C言語これくらい覚えよう

桂田 祐史（かつらだ まさし）

2017年9月29日, 2019年10月14日

この文書は http://nalab.mind.meiji.ac.jp/~mk/labo/text/ に最新版を置きます。

入出力、繰り返し、関数を自分で定義する、ベクトルと行列、これくらいできれば…

目 次

1 勉強の仕方

(新装開店を目指し準備中 — もうそろそろ現象数理学科の学生向けの案内を書くべきだ)

2 C コンパイラー

C 言語で書かれたプログラムをコンピューターが理解できる形式 (機械語, machine language) に翻訳するプログラムを **C コンパイラー** (C compiler) と呼ぶ。

(普通は単にコンパイルするだけでなく、必要なライブラリィとリンクすることで、いわゆる実行形式のファイルを作る。)

現象数理学科 Mac では、Apple の用意した Xcode と呼ばれるソフトウェアをインストールしてあり、そこに Clang+LLVM のコンパイラー `cc` が含まれている。

現象数理学科 Mac では、グラフィックス・ライブラリィ GLSC を利用する C プログラムのコンパイルのために `cglsc` コマンドというのを用意してある。それは中から cc を呼び出しているので、結局は Clang+LLVM を利用することになる。

`cc` の使い方は、UNIX で伝統的な `cc` の使い方と互換性がある。よく使うオプションなどは「C 言語で数値計算プログラミング」1. コンパイル、実行の仕方[1] で解説しておいた。

(脱線) アセンブリー言語に翻訳してみる 機械語はビット列なので人間には読みにくいが、機械語とほぼ一対一に対応するアセンブリー言語という形式に変換すると、どのような機械語に変換されるのかが (まあまあ) 分かりやすい。`cc -S` とすると、C 言語のプログラムをアセンブリー言語に変換出来る。

アセンブリー言語のプログラムを機械語に変換することを「アセンブルする」と言い、アセンブルするためのプログラムをアセンブラーと呼ぶ。実は `cc` はアセンブラーの機能も備えている。

以下の例では、小さな C プログラム `prog.c` を、アセンブリー言語に翻訳したプログラム `prog.s` を作り、それを表示した後、`prog.s` をアセンブル＆リンクして実行形式 `a.out` を作って実行している。

[1] `http://nalab.mind.meiji.ac.jp/~mk/labo/studying-C/Programing-in-C/node5.html`

(以下で bash-3.2$ はプロンプトで、これはユーザーが入力するものではない。)

```
bash-3.2$ ls
prog.c
bash-3.2$ cat prog.c
#include <stdio.h>

int main(void)
{
  int a,b,c,d;
  a = 1; b = 2; c = 3;
  d = a * b + c;
  printf("%d*%d+%d=%d\n", a, b, c, d);
  return 0;
}
bash-3.2$ cc -S prog.c
bash-3.2$ ls
prog.c  prog.s
bash-3.2$ cat prog.s
(中略)
bash-3.2$ cc prog.s
bash-3.2$ ls
a.out*  prog.c  prog.s
bash-3.2$ ./a.out
1*2+3=5
bash-3.2$
```

次に (macOS 上の cc で作成した場合の) prog.s の内容を掲げる。$1 \times 2 + 3$ という計算をして、printf() を呼び出しているのが分かるだろうか？

prog.s — アセンブリ言語プログラムの例

```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 13
        .globl  _main                   ## -- Begin function main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq   %rbp
Lcfi0:
        .cfi_def_cfa_offset 16
Lcfi1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Lcfi2:
        .cfi_def_cfa_register %rbp
        subq    $32, %rsp
        leaq    L_.str(%rip), %rdi
        movl    $0, -4(%rbp)
        movl    $1, -8(%rbp)
        movl    $2, -12(%rbp)
        movl    $3, -16(%rbp)
        movl    -8(%rbp), %eax
        imull   -12(%rbp), %eax
        addl    -16(%rbp), %eax
        movl    %eax, -20(%rbp)
        movl    -8(%rbp), %esi
        movl    -12(%rbp), %edx
        movl    -16(%rbp), %ecx
        movl    -20(%rbp), %r8d
        movb    $0, %al
        callq   _printf
        xorl    %ecx, %ecx
        movl    %eax, -24(%rbp)         ## 4-byte Spill
        movl    %ecx, %eax
        addq    $32, %rsp
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function
        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz  "%d*%d+%d=%d\n"


.subsections_via_symbols
```

3　簡単な数値計算プログラミングで良く使う機能

3.1　最短の C プログラム

C のプログラムは関数定義の集合である。特に main() という名前の関数は必ず存在する必要がある (ここから実行が開始される)。そこで最短の C プログラムは次のようになる。ここでは説明のための注釈 (/* と */ で括られた部分) を書いたのであまり短くはないが。

prog01.c

```
/* prog01.c --- (注釈は除いて) 最短の C プログラム */

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

コンパイルと実行

```
bash-3.2$ cc -o prog01 prog01.c  (あるいは cglsc prog01.c)
bash-3.2$ ./prog01
bash-3.2$                        ← 確かに何もしない
```

なお、警告が出ても良ければ、

これこそ本当の最短 C プログラム？ (ちょっと文法違反しているけど)

```
main(){}
```

という 8 文字のプログラムに切り詰めることも出来る。

なお、最近の C 言語では、// から行末までは注釈という、1 行注釈が使えるようになったので、上のプログラムの 1 行目は

```
// prog01.c --- (注釈は除いて) 最短の C プログラム
```

のように書くことも出来る。

蛇足 (main() 中の return の意味) `return 0;` はどういう意味があるかというと、OS に 0 という値を返している。実際、上のプログラムの実行直後に`echo $status` と打つと 0 と表示される。プログラムがどういう状況で終了したかを OS に知らせる目的で使うことができる。通常 0 は正常に終了したことを示す習慣である。 ■

3.2 Hello world

次が有名な[2] “Hello world プログラム” である。

prog02.c

```
/* prog02.c --- 有名な Hello world プログラム */

#include <stdio.h>

int main(void)
{
  printf("Hello, world.\n");
  return 0;
}
```

コンパイルと実行

```
bash-3.2$ cc -o prog02 prog02.c  (あるいは cglsc prog02.c)
bash-3.2$ ./prog02
Hello, world.
bash-3.2$
```

[2] 上で紹介したカーニハン&リッチーの本の最初のプログラム例である。

- printf() は標準出力 (通常は画面と結合) への書式付き出力を行う関数である。
- 二つの引用符 " " で囲まれた部分は文字列となる。文字列中の \n は改行文字を表す。(なおバックスラッシュ \ は、JIS コードでは円記号 ¥ に相当する。)

printf() による文字列の標準出力への出力

```
printf(文字列);
例えば
printf("こんにちは\n");
```

- printf() の「宣言」をするために stdio.h というファイルを #include という命令を用いてインクルードする (stdio=standard input and output の略, h は header file の頭文字)。その実体は /usr/include/stdio.h というファイルである。

問 一つの printf() で、縦に

```
あ
い
う
え
お
```

と書くにはどうしたら良いか？

3.3 整数の入出力と「5 則」演算

次に掲げるプログラムは、二つの整数を読み込んで、それらの和、差、積、(整数の) 商、余りを表示するプログラムである。

prog03.c

```
/* prog03.c --- 整数の入出力と 5 則 (?) 演算 */

#include <stdio.h>

int main(void)
{
  int a, b;

  printf(" 二つの整数を入力して下さい: ");
  scanf("%d%d", &a, &b);

  printf(" 入力された整数は %d, %d です。\n", a, b);

  printf(" 和=%d, 差=%d, 積=%d, 商 (の整数部分)=%d, 余り=%d\n",
         a + b, a - b, a * b, a / b, a % b);

  return 0;
}
```

C 言語で扱える値には「型」というものがある。

- (ある範囲の[3]) 整数値を記憶するために **int** という型が用意されている (整数は英語で integer, 形容詞は integral)。
- int 型の変数を宣言 (定義) するには

```
int 変数名のリスト;
例えば
int a;
int b, c, d;  /* カンマで区切って名前を並べる */
```

とする。

- **標準入力** (普段はキーボードからの入力に結合) から整数型の変数に値を入力するには、素朴には `scanf()` という関数を使う (`scanf()` は、実はトラブル・メーカーなのだが、とりあえず無視)。
- **標準出力** (普段は端末画面への出力に結合) に整数型の式の値を出力 (表示) するには、`printf()` という関数を使う。
- `scanf()`, `printf()` で「**書式** (format)」を指定する場合、整数型のデータには普通 `%d` を用いる (d は decimal (10 進数) の略。実は 8 進数、16 進数などで入出力できる)。

`print()` で整数を 10 進数で出力)

```
printf("値は %d\n", 整数型の式);
例えば
printf("a=%d, b=%d\n", a, b);
printf("a の 3 倍=%d\n", 3 * a);
```

`scanf()` で 10 進数表記された整数を整数型変数に入力

```
scanf("%d", &整数型変数);
例えば
scanf("%d", &a);
scanf("%d%d", &b, &c);
```

&変数名 として、変数へのポインター (アドレス) を `scanf()` に渡すのがミソ。`scanf()` の使い方は、とりあえずパターンとして覚えてしまおう。

- 整数型の式を作る演算子として、和 `+`, 差 `-`, 積 `*`, 整商 `/`, 剰余 `%` がある。

3.4 浮動小数点数の入出力と四則演算

C 言語で整数以外の実数を扱うには、**浮動小数点数** (floating point numbers) という型を用いる。

[3]Mac 環境の cc (中身は Clang+LLVM らしい) では、$-2^{31} \leq n \leq 2^{31}-1$ を満たす整数 n. $2^{31} = 2147483648 \fallingdotseq 21$ 億 $= 2.1 \times 10^9$ であり、10 進 9 桁程度。これ以外に、通常 64 ビットの `long long int` 型もある (定数を表すには、`2147483649LL` あるいは `214748364911` のように "`LL`" や "`ll`" をつける。`printf()` や `scanf()` の書式では、"`%lld`" のようにする。)。さらに `__int128` という型もあるが、まだ十分に規格が整備されていないので使いにくい。

prog04.c

```
/* prog04.c --- 実数の入出力と四則演算 */

#include <stdio.h>

int main(void)
{
  double a, b;

  printf(" 二つの実数を入力して下さい: ");
  scanf("%lf%lf", &a, &b);

  printf(" 入力された実数は %f, %f です。\n", a, b);

  printf(" 和=%f, 差=%f, 積=%f, 商=%f\n", a + b, a - b, a * b, a / b);

  return 0;
}
```

- 精度、範囲により三種類の型 **float**, **double**, **long double** がある。最近、実質的標準となっている IEEE754 規格を採用したシステムでは、精度は 10 進法に換算して、`float` が 7 桁, `double` が 16 桁弱である。値の範囲は `float` が $10^{-38} \sim 10^{38}$, `double` が $10^{-308} \sim 10^{308}$ である。
- 私のおすすめは「浮動小数点数は double だけ使う」である (理由は省略する)。

書式指定ミニマム

- printf() で double データを小数形式で出力するには %f という書式を使う。

```
printf("a=%f\n", a);
printf("b=%f, c=%f\n", b, c);
```

- scanf() で、double 型の変数に入力する場合の書式は %lf (他にもあるがこれ一つで十分) を使う (l はエル L の小文字)。

```
scanf("%lf", &a);
scanf("%lf%lf", &b, &c);
```

入力と出力で書式が %lf, %f と食い違うのは、歴史的経緯でしかたない。

- その他に指数形式で出力する %e, 小数形式と指数形式のうちで「コンパクト」な方を (そのときの式の値に応じて) 選ぶ %g がある。

- 表示の桁数 (n), 小数点以下の数字の桁数 (m) を指定するには $\boxed{\%n.m\mathtt{f}}$ とする。

```
printf("%20.15f\n", a);
```
(幅 20 桁、小数点以下 15 桁、小数形式で表示する)

例えば

```
double x=1.2345678901234567;
printf("12345678901234567890\n");
printf("%8.4f\n", x);
```

とすると

```
12345678901234567890
  1.2346
```

となる。

int と double の使い分け方

int で表すことの出来る値は double でも表すことが出来る場合が多いが、配列の添字など double は使えない場合もあり、きちんと使い分けるべきである。

(1) 整数でない半端な値が出て来る可能性があれば当然 double

(2) 配列の添字は (絶対) int

(3) 集合の要素数、番号、繰り返しの回数など、本来整数値しか取り得ない場合は (普通は) int 時々 int では表現不可能だが、double では表現できるような値を扱うために double を使う、ということはある。— しかし、最近では long long int 型が安心して使えるようになったので[a]、この用法は時代遅れかもしれない。

[a] 細かいことを言うと、long long int は (普通) 64 ビット、double の仮数部は (IEEE754 の場合) 53 ビットなので、long long int の方が 11 ビット分多い。

3.5 数学関数ライブラリィの利用

通常数学に現われる初等関数や、平方根 $\sqrt{\ }$, n 乗根 $\sqrt[n]{\ }$, 絶対値 $|\ \ |$ などは、C 言語では数学関数ライブラリィに用意されている。

- 使用する数学関数の

  (1) 返す値の型

  (2) 引数 (argument) の型

  を宣言するためには `math.h` をインクルードするのが手っ取り早い。一度 /usr/include/math.h を見てみることを勧める。

- リンクするには `-lm` オプションが必要である[4]。

prog05.c

```
/* prog05.c --- 数学 (関数) ライブラリィにある関数の呼び出し */

#include <stdio.h>
#include <math.h>    /* この行に注目 */

int main(void)
{
  double x;

  printf("一つの実数を入力してください: ");
  scanf("%lf", &x);

  /* ルート (非負の平方根) */
  printf("sqrt(%g)  =%g\n", x, sqrt(x));
  /* 三角関数 sin */
  printf("sin(%g)   =%g\n", x, sin(x));
  /* e を底とする指数関数 */
  printf("exp(%g)   =%g\n", x, exp(x));
  /* 自然対数 */
  printf("log(%g)   =%g\n", x, log(x));
  /* 常用対数 (10 を底とする対数) */
  printf("log10(%g) =%g\n", x, log10(x));
  /* 双曲線関数 hyperbolic sin) */
  printf("sinh(%g) =%g\n", x, sinh(x));
  /* 巾乗 (power) --- ここでは 3 乗根 */
  printf("pow(%g,%g)=%g\n", x, 1.0/3.0, pow(x, 1.0/3.0));
  /* 逆三角関数 Arctan */
  printf("atan(%g)  =%g\n", x, atan(x));
  /* 絶対値 */
  printf("fabs(%g)  =%g\n", x, fabs(x));
  /* 整数部分 (-∞に向かっての切り捨て = いわゆる Gauss の括弧) */
  printf("floor(%g) =%g\n", x, floor(x));
  /* ∞に向かっての切り上げ */
  printf("ceil(%g) =%g\n", x, ceil(x));
  /* 最も近い整数への丸め (≒四捨五入) */
  printf("rint(%g) =%g\n", x, rint(x));

  return 0;
}
```

[4] ライブラリィは "libname.a" という名前のファイルとして格納されているのが普通で、この中に入っているコードをリンクするには `-l`name というコマンドライン・オプションをコンパイラーに指定する。数学関数ライブラリィの名前は `libm.a` なので (やけに短い名前ですね)、それをリンクするには `-lm` とすることになる。

コンパイルと実行

```
% cc -o prog05 prog05.c -lm  (-lm が必要。l はエル L の小文字。)
% ./prog05
(結果省略)
```

整数/整数 に注意

上のプログラムで $\frac{1}{3}$ を計算するのに `1.0/3.0` としているのに注意。`1/3` は整除で 0 になってしまう。`(double)1/(double)3` とすべきかもしれないが、入力が面倒だし、読みづらいので `1.0/3.0` あるいは `1.0/3` などにすることを奨める。
(**細かい注**) `1.0` という表記はコンピューターでなければ、常識的には有効数字 2 桁の数という意味になる。だからあまり誉められた書き方でないかもしれない。`1.` とする人もいるが… まあ好みの問題でしょうか。

円周率 π, 自然対数の底 e など、どうやって用意する？

簡単な応用として、円周率 $\pi = 3.1415926535\cdots$ や自然対数の底 (Napier の数) $e = 2.7182818284\cdots$ が必要な場合に

```
#include <math.h>
double pi, e;
...
int main()
{
  ...
  /* main() の先頭付近で pi, e を初期化 */
  pi = 4.0 * atan(1.0);
  e = exp(1.0);
```

のようなコードを書くというのがあります。`int main()` の前で宣言し (グローバル変数になります)、プログラム開始早々に代入文を実行して値を設定するわけです。

余談　いわゆる特殊関数が使いたい場合は、ライブラリィを探すことになるであろう。その場合、C よりは C++ の方が見つかりやすいかもしれない。

練習問題　実係数の 2 次方程式 $ax^2 + bx + c = 0$ ($a, b, c \in \mathbf{R}$, $a \neq 0$) を解くプログラムを作れ。

3.6　for 文による繰り返し

ある条件が満たされている間の繰り返しには `for()` が便利である。

```
for（最初にすること; 継続の条件; 各繰り返しの最後にすること）
  文;
```

括弧 { } でくくることにより、複数の文をまとめた複文が使える。

```
for（最初にすること; 継続の条件; 各繰り返しの最後にすること）{
  文1;
  文2;
  ...
  文n;
}
```

よくあるパターンとして、「n 回の繰り返し」をするために次のように書くことがある。

n 回の繰り返し

```
for (i = 0; i < n; i++) {
   文;
   文;
}
```

応用として、「$n = 1, 2, \ldots$ に対して 2^n, 2^{-n} を計算して表示せよ」という問題を解いてみよう。$x_n := 2^n$, $y_n := 2^{-n}$ とおくと、数列 $\{x_n\}$, $\{y_n\}$ は

$$x_0 = 1, \quad x_n = 2x_{n-1} \quad (n = 1, 2, \ldots),$$
$$y_0 = 1, \quad y_n = y_{n-1}/2 \quad (n = 1, 2, \ldots)$$

という漸化式で定義できることに注目し、それを用いて計算してみよう。

prog06.c

```
/* prog06.c --- for 文による繰り返し (1) */

#include <stdio.h>

int main(void)
{
  int i, n;
  double x, y;

  printf("自然数を一つ入力してください: ");
  scanf("%d", &n);

  x = 1.0; y = 1.0;
  for (i = 1; i <= n; i++) {
    /* x をその 2 倍で、y をその 1/2 倍で置き換える */
    x = 2 * x; y = y / 2;
    /* 次の %g を %f や %e に変えて試してみよう */
    printf("2 の%d 乗=%g, 1/2 の%d 乗=%g\n", i, x, i, y);
  }
  return 0;
}
```

prog06 の実行結果

```
% ./prog06
自然数を一つ入力してください: 20
2 の 1 乗=2, 1/2 の 1 乗=0.5
2 の 2 乗=4, 1/2 の 2 乗=0.25
2 の 3 乗=8, 1/2 の 3 乗=0.125
2 の 4 乗=16, 1/2 の 4 乗=0.0625
2 の 5 乗=32, 1/2 の 5 乗=0.03125
2 の 6 乗=64, 1/2 の 6 乗=0.015625
2 の 7 乗=128, 1/2 の 7 乗=0.0078125
2 の 8 乗=256, 1/2 の 8 乗=0.00390625
2 の 9 乗=512, 1/2 の 9 乗=0.00195312
2 の 10 乗=1024, 1/2 の 10 乗=0.000976562
2 の 11 乗=2048, 1/2 の 11 乗=0.000488281
2 の 12 乗=4096, 1/2 の 12 乗=0.000244141
2 の 13 乗=8192, 1/2 の 13 乗=0.00012207
2 の 14 乗=16384, 1/2 の 14 乗=6.10352e-05
2 の 15 乗=32768, 1/2 の 15 乗=3.05176e-05
2 の 16 乗=65536, 1/2 の 16 乗=1.52588e-05
2 の 17 乗=131072, 1/2 の 17 乗=7.62939e-06
2 の 18 乗=262144, 1/2 の 18 乗=3.8147e-06
2 の 19 乗=524288, 1/2 の 19 乗=1.90735e-06
2 の 20 乗=1.04858e+06, 1/2 の 20 乗=9.53674e-07
%
```

繰り返しの命令には、他に `while () ...` や `do ... while ()` などがある。

3.7 for 文による繰り返し (2) 応用: 級数の和

級数の和

$$S_n = a_1 + a_2 + \cdots + a_n$$

を計算するための定跡がある。

$$S_0 = 0$$

と約束すれば

$$S_i = S_{i-1} + a_i \quad (i = 1, 2, \ldots, n)$$

という漸化式が成り立つ。これから次の手順を得る。

$S_n = \sum_{i=1}^{n} a_i$ の計算手順

(1) 部分和 S_i を記憶しておく変数 `s` を定義する。

(2) `s = 0;`

(3) `s = s +` a_i`;` を $i = 1, 2, \ldots, n$ について繰り返す。

次のプログラムは、与えられた自然数 n に対して $1^2 + 2^2 + \cdots + n^2$ を計算して表示するものである。

prog07.c

```
/*
 * prog07.c --- for 文による繰り返し (2)
 */

#include <stdio.h>

int main(void)
{
  int i, n;
  double s;

  printf("自然数を一つ入力してください: ");
  scanf("%d", &n);

  s = 0.0;
  for (i = 1; i <= n; i++) {
    /* 次の文では s に i を加えている。これは s = s + i * i; とも書ける。 */
    s += i * i;
  }
  printf(" 1 から %d までの自然数の平方の和=%g\n", n, s);
  return 0;
}
```

後の練習問題にしておいた、$S_n = \sum_{k=0}^{n} \frac{1}{k!}$ を計算するプログラムを自力で書いてみると良い。**ヒント:** 一般項 $a_k = 1/k!$ も漸化式で計算すると都合が良い。「漸化式を使って計算しよう」[5] という資料を準備しておいた。

3.8 while による繰り返しと if 文による分岐 (場合わけ)

ある条件が満たされている間、文を繰り返し実行するために、while がある。

```
while (条件式)
  文;
```

条件式を作るためには、例えば以下のような比較演算子が使える。

==	(等しい)	!=	(等しくない)
<	(小さい)	<=	(以下≦)
>	(大きい)	>=	(以上≧)

条件式が成り立っているかどうかで、次に何を実行するか、場合わけをするために、if という命令がある。

(1)
```
if (条件式) 文;
```

(2)
```
if (条件式)
  文1;
else
  文2;
```

[5] http://nalab.mind.meiji.ac.jp/~mk/keisanki2-2005/zenkashiki/

prog08a.c

```
/*
 * prog08a.c --- 実係数２次方程式を解く
 */

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

int main(void)
{
  double a, b, c, D, sqrtD, re, im, x1, x2;
  printf("2 次方程式 a x^2+b x+c=0 を解く。\n");
  printf("a, b, c: ");
  scanf("%lf%lf%lf", &a, &b, &c);
  D = b * b - 4 * a * c;
  if (D > 0) {
    sqrtD = sqrt(D);
    x1 = (- b + sqrtD) / (2 * a);
    x2 = (- b - sqrtD) / (2 * a);
    printf("相異なる２つの実数解: %20.15g, %20.15g\n", x1, x2);
  }
  else if (D == 0.0) {
    printf("ただ一つの実数解 (重根): %20.15g\n", - b / (2 * a));
  }
  else {
    re = - b / (2 * a);
    im = sqrt(- D) / (2 * a);
    printf("相異なる２つの虚数解: %20.15g ±%20.15g  i \n", re, im);
  }
  return 0;
}
```

prog08.c

```
/* prog08.c --- while による繰り返しと if 文による分岐 (場合わけ) */

/*
 * 与えられた自然数が偶数ならば 2 で割り、そうでなければ 3 倍して 1 を
 * 加えるという操作を繰り返すと、最後には必ず 1 になる、らしい
 * (簡単そうだが証明されていない)。
 * このことを実験するプログラム。
 */

#include <stdio.h>

int main(void)
{
  int n;

  printf(" 自然数を入力してください: ");
  scanf("%d", &n);

  while (n != 1) {
    printf("%d\n", n);
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
  }
  printf("%d\n", n);
  return 0;
}
```

3.9 ユーザー定義の関数

自分で関数が定義できる。

- 関数宣言 (関数の返す値、引数の型を指定する) の書き方は、

```
関数の返す値の型名 関数名 (引数の型のリスト);
例えば
double square(double);
あるいは
double square(double x);
```

- 関数の定義は、まず

```
関数の返す値の型名 関数名 (引数の型 引数名 の繰り返し)
{
    ....
    ....
}
```

の形をしていて、値は

```
return 式;
```

という形の実行文で返す。例えば

```
double square(double x)
{
  return x * x;
}
```

例えば

$$f(x) := \begin{cases} \dfrac{\sin x}{x} & (x \neq 0) \\ 1 & (x = 0) \end{cases}$$

で定義される関数を定義するには次のように書けば良い。

prog09.c[6]

```
/* prog09.c --- ユーザー定義の関数 */

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

/* 次の文が関数 f の「宣言」 */
double f(double);

int main(void)
{
  int i, n;
  double a;

  printf(" 自然数を入力してください: ");
  scanf("%d", &n);

  a = 1.0;
  for (i = 0; i < n; i++) {
    a /= 2;
    printf("1-sin(%g)/%g=%g\n", a, a, 1.0-f(a));
  }
  return 0;
}

/* 以下、関数 f() の定義 */
double f(double x)
{
  if (x == 0.0)
    return 1.0;
  else
    return sin(x) / x;
}
```

3.10 簡単なファイル入出力

input.data

```
2 3
```

というように 1 行に 2 つの整数が記録されているファイルを読み込んで、その 2 数の和を計算し、その結果を

output.data

```
5
```

のようにファイルに記録するにはどうすれば良いか。

[6] http://nalab.mind.meiji.ac.jp/~mk/labo/text/cminimum-prog/prog09.c

prog13.c

```
/*
 * prog13.c --- fopen(), fclose(), fprintf(), fscanf() を使ったファイル入出力
 *  コンパイルは gcc -o prog13 prog13.c
 */

#include <stdio.h>

int main(void)
{
  int a, b, sum;
  FILE *in, *out;

  in = fopen("input.data", "r");
  /* 本当はここで in が NULL でないかチェックすべき */
  fscanf(in, "%d%d", &a, &b);
  fclose(in);

  sum = a + b;
  printf("%d と %d の和は %d\n", a, b, sum);

  out = fopen("output.data", "w");
  fprintf(out, "%d\n", sum);
  fclose(out);

  return 0;
}
```

なお、fopen() に失敗することも多い。エラー・チェックをするように修正すると次のようになる。

prog13check.c

```
/*
 * prog13check.c --- fopen(),fclose(),fprintf(),fscanf() を使ったファイル入出力
 *  コンパイルは gcc -o prog13 prog13.c
 */

#include <stdio.h>

int main(void)
{
  int a, b, sum;
  FILE *in, *out;

  if ((in = fopen("input.data", "r")) == NULL) {
    fprintf(stderr, "input.data を読むために開こうとして失敗しました。\n");
    exit(1);
  }
  fscanf(in, "%d%d", &a, &b);
  fclose(in);

  sum = a + b;
  printf("%d と %d の和は %d\n", a, b, sum);

  if ((out = fopen("output.data", "w")) == NULL) {
    fprintf(stderr, "output.data を書くために開こうとして失敗しました。\n");
    exit(1);
  }
  fprintf(out, "%d\n", sum);
  fclose(out);

  return 0;
}
```

3.11 ベクトルの扱い — 配列とポインター

3.11.1 方針

ベクトルはコンパイル前からサイズ (次元) が分かっていて、それが小さければ、1 次元配列で扱うのが簡単であろう。サイズが大きい場合に自動変数 (関数の中で宣言する変数) として宣言するのは、避けるべきである。

サイズが事前に分からないか、分かっていても大きい場合は、malloc() で動的に割りあてるのが良いであろう。

```
double a[1000]; // 1000 を変数にすることは出来ない

int main(void)
{
  double b[1000]; // 大きなデータは難点あり。スタックサイズ不足で Seg fault 等

  double *c;      // おすすめ
  c = malloc(sizeof(double) * 1000); // 1000 を変数にも出来るし、大きくても大丈夫
  if (c == NULL) {
    // エラー処理
  }
  ...
```

3.11.2 prog10.c

ベクトルのサイズ N が事前に分かっていて、小さい場合。

```
/*
 * prog10.c --- 配列
 *   コンパイルするには、たとえば gcc -o prog10 prog10.c
 */

#include <stdio.h>

#define N (3)

/* n 次元のベクトル x, y の内積を計算する */
double inner_product(double x[], double y[], int n)
{
  int i;
  double s;
  s = 0.0;
  for (i = 0; i < n; i++)
    s += x[i] * y[i];
  return s;
}

int main(void)
{
  int i;
  double a[N], b[N];

  printf("二つの %d 次元ベクトルの内積を計算します。\n", N);

  printf("一つ目のベクトル a の入力\n");
  for (i = 0; i < N; i++) {
    printf(" %d 番目の成分=", i + 1); scanf("%lf", &a[i]);
  }

  printf("二つ目のベクトル b の入力\n");
  for (i = 0; i < N; i++) {
    printf(" %d 番目の成分=", i + 1); scanf("%lf", &b[i]);
  }

  printf("内積=%g\n", inner_product(a, b, N));
  return 0;
}
```

3.11.3 prog11.c

上のプログラム prog10.c では、N の定義を書き変えることでベクトルの次元を変えることができるが、(C99 以前の C の規格では) プログラムの実行中にベクトルの次元を選ぶことはできない。そうするには malloc() のような関数を利用して、動的にメモリーを確保する必要がある。ポインターの扱いにも慣れる必要がある。

```
/*
 * prog11.c --- 「動的な配列」をポインターで実現する
 *   コンパイルするには、たとえば gcc -o prog11 prog11.c
 */

#include <stdio.h>
#include <stdlib.h>  /* malloc() の宣言 */

/* n 次元のベクトル x, y の内積を計算する */
```

```
double inner_product(double *x, double *y, int n)
{
  int i;
  double s;
  s = 0.0;
  for (i = 0; i < n; i++)
    s += x[i] * y[i];
  return s;
}

int main(void)
{
  int i, N;
  double *a, *b;

  printf("二つのベクトルの内積を計算します。\n");
  printf("次元を入力してください: "); scanf("%d", &N);
  a = malloc(sizeof(double) * N);
  b = malloc(sizeof(double) * N);
  if (a == NULL || b == NULL) {
    fprintf(stderr, "ベクトルを記憶するメモリーの確保に失敗しました。\n");
    exit(1);
  }

  printf("一つ目のベクトル a の入力\n");
  for (i = 0; i < N; i++) {
    printf(" %d 番目の成分=", i + 1); scanf("%lf", &a[i]);
  }

  printf("二つ目のベクトル b の入力\n");
  for (i = 0; i < N; i++) {
    printf(" %d 番目の成分=", i + 1); scanf("%lf", &b[i]);
  }

  printf("内積=%g\n", inner_product(a, b, N));

  /* a, b のために確保したメモリーを解法する
   * プログラムの最後だからなくても良いが解放の仕方の説明用 */
  free(a); free(b);

  return 0;
}
```

3.11.4 prog11-C99.c

C 言語の新しい規格である C99 では、可変長配列 (variable length array) が導入されたので、次のようなプログラムが書ける。

(でも、この機能は C11 ではオプションになったらしく、C++14 では外されたらしい。それと可変長配列はスタックに確保されるので、あまり大きなサイズは使えない (手元の Mac でやったら 8 MB くらいが上限だった)。大きな行列の実現に使うのは良くないだろう。個人的にはこの機能を使うコードは書かないことにする。)

```
/*
 * prog11-C99.c --- 可変長配列と任意位置での変数宣言
 *   コンパイルするには、たとえば gcc -o prog11-C99 prog11-C99.c
 */

#include <stdio.h>

/* n 次元のベクトル x, y の内積を計算する */
```

```
double inner_product(double *x, double *y, int n)
{
  int i;
  double s;
  s = 0.0;
  for (i = 0; i < n; i++)
    s += x[i] * y[i];
  return s;
}

int main(void)
{
  int i, N;

  printf("二つのベクトルの内積を計算します。\n");
  printf("次元を入力してください: "); scanf("%d", &N);

  double a[N], b[N];

  printf("一つ目のベクトル a の入力\n");
  for (i = 0; i < N; i++) {
    printf(" %d 番目の成分=", i + 1); scanf("%lf", &a[i]);
  }

  printf("二つ目のベクトル b の入力\n");
  for (i = 0; i < N; i++) {
    printf(" %d 番目の成分=", i + 1); scanf("%lf", &b[i]);
  }

  printf("内積=%g\n", inner_product(a, b, N));

  return 0;
}
```

3.11.5 prog11-C++.C

ちなみに、C++ の場合は、関数 malloc() と free() よりも、演算子 new と delete を使うべきである。例えば次のようなプログラムになる。

```
/*
 * prog11-C++.cpp --- 「動的な配列」をポインターで実現する
 *   コンパイルするには、たとえば g++ -o prog11-C++ prog11-C++.cpp
 */

#include <iostream>
#include <cstdlib>
using namespace std;

/* n 次元のベクトル x, y の内積を計算する */
double inner_product(double *x, double *y, int n)
{
  int i;
  double s;
  s = 0.0;
  for (i = 0; i < n; i++)
    s += x[i] * y[i];
  return s;
}

int main(void)
{
```

```
  int i, N;
  double *a, *b;

  cout << "二つのベクトルの内積を計算します。" << endl;
  cout << "次元を入力してください: ";
  cin >> N;
  a = new double [N];
  b = new double [N];
  if (a == NULL || b == NULL) {
    cerr << "ベクトルを記憶するメモリーの確保に失敗しました。" << endl;
    exit(1);
  }

  cout << "一つ目のベクトル a の入力\n";
  for (i = 0; i < N; i++) {
    cout << i+1 << " 番目の成分= ";
    cin >> a[i];
  }

  cout << "二つ目のベクトル b の入力" << endl;
  for (i = 0; i < N; i++) {
    cout << i+1 << " 番目の成分=";
    cin >> b[i];
  }

  cout << "内積=" << inner_product(a, b, N) << endl;

  /* a, b のために確保したメモリーを解法する
   * プログラムの最後だからなくても良いが解放の仕方の説明用 */
  delete [] a;
  delete [] b;

  return 0;
}
```

3.12 行列の扱い — 2 次元配列とポインターのポインター

3.12.1 方針 (私の見解)

行列もコンパイル時にサイズ (行の数、列の数) が分かっていて、それらが小さければ、2 次元配列で扱うのが簡単であろう。

そうでない場合、従来の C ではとても悩ましかった。C99 で導入された可変長配列の機能は、行列のサイズが小さい場合には有効であろう。特に従来からあった「C には Fortran の整合配列の機能がないのでライブラリィが作りにくい」という批判は、可変等配列の機能を利用することで解決出来る。しかし、この可変長配列の機能は、新しい C11 の規格ではオプションにされてしまった。GCC と LLVM では相変わらず使えるようであるが、この機能を使い続けるかどうかは難しい問題である。

一方、数値計算に現れる大規模行列や 2 次元格子上の数値データなど、大規模なデータを扱うには、可変長配列は不適当だと考える。やはり大規模行列や 2 次元格子上の数値データは、`malloc()` 等を使って動的に確保し、ポインターを使って扱うのが適当であろう。

C++ が利用可能な場合、**Eigen**[7] のようなベクトル、行列を扱うためのクラス・ライブラリィが色々利用可能であるので、C++ に乗り換えるのが良いかもしれない。

[7] `http://eigen.tuxfamily.org`

3.12.2 prog12.c

行列のサイズが分かっていて、それが小さい場合は 2 次元配列で簡単に扱える。

```
/*
 * prog12.c --- 2 次元配列で行列を
 */

#include <stdio.h>

void display(double A[2][2])
{
  int i, j;
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++)
      printf("%7.2f ", A[i][j]);
    printf("\n");
  }
}

int main(void)
{
  int i, j;
  double a[2][2] = {{1,2},{3,4}};
  double b[2][2], c[2][2];

  /* b の入力 */
  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++) {
      printf("b[%d][%d]=", i, j);
      scanf("%lf", &b[i][j]);
    }
  /* c:=a+b */
  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
      c[i][j] = a[i][j] + b[i][j];

  /* a,b,c を表示 */
  printf("a=\n"); display(a);
  printf("b=\n"); display(b);
  printf("c=\n"); display(c);

  return 0;
}
```

prog12 の実行結果

```
oyabun% gcc prog12.c
oyabun% ./a.out
b[0][0]=4
b[0][1]=3
b[1][0]=2
b[1][1]=1
a=
   1.00    2.00
   3.00    4.00
b=
   4.00    3.00
   2.00    1.00
c=
   5.00    5.00
   5.00    5.00
oyabun%
```

以下は初めて学ぶときには省略しても構わない。理解するには、各プログラムを印刷してじっくり読み比べることを勧める。

3.12.3 prog12a.c

行列のサイズが分かっていなくても、サイズが小さい場合は、可変長の2次元配列で簡単に扱える。

```
/*
 * prog12a-new.c --- C99 の可変長配列で行列を
 */

#include <stdio.h>

/* 行列を表示する --- 整合配列風の引数渡し */
void display(int m, int n, double A[m][n])
{
  int i, j;
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
      printf("%7.2f ", A[i][j]);
    printf("\n");
  }
}

int main(void)
{
  int i, j, k, m, n;
  /* サイズを決定してから行列を記憶する変数を定義する */
  printf("m= "); scanf("%d", &m);
  n = m;
  double a[m][n], b[m][n], c[m][n]; // 注: m*n が大きいと異常終了する

  /* a の値の設定 */
  k = 0;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      a[i][j] = k++;

  /* b の入力 */
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      printf("b[%d][%d]=", i, j);
      scanf("%lf", &b[i][j]);
    }
  /* c:=a+b */
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      c[i][j] = a[i][j] + b[i][j];

  /* a,b,c を表示 */
  printf("a=\n"); display(m, n, a);
  printf("b=\n"); display(m, n, b);
  printf("c=\n"); display(m, n, c);

  return 0;
}
```

```
$ ./prog12a-new
m= 2
b[0][0]=1
b[0][1]=2
b[1][0]=3
b[1][1]=4
a=
   0.00    1.00
   2.00    3.00
b=
   1.00    2.00
   3.00    4.00
c=
   1.00    3.00
   5.00    7.00
$
```

3.12.4 prog12b.c

プログラムの実行中にサイズを選べるようにするには、行列を double へのポインターのポインターとして表現し、malloc() を使って動的にメモリーを確保することになるが、手順はやや複雑になる。ポインターに慣れていないと難しいかもしれない。

```
/*
 * prog12b.c --- ポインターのポインターで行列を実現
 */

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

/* double へのポインターを vector, vector へのポインターを matrix とする */
typedef double *vector;
typedef vector *matrix;

/* matrix データを表示する */
void display(int m, int n, matrix A)
{
  int i, j;
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
      printf("%7.2f ", A[i][j]);
    printf("\n");
  }
}

/* 行数、列数を指定して matrix を確保（エラー・チェックつき）*/
matrix new_matrix(int m, int n)
{
  int i;
  vector ap;
  matrix a;
  if ((a = malloc(sizeof(vector) * m)) == NULL) {
    return NULL;
  }
```

```
  if ((ap = malloc(sizeof(double) * (m * n))) == NULL) {
    free(a);
    return NULL;
  }
  for (i = 0; i < m; i++)
    a[i] = ap + i * n;
  return a;
}

/* matrix データを解放 */
void free_matrix(matrix a)
{
  free(a[0]);
  free(a);
}

int main(void)
{
  int i, j, m, n;
  matrix a, b, c;

  m = 2; n = 2;
  a = new_matrix(m, n);
  b = new_matrix(m, n);
  c = new_matrix(m, n);
  if (a == NULL || b == NULL || c == NULL) {
    fprintf(stderr, "メモリーの確保に失敗しました。\n");
    exit(1);
  }

  /* a の値の設定 */
  a[0][0] = 1; a[0][1] = 2;
  a[1][0] = 3; a[1][1] = 4;

  /* b の入力 */
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      printf("b[%d][%d]=", i, j);
      scanf("%lf", &b[i][j]);
    }
  /* c:=a+b */
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      c[i][j] = a[i][j] + b[i][j];

  /* a,b,c を表示 */
  printf("a=\n"); display(m, n, a);
  printf("b=\n"); display(m, n, b);
  printf("c=\n"); display(m, n, c);

  return 0;
}
```

4 練習問題

これまで卒研など色々な機会に練習用の問題を出してきた。

4.1 比較的簡単なもの

[1] 自然数 n を入力されたとき、$1+2+\cdots+n$ を計算して、その結果を表示するプログラムを書け[8]。

[2] 実数 a, 自然数 n を入力されたとき、a^n を計算して、その結果を表示するプログラムを書け[9]。

[3] 自然数 k, n を入力されたとき、1^k, 2^k, $\cdots$, n^k を表示するプログラムを書け[10]。

[4] $a_1 = 1$, $a_{n+1} = 3a_n + 2$ $(n \geq 1)$ で定まる数列の最初の 100 項を計算して表示するプログラムを書け (n が大きいときは近似値で構わない — 正確に計算するのは面倒である)。

[5] $a_0 = 1$, $a_1 = 1$, $a_{n+2} = a_{n+1} + a_n$ $(n \geq 0)$ で定まる数列 (フィボナッチ数列) $\{a_n\}$ に対して、$n = 1, 2, \ldots, 100$ の順に a_n と a_n/a_{n-1} を計算して表示せよ。(n が大きいときは近似値で構わない — すべて正確に計算するには、多倍長計算ライブラリィを使うなどの工夫が必要になる。)
[番外] a_n/a_{n-1} の極限は何か？ (簡単な数学の問題)
[それでも尋ねてみるか] どの n まで a_n が正しく計算出来ているか?

[6] n を与えられたとき

$$S_1(n) := \sum_{k=1}^{n} \frac{1}{k}, \quad S_2(n) := \sum_{k=1}^{n} \frac{1}{k^2}$$

を計算するプログラムを書け。計算結果は `double` の精度一杯まで表示せよ。
[番外] $\lim_{n\to\infty} S_2(n) = \pi^2/6$ であることが知られている。点 $(n, \pi^2/6 - S_2(n))$ を両側対数目盛でプロットすることによって、収束の速さを調べよ。

[7] 自然数 n を入力されたとき、$S_n := \sum_{k=0}^{n} \frac{1}{k!}$ を計算して結果を表示するプログラムを書け ($1/k!$ は漸化式で計算するのが簡単)。

[8] 関数 $f\colon [a,b] \to \mathbf{R}$ のグラフを描くプログラムを書け。ただし f を計算する関数をプログラム中に書くものとする。特に

$$f(x) := \begin{cases} x & (0 \leq x \leq 1/2) \\ 1-x & (1/2 \leq x \leq 1) \\ 0 & (\text{それ以外}) \end{cases}$$

の場合に $-0.2 \leq x \leq 1.2$ の範囲のグラフを描け。

[9] (ここまでの応用) 与えられた自然数 n, 実数 x に対して

$$s_n(x) := \sum_{k=1}^{n} \frac{(-1)^{k+1}}{(2k-1)!} x^{2k-1}$$

を計算する関数を書き、$n = 1, 2, 3, \cdots, 10$ に対して $s_n(x)$ $(0 \leq x \leq 2\pi)$ のグラフを描け (「Taylor 級数のプログラミングには漸化式を」)。

---

[8] もちろん $n(n+1)/2$ を計算すれば簡単なわけだが、繰り返しの練習なので、正直に 1 から n まで足すこと。
[9] もちろん冪乗を計算する `pow()` を使えば簡単だが、繰り返しの練習なので…
[10] `for` を二重に用いるプログラムを書こう。

[10] $\mathbf{R}^3$ における極座標をデカルト座標 (直交座標) に変換するコードは簡単で、

```
x = r * sin(theta) * cos(phi);
y = r * sin(theta) * sin(phi);
z = r * cos(theta);
```

とすれば良い。この逆をする (つまりデカルト座標を極座標に変換する) 関数 `d2p()` を作って、動くことをチェックせよ[11]。

```
d2p(x, y, z, &r, &theta, &phi);
```

のようにして呼び出せるようにすること。

[11] 常微分方程式の初期値問題を Euler 法、Runge-Kutta 法などの数値解法で近似的に解けることは重要である。以下の初期値問題を解くプログラムを作れ。結果を可視化せよ。

(1) $x'(t) = x(t)$, $x(0) = 1$.
$x(1) = e$ となるはずだが、計算で得た値と比較せよ。

(2) $x''(t) = -g - \gamma x'(t)$, $x(0) = H$, $x'(0) = 0$. ただし $g > 0$, $\gamma \geq 0$ は定数とする。
これは速度に比例する空気抵抗が存在する場合の自由落下を記述する微分方程式である。
実は (抵抗の大きさが速さの自乗に比例する) $x''(t) = -g - \gamma |x'(t)| x'(t)$ が正しいという説もある。

(3) $\dfrac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$, $x(0) = x_0$, $y(0) = y_0$. ここで a, b, c, d は与えられた定数である。さまざまな (x_0, y_0) に対して解軌道を描け。

[12] 方程式 $\cos x - x = 0$ の実数解を、二分法または Newton 法で計算するプログラムを作れ。

($f(x) := \cos x - x$ は狭義単調減少で、$f(0) = 1 > 0$, $f(1) = \cos 1 - 1 < 0$ であるから、区間 $(0, 1)$ にただ 1 つの実数解を持つことが分かる。)

4.2 ベクトルの扱いの練習

(ここは大規模工事が必要なので…参考程度に)

[5] 長さ n の配列で n 次元のベクトルを表現することができる。n 次元ベクトル $\vec{a} = (a_1, a_2, \cdots, a_n)$ の最大値ノルム

$$\|\vec{a}\|_\infty := \max_{i=1,2,\cdots,n} |a_i|$$

を計算する関数を作ることを目標にする。手始めに、標準入力から入力された 3 次元ベクトルの最大値ノルムを計算する、次のようなプログラムを書き始めた。関数 `maxnorm0()` を完成させよ。

[11] ちょっと短かすぎて悪い名前だが、Decartes 座標 to Polar 座標、のニュアンス。

```
#include <stdio.h>

#define N 3

int main(void)
{
    int i;
    double a[N], maxnorm0();
    for (i = 0; i < N; i++)
        scanf("%lf", &a[i]);
    printf("maxnorm=%g\n", maxnorm0(a));
}

double maxnorm0(....
```

(関数に配列を引数として渡す方法、最大値の計算法など)

[6] (前問の続き) ベクトルの長さ n は、プログラムの実行時に変更したいと考えて、次のようなプログラムを書き始めた。関数 maxnorm() を完成させよ。

```
#include <stdio.h>

#define MAXN 1000

int main(void)
{
    int i, n;
    double a[MAXN], maxnorm();
    printf("input n (<= %d): ", MAXN); scanf("%d", &n);
    if (n > MAXN)
        exit(1);
    for (i = 0; i < n; i++)
        scanf("%lf", &a[i]);
    printf("maxnorm=%g\n", maxnorm(a, n));
}

double maxnorm(....
```

できれば maxnorm() の定義は別のファイルに書いて、分割コンパイルできるようにする。
(再利用可能性の高い関数を書く方法など)

[7] 前問のプログラムでは MAXN の定義を書き換えれば、大きな次元のベクトルに対応するプログラムに変更できるが、それでもプログラムに変更が必要なのは面倒である。また、大きな MAXN をあまり大きくすると、無駄である。そこで、ベクトルの長さ n をプログラムの実行時に指定できるようなプログラムが望ましい。これは次のように malloc() を使うように書き換えれば良い。

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

int main(void)
{
    int i, n;
    double *a, maxnorm();

    printf("input n: "); scanf("%d", &n);
    a = malloc(n * sizeof(double));
    if (a == NULL) {
        fprintf(stderr, "メモリーが確保できません\n");
        exit(1);
    }
    /* 以下、プログラムの字面は前問とまったく同じで OK */
    for (i = 0; i < n; i++)
        scanf("%lf", &a[i]);
    printf("maxnorm=%g\n", maxnorm(a, n));
}

double maxnorm(....
```

(C++ を勉強しているならば、同じことを C++ で書いてみよ。)
長さ n のベクトルを標準入力から二つ読み込んで、それらの和、内積、ユークリッド・ノルムと、二つのベクトルのなす角度を計算するプログラムを作れ。後で、利用できるように、`addvector()`, `dotproduct()`, `norm()`, `angle()` のような関数を作ること。

[8] ある理由から

$$\texttt{a[0]}, \texttt{a[1]}, \texttt{a[2]}, \cdots, \texttt{a[100]}$$

の両端 `a[0]`, `a[100]` を除いた

$$\texttt{a[1]}, \texttt{a[2]}, \cdots, \texttt{a[99]}$$

の絶対値の最大値を知りたいとする。`maxnorm()` を使って求めるには、どうすればよいか。

A 関数へのポインターのプログラム例

「これくらいは覚えよう」というものではないかもしれないけれど…

A.1 比較的ありがちな汎関数の定義

$I(f,a,b) := \int_a^b f(x)\,dx$ は、関数 f を変数にしている、いわゆる汎関数である。というわけで、数値積分公式のプログラミングなどでは、関数を引数にするのが自然である。次は台形公式により数値積分を行う、関数 `trapezoidal()` のプログラム例である。

```
/*
 * ex1.c --- 1/x の [1,2] における定積分を複合台形則で計算する
 */

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

typedef double rrfunction(double);

rrfunction f;
double trapezoidal(rrfunction, double, double, int);

int main()
{
  int N;
  double I, T;

  I = log(2.0);

  printf("台形則\n");
  printf("    N                近似値                      誤差\n");
  for (N = 1; N <= (1 << 16); N *= 2) {
    T = trapezoidal(f, 1.0, 2.0, N);
    printf("%5d\t%25.18f\t%e\n", N, T, I - T);
  }
  return 0;
}

double trapezoidal(rrfunction f, double a, double b, int N)
{
  int j;
  double h, T;
  h = (b - a) / N;
  T = (f(a) + f(b)) / 2;
  for (j = 1; j < N; j++)
    T += f(a + j * h);
  T *= h;
  return T;
}

double f(double x)
{
  return 1 / x;
}
```

A.2 コマンドを登録して呼び出す

関数へのポインターを変数に代入しておいて、その変数を用いて呼び出す、ということが出来ます。

```
#include <stdio.h>

/* 引数を取らない void 型の関数の型を vvfunc と名づける */
typedef void vvfunc(void);
/* vvfunc へのポインタ変数 a を定義 */
vvfunc *a = NULL;

void f(void) { printf("Hello\n"); }

void g(void) { printf("Bye\n"); }

int main()
{
  a = f;
  a();
  a = g;
  a();
  return 0;
}
```

```
bash-3.2$ cc -o prog1 prog1.c
bash-3.2$ ./prog1
Hello
Bye
bash-3.2$
```

関数へのポインターを配列に収めておくことも出来ます。

```
#include <stdio.h>

/* 引数を取らない void 型の関数の型を vvfunc と名づける */
typedef void vvfunc(void);

void f(void) { printf("Hello\n"); }

void g(void) { printf("Bye\n"); }

int n = 0;
vvfunc *commands[10];

void register_command(vvfunc *F)
{
  commands[n++] = F;
}

void all_commands(void)
{
  int i;
  for (i = 0; i < n; i++)
    commands[i]();
}

int main()
{
  register_command(f);
  register_command(g);
  all_commands();
  return 0;
}
```

```
oyabun% cc -o prog2 prog2.c
oyabun% ./prog2
Hello
Bye
oyabun%
```

工夫すると、個々の関数に何かキー (ここでは文字) を対応させておいて、そのキーで関数を呼び出すことが出来ます。ここまで出来ると色々応用が効きます。

```
#include <stdio.h>

/* 引数を取らない void 型の関数の型を vvfunc と名づける */
typedef void vvfunc(void);

void f(void) { printf("Hello\n"); }

void g(void) { printf("Bye\n"); }

/* ------------------------------------------------- */
int n = 0;
vvfunc *commands[10];
char keys[10];

void register_command(vvfunc *F, char c)
{
  keys[n] = c;
  commands[n++] = F;
}

void command(char c)
{
  int i;
  for (i = 0; i < n; i++)
    if (c == keys[i]) {
      commands[i]();
      return;
    }
}
/* ------------------------------------------------- */

int main()
{
  register_command(f, 'H');
  register_command(g, 'B');
  command('H');
  command('B');
  return 0;
}
```

```
oyabun% cc -o prog3 prog3.c
oyabun% ./prog3
Hello
Bye
oyabun%
```