが置かれています)にある `math.h` の中を検索して `PI` という文字列が含まれている行を表示します。多数のオプションがありますが、当該行だけでなく前後それぞれ `n` 行も加えて表示するオプション `-n` は覚えておくといいでしょう。

### C.5.3 diff

2つのファイルを比較してその違いを表示するコマンドです。詳しくは述べませんが、プログラムソースの差分ファイル `*.diff` (あるいはパッチファイル `*.patch`) を作成する場合の必需品です。また2つのディレクトリを比較することもできますから、バックアップの確認にも重宝します。

🔍 diffに関連して、`sdiff`、`diff3`、`cmp`、`patch` コマンドを `man` で調べてみましょう。

### C.5.4 date

システム時計に日付や時刻の表示設定を行うコマンドです。設定すなわちシステム時計の変更は重大な事柄なので、`root` 以外は実行できません。引数なしならば現在の時間を表示します。

```
$ date
Tue Mar 16 10:29:33 JST 1999
```

時間の表示以外にも、時間の計算もこなしてくれます。例えば、あと34日後の日付、27日前の日付は

```
$ date --date '31 days'
Fri Apr 16 11:07:49 JST 1999
$ date --date '27 days ago' +%D
02/17/99
```

2番目の例では、オプション `+` によって表示の書式を設定してみやすくしました。

### C.5.5 sort

C 言語のアルゴリズムの学習では必ず顔を出す、整列 (sort) コマンドです。デフォルトは文字 (すなわち ascii コード) で比較しますが、`-n` を指定すれば数値順、`-r` で昇降を逆にします。また、行内の (空白文字で区切られた) レコードを指定することもできます。

## C.6 ファイルの保管

教室と自宅のコンピュータ間のファイル輸送は、電話回線などを利用して自宅から FTP するというのがこれからの情報社会の姿と思いますが、初期立ち上げに (モデムなどの購入費用を含めて) ちょっと手間がかかります。現在ほとんどのコンピュータで使用可能な保存媒体はフロッピーディスク (FD) です。FD は容量が 1.44MB (Linux では 1.7MB のフォーマットも可能ですが不安定かも) とあまり大きくないので、圧縮をかけて保存する、複数に分割する等の工夫が必要です。そのためのツールを紹介します。

### C.6.1 tar

GNU tar は、複数のファイルを一つにまとめるアーカイバと呼ばれるツールの代表格です。性能の高い圧縮機能も付いていて、現在最も一般的に利用されています。あるディレクトリ `foo` 以下のファイルを **gzip** (C.6.2 参照) を用いてまとめて圧縮して `foo.tar.gz` という名前のファイルとして保存するには

```
tar -zvcf foo.tar.gz foo/
```

とします。オプション `-c` が書庫を作成 (create) する指定です。`-v` は冗長 (verbose) メッセージ、`-z` は圧縮をかける、そして `-f file` がファイル名を指定するオプションです。逆に、圧縮書庫化されたファイルを展開するには

```
tar -zxvf foo.tar.gz
```

とします。オプション `-x` が引き出し (extract) の指定です。`-x` の代わりに `-t` を指定すると、ファイル名のリストを表示するだけで展開は実行されません。サブディレクトリ以下ではなくカレントディレクトリに展開される場合もあるので、まず確かめた方が無難です。なお tar でかめた場合には拡張子 `.tar` を、またさらに圧縮をした場合は `.tar.gz` あるいは `.tgz` を付けるのが常識です。また、圧縮に **bzip2** (C.6.3 参照) を用いる場合にはオプション `I` を使います。例えば

```
tar -Ivcf foo.tar.bz2 foo/
tar -Ivxf foo.tar.bz2
```

ファイル指定オプション `-f` を付けなければ、標準入力や標準出力がデータの通り道となります。その場合も心得ておくといいかもかもしれません。例えば、

```
tar -c directr/* | cat > directr.tar
tar -zcv directr | cat > dircetr.tar.gz
cat directr.tar | tar -tv
cat directr.tgz | tar -zx
cat directr.tgz | tar -zxf -
```

などです。最後の方法は標準入力を特殊ファイル名 “-” で指定できることを示したものです。

### C.6.2 gzip, gunzip

一つのファイルを圧縮するツールです。圧縮するファイルが `foo` である場合

```
gzip foo
```

とすると、`foo.gz` が作成され `foo` は削除されます。元の `foo`を残したい場合には標準出力にデータを流すオプション `-c` を用いて

```
gzip -c foo | cat > foo.gz
```

とします。gzip で圧縮されたファイルには拡張子 `.gz` を付けるのが常識です。gzip で圧縮されたファイルを伸長するにはオプション `-d` を使います。**gunzip** を用いる方が覚えやすいかもしれません。

```
gzip -d foo.gz あるいは gunzip foo.gz
```

`foo` が復元して `foo.gz` は削除されます。疑り深い人は

```
gzip -dc foo.gz | cat > foo あるいは
gunzip -c foo.gz | cat > foo
```

という方法を好むかもしれません。圧縮率と処理時間は相反するので、どちらを優先するかというオプション - 数字 (1-9) が存在します。どの程度の差が出るのか試してみましょう。

関連ツールには、圧縮された状態のファイルを less する `zless`、実行可能なまま圧縮を掛ける `gzexe`、圧縮ファイル同士の比較 `zcmp`、compress で圧縮されたファイルを gzip で新たに圧縮し直す (サイズが小さくなります) `znew` などがあります。

### C.6.3 bzip2, bunzip2

gzip とは異なる圧縮アルゴリズムを用いているツールです。一般的に gzip より圧縮率が高くなります、が処理時間が長くなります。オプションは gzip にほぼ同じです。拡張子は `.bz2` です。



### C.6.4 compress, uncompress

UNIX 標準の圧縮ツールです。いまだにこの形式で圧縮されている場合もあります。伸長のオプション `-d` が無いので注意してください。拡張子は `.Z` です。GNU tar は `compress` を認識します。したがって、`compress` 自身はもう不要かもしれません。

### C.6.5 lha, zip

MS 社関連の OS では `lha` (拡張子 `.lzh`) や `zip` (拡張子 `.zip`) が一般的に使われているようです。Linux にも移植されていますから問題なく処理できます。それぞれのツールによる圧縮具合を表 21 に示します。バイナリファイル `mule` (2666388 bytes) とテキストファイル `test.txt` (1299472 bytes) を使いました。

表 21 圧縮ツールの比較

| ツール      | mule    | test.txt |
|----------|---------|----------|
| gzip     | 758026  | 433104   |
| bzip2    | 740733  | 128256   |
| compress | 1068902 | 516577   |
| lha      | 778154  | 484976   |
| zip      | 761410  | 434780   |

### C.6.6 mtools

ファイルを保存するには FD が (論理) フォーマットされている必要があります。市販の DOS/V 用の FD は MS-DOS フォーマットされている筈ですから、そのまま使うことにしましょう。UNIX 系 OS から MS-DOS 用の FD に読み書きを行う便利なツールが `mtools` です。複写するには

```
mcopy file.ux a:\file.dos
mcopy a:\file.dos file.ux
```

とします。MS-DOS では **8 文字+拡張子 3 文字** という名前の長さの上限がありますから注意しましょう (かなり窮屈です)。また、テキストファイルの場合には、行末のコードを相互に変換した方が良いのでオプション `-t` を付けてください。DOS-FD 上のファイルの削除は `mdel`、リスト表示は `mdir`、テキストファイルの内容表示は `mttype` で実行できます。また、フォーマットをやり直すには `mformat` です。実験的ですが、`mformat -F a:` で MS-Win の FAT32 用のフォーマットも可能です。これですと、長い名前も使えます。

④ FD を Linux の ext2 でフォーマットするには、

```
mke2fs /dev/fd0
```

を実行します。しかし、このコマンドの実行には root 権限が必要です。また、使用する場合には、

```
mount /mnt/floppy
umount /mnt/floppy ← FD を抜く前に必ず実行
```

のように `mount/unmount` する必要があります (一般 user では不可かもしれません)。少し手間ですが、`mount` されていれば Linux から普通のファイルとして扱えますから、全てのコマンドが適用できます。

### C.6.7 split

一つのファイルを指定した大きさのファイルに分割するツールです。書式は

```
split [-b bytes[bkm]] [infile [outfile-prefix]]
```

`-b` で大きさを指定します。単位に `k` (キロ) `m` (メガ) を使うこともできます。outfile-prefix を指定すると、その後ろに `aa,ab,ac,...` と順番付けをします。

```
$ split -b 100k circle.pgm circle.
$ ls -l circle.*
-rw-r--r-- 1 matuda users 102400 Mar 25 00:26 circle.aa
-rw-r--r-- 1 matuda users 102400 Mar 25 00:26 circle.ab
-rw-r--r-- 1 matuda users 53645 Mar 25 00:26 circle.ac
-rw-r--r-- 1 matuda users 258445 Mar 24 23:59 circle.pgm
```

これらをまとめてファイルを復元するには `cat` を使って

```
cat circle.a* > circle.pgm
```

とします。集めようとしているファイル `circle.a*` は、他のファイルが紛れ込まないように注意して名前を指定してください。

### C.6.8 uuencode, uudecode

ネットワーク資源が乏しい時代には、メールで大きなファイルを送るのは御法渡でした。実際 50 キロバイトを超えるようなメールは配送されないといったこともあったようです。現在は、数メガを超える画像ファイルを何のためらいもなく添付する人々が増えました。大きさの制限は緩くなったのですが、今も昔も、メールにはバイナリを直接含ませることができません。バイナリファイルは、メールに含められるように変換する必要があるのです。uuencode は、バイナリファイルを ASCII (7bit) テキストファイルに変換するツールです。使い方は

```
uuencode infile name > outfile
cat infile | uuencode name > outfile
```

が標準的です。サイズを小さくするために普通は `gzip` されたファイルをエンコードする場合があります。具体的な手順を追ってみましょう。

```
$ ls -s 15Linux.dvi
208 15Linux.dvi  ← 208k もあります
$ gzip 15Linux.dvi; ls -s 15Linux.dvi.gz
67 15Linux.dvi.gz  ← 小さくなりました
$ uuencode 15Linux.dvi.gz 15Linux.dvi.gz > 15Linux.dvi.gz.uu
$ ls -s 15Linux.dvi.gz.uu
92 15Linux.dvi.gz.uu  ← 最終的には 92k になりました
$ head -5 15Linux.dvi.gz.uu  ← 最初の 5 行を表示します
begin 644 15Linux.dvi.gz
M'XL("!!L^C8''SSU3&EN=7@N9'9I'.Q;#7!;57:6I=CY(1#"/PD$'"1C$O_(
MLAU;MN7_6++\E\Q0,20DD6TY_I$MIY+!0!*X!;IE09J\O)50D]E56J!ETRV[
M6Z93"FU#6;KLJ%N2HD'+TX0R_,>=@<0A(9"L>\ZY]ST]R0YEEMT..Q.-GR2_
M=^\]YYY[[G=[M$9?<9#>UZZH5('+\`2V]>[`Z`V3L>&!L/W%QDM5H+S,4%
```

こうして、15Linux.dvi.gz.uu という ASCII ファイルが出来ました。uudecode は、uuencode されたファイルからバイナリファイルを復元します。これは単純に

```
uudecode foo
```

とします。受信したメールを直接処理しても大抵は旨くデコードできます。なぜなら uuencode 部分は begin と end で囲まれているので、uudecode が自動的に判断するからです。

#### ish, uueview

MS-DOS の世界では、バイナリからテキストへのエンコードは ish というツール（開発者は日本人です）がよく使われていました。後にそれは、UNIX にも移植されました。他にも MS-Windows では MIME 標準の Base64, Macintosh では BinHex でエンコードされる場合があります。uueview はそれらにも対応した強力なデコーダーで、UNIX 以外の OS と添付バイナリファイルをやり取りする上では必要不可欠なツールです。

また、わざわざテキストを base64 でエンコードして送ってくる MS-Win ユーザーがいます（テキストはそのまま含めましょう、圧縮したならば話は別ですが）。このような困ったメールに対処するには、漢字コード変換ツール **nkf** が使えます。base64 エンコードされ部分を取り出して

```
nkf -mB encodedfile
```

とすれば、テキストが復元されます。

## C.7 bc

時折  $\pi$  の値を何億桁も計算する競争が新聞の記事になることがあります。このことから判るように、任意精度の計算はコンピュータの能力および言語・アルゴリズムの確かさのデモンストレーションとなります。C 言語では、倍精度変数 double は 15 桁の精度しかありませんし、整数も 32 ビットなので限界があります。めったに困ることはないのですが、統計力学などで顔を出す  $N$  の階乗  $N!$  がどこまで計算できるか試してみるといいです。32 ビット整数では、50! さえ計算できないのです。結局、桁の長い数値を扱うには、専用のライブラリを使う必要があるのですが、gcc の標準にはないのですぐには利用できません。では、例えば自然対数の底や  $\pi$  の値を 50 桁で表示してみたい等という場合にはどうしたらいいのでしょうか。答えは“**bc を使え**”です。bc は対話的に使うことのできる任意精度計算機

です。また、簡単なプログラムを組むことも可能です。制御文の書式は C 言語に準じていますが、標準で使える関数が少ないので、自分で関数定義をすることになるでしょう。

対話的に使ってみましょう。組み込み数学関数を使うオプション -l とバージョンなどのメッセージを表示しないオプション -q を指定して起動します。

```
$ bc -lq
s(1)  ← sin(1) の値
.84147098480789650665
e(1)  ← e1 すなわち e の値
2.71828182845904523536
scale=50  ← 小数点以下を 50 桁に
pi=4*a(1)  ← 変数 pi に  $\pi$  の値を代入
pi  ←  $\pi$  の値
3.14159265358979323846264338327950288419716939937508
quit  ← あるいは [Ctrl]+[d] で終了
$
```

続いて  $N$  の階乗の値を 50 まで表示するプログラムを組んでみましょう。bc のオンラインマニュアルに記載がありますが、以下の内容で、fact.bc という名前のファイルを作成してください。

```
#!/usr/local/bin/bc
```

```
define f(x) {
    if (x <= 1) return (1);
    return (f(x-1) * x);
}
for (i=1;i<50;i++) f(i);
quit
```

これを bc にファイルを読み込んで実行させます。

```
$ bc -lq fact.bc
1
2
6
24
120
720
(略)
...
1241391559253607267086228904737337503852148635467760000000000
608281864034267560872252163321295376887552831379210240000000000
```

繰り返しますが、C 言語でこのプログラムを書くのは結構大変なのです。最後に read() を使って、対話的に  $N!$  を求めるプログラムを作ってみましょう。

```
#!/usr/local/bin/bc -lq
```

```
define f(x) {
    if (x <= 1) return (1);
    return (f(x-1) * x);
}

while(1) {
    print "N = ? (0 で終了) "; n=read();
    if (n == 0) break;
    f(n);
}
quit
```

のような感じではどうでしょう（ファイル名 ifact.bc）。

```
$ bc -lq ifact.bc
N = ? (0 で終了) 10
3628800
N = ? (0 で終了) 50
30414093201713378043612608166064768844377641568960512000000000000
N = ? (0 で終了) 0
cck999:~$
```

かなり楽しめますね。

表 22 bc で使われる重要な命令：C 言語と異なるもの

| 特徴的なコマンド |                                  |
|----------|----------------------------------|
| scale=N  | 小数点以下を N 桁にする                    |
| z=read() | 変数 z に標準入力から値を読み込む               |
| x^N      | x の N 乗：整数に限られています。              |
| 関数       |                                  |
| sqrt(x)  | $\sqrt{x}$ （マニュアルには文であると記されています） |
| s(x)     | $\sin(x)$                        |
| c(x)     | $\cos(x)$                        |
| a(x)     | $\arctan(x)$                     |
| l(x)     | $\ln x, \log_e(x)$               |
| e(x)     | $e^x, \exp(x)$                   |
| j(n,x)   | $n$ 次 Bessel 関数 $J_n(x)$         |

# 索引

`**argv`, 11  
`++`, 17  
`--`, 17  
`_Complex`, ⇒ 複素数型

`a.out`, 5  
`argc`, 11

`bash`, 58  
`bc`, 67  
`break`, 15, 19  
`bunzip2`, 65  
`bzip2`, 65

C99, 8  
`cat`, 61  
`cd`, 61  
`char`, ⇒ 整数型  
`column`, 63  
`complex`, ⇒ 複素数型  
`compress`, 66  
`continue`, 19  
`cp`, 62  
CR + LF, 3

`date`, 64  
`diff`, 64  
`dirs`, 61  
`do...while`, 18  
`dos2unix`, 3  
`double`, ⇒ 実数型  
`drand48`, 26

`expand`, 62

`fflush()`, 48  
`fgetc()`, 36  
`fgets()`, 36  
`find`, 62  
`float`, ⇒ 実数型  
`fold`, 62  
`for`, 18  
`fputc()`, 36  
`fputs()`, 36  
`fread()`, 49  
`fwrite()`, 49

`gcc`, 5  
`getc()`, 36  
`getchar()`, 36  
`gets()`, 36  
GNU, 5  
`goto`, 20  
`grep`, 64  
`gunzip`, 65  
`gzip`, 65

`head`, 61  
HOME, 59

`iconv`, 2  
`if...else statement`, 13  
`indent`, 64  
`index()`, 36  
`infocmp`, 28  
`int`, ⇒ 整数型  
`int32_t`, ⇒ 整数型  
`ish`, 67

K&R, 1  
Kernihan B.W., 1

KNOPPIX, 7  
`kterm`, 28

`less`, 61  
LF, 3  
`lha`, 66  
`locate`, 62  
`login`, 58  
`logout`, 58  
`lonf double`, ⇒ 実数型  
`long int`, ⇒ 整数型  
`long long int`, ⇒ 整数型  
`ls`, 6, 60  
`lv`, 2

`mkdir`, 60  
`mttools`, 66  
`mv`, 62

`nkf`  
--unix, 4  
null character, 33

`paste`, 63  
PATH, 6  
`permission`, 60  
`popd`, 61  
`printf()`, 5, 10  
`pushd`, 61  
`putc()`, 36  
`putchar()`, 36  
`puts()`, 36  
`pwd`, 60

`qsort()`, 54

`return`, 20  
`rev`, 61  
`rindex()`, 36  
Ritche D.M., 1  
`rm`, 62  
`rmdir`, 60

`scanf()`, 10  
`secure`, 6  
`short int`, ⇒ 整数型  
`signed`, ⇒ 整数型  
`sizeof`, ⇒ `sizeof` 演算子  
`sort`, 65  
bubble sort, 52  
`sorting`, 51  
`split`, 66  
`stderr`, ⇒ 標準エラー出力  
`stdin`, ⇒ 標準入力  
`stdout`, ⇒ 標準出力  
`strcat()`, 36  
`strchr()`, 36  
`strcmp()`, 34  
`strcpy()`, 34  
`strlen()`, 34  
`strncpy()`, 34  
`strpbrk()`, 36  
`strchr()`, 36  
`strstr()`, 36  
`struct`, 42  
`switch...case statement`, 15  
`system()`, 49

`tac`, 61  
`tail`, 61  
`tar`, 65

`tee`, 64  
`terminfo`, 28  
`time`, 52  
`typedef`, 44

`uncompress`, 66  
UNICODE, 3  
`unix2dos`, 3  
`unsigned`, ⇒ 整数型  
`utf8`, 3  
`uudecode`, 66  
`uudeview`, 67  
`uuencode`, 66

VmWare, 7

`while`, 17

Xemacs, 6

`zip`, 66

ASCII コード, 2, 9  
値渡し (pass by value), 27  
アドレス演算子, 24  
アドレス演算子 (&), 28  
アドレス渡し, 28  
アルゴリズム, 13

インタープリタ, 5

エディター, 5  
エラー処理, 12  
演算子, 7  
—の優先順位 (優先度), 7

拡張子, 5  
拡張ユニックスコード (eucj), 2  
可視文字, 5  
仮想化, 7  
型, 8

暗黙の型変換, 25  
型指定子, 8  
型変換, 25  
基本型, 8  
キャスト演算子, 25  
実数型, 9  
double, 9  
float, 9  
long double, 9  
整数型, 9  
char, 9  
int, 9  
int32\_t, 9  
long int, 9  
long long int, 9  
short int, 9  
signed, 9  
unsigned, 9  
派生型, 8  
複素数型, 10  
void 型, 8  
明示的型変換, 25

画面制御, 28  
寡黙な OS, 6  
カレントディレクトリ (./), 6, 59  
関係演算子, 13  
—と代入演算子の混同, 13  
漢字コード, 2  
関数, 27  
—の宣言, 27



- の定義, 27
- の引数, 27
- のプロトタイプ宣言, 27
- の呼び出し, 27
- 間接演算子 (\*), 29
- キャスト演算子, 25
- 行末コード, 3
- 許可属性, 60
- 繰り返し
  - 実数によるループ制御, 20
- 繰り返し (loop), 17
- 計算量のオーダー, 52
- 減分演算子 (--), 17
- 構造化プログラミング, 57
- 構造体, 42
  - アロー演算子, 43
  - タグ名, 42
  - ドット演算子, 43
  - の宣言, 初期化, 42
  - の代入, 42
  - のポインタ宣言, 43
  - メンバ, 42
- 後置記法, 17
- 固定小数点数, 7
- コンパイラ, 5
- 再帰関数, 30
- sizeof 演算子, 15
- 3 項演算子 (a?:b:c), 15
- 算術演算子, 17
- C 言語, 1
- シェル, 58
- 識別子, 8
- JIS コード (jis), 2
- 実行ファイル, 5
- 実数型, ⇒ 型
- シフト JIS コード (sjis), 2
- 条件演算子 (a?:b:c), 15
- 条件分岐, 13
- 条件分岐命令, 13
- 剰余, 7
- 剰余演算子, 23
- 書式付き出力関数, 10
- 書式付き入力関数, 10
- 真偽値, 13
  - false, 13
  - true, 13
- 図式化, 56
  - PAD, 56
  - pad2ps, 57
  - フローチャート (PF), 56
- ストリーム (stream), 47
- 整数型, ⇒ 型
- 整列 (sort)
  - クイックソート, 53
  - バブルソート (bubble), 52
- 整列 (sorting), 51
  - 単純選択ソート (selection), 52
- 絶対パス, 59
- セミコロン (;), 5
- 選択 (selection), 57
- 前置記法, 17
- 相対パス, 59
- 増分演算子 (++), 17
- 添字演算子 ([ ]), 22
- ソースファイル, 5
- 代入 (assignment), 13
  - 複合代入演算子 (op=), 17, 25
  - 関係演算子との混同, 13
  - 単純代入演算子 (=), 13, 17
- 多次元配列, 23
- 多重ループ, 18
- 端末エミュレータ, 6
- テキスト形式のデータ, 48
- テキストファイル, 4
- 内部表現, 48
- ナル文字 (\0, null character), 33
- 排他的論理和 (exclusive OR), 13
- バイナリ形式のデータ, 48
- パイプ (|), 63
- 配列, 22
  - の初期化, 22
  - 添字, 22
  - 添字演算子, 22
- 反復 (iteration), 57
- 標準出力, 5
- 標準数学関数, 31
- ファイル, 47
  - のオープン, 47
  - のクローズ, 48
- FILE 構造体, 47
- ファイル入出力, 47
- ファイルポインタ, 47
  - の取得, 47
- 不可視文字, 5
- 複素数
  - 型汎用マクロ, 50
  - の計算, 50
- 複素数型, 10
- 符号化, 2
- 浮動小数点数, 7
- プリプロセッサ命令, 46
  - #define, 46
  - #include, 46
  - マクロ定義, 46
- プロセス (process), 49
- プロンプト, 6
- 変換子, 5
- 返却値 (return value), 20
- 変数
  - のアドレス, 29
- void 型, 10
- ポインタ, 38
  - アドレス演算子 (&), 38
  - アドレス定数, 38
  - 関数へのアドレス渡し, 39
  - 間接演算子 (\*), 38
  - 定数ポインタ, 38
  - と配列名, 38
  - の演算, 41
  - の配列, 41
  - 複雑なポインタ宣言, 38
  - ポインタ変数, 38
  - ポインタ変数の初期化, 39
- ホームディレクトリ, 59
- マクロ定義, 46
  - 関数形式, 46
  - 副作用, 46
- 無条件分岐, 19
  - break, 19
  - continue, 19
  - goto, 19
  - return, 19
- main 関数, 5, 11
  - の引数, 11
- 文字
  - 文字の分類, 35
- 文字定数, 15
  - は int 型, 15
- 文字列, 33
  - の検索, 36
  - の操作, 34
  - の長さ, 34
  - の比較, 34
  - の複写, 34
  - の連結, 36
- ユニコード, 3
- 予約語, 8
- リダイレクト (>, <, >>), 63
- 連接 (sequential processes), 57
- 論理演算, 13
- 論理演算子, 13
- 論理演算表, 13