プログラミング AI, プロジェクタ資料
担当: 元木達也 (知能情報)

プログラミング AI

(知能情報システムプログラム2年)

(協創経営プログラム　　3年)

(創生学部知能情報システム領域学習科目パッケージ2年)

0 ガイダンス

0–1 受講に当っての留意事項

旧カリキュラムにおける位置付け

- 平成28年度以前に入学した情報工学科受講生については、(合格したら)「プログラミング実習I」に読み替えられる。

必要な予備知識

- 1年次第3~4タームの「プログラミング基礎I, II」を既に履修して(ある程度の理解をして)いることを前提に話を進める。

授業の進め方

- 講義と演習／実習を交互に行う。
  ⇨ 基本的には、1限は講義、2限は演習／実習。

- Cプログラムの書き方の詳細は参考書等に書かれているので、授業では細かい話はしない。

⇨

授業に出席するだけではこの授業の単位を取れないことを自覚して下さい。
⇨ 予習，質問

0–2 達成目標

- C言語を自在に使いこなせるようになる。

例えば、
必要に応じてデータ構造を定義できる、
動的データ構造も扱える、など。

- UNIXプログラミング環境に慣れ、大規模なプログラムを効率的に開発するための考え方を理解する。

例えば、
make, ... など。

知能情報システムプログラムにおける学習・教育目標との対応:

対応	プログラムの到達目標
	（１）知識・理解
	・・・・・・・・・・・・・・・・・・・・・・・・・・・・
○	c) コンピュータのソフトウェアに関する基礎的知識を修得する。
	・・・・・・・・・・・・・・・・・・・・・・・・・・・・
	（２）当該分野固有の能力
	・・・・・・・・・・・・・・・・・・・・・・・・・・・・
○	c) プログラム等の要求条件を理解し、 プログラム設計等の作業スケジュールを立て、 プログラム作成等を計画通りに実行できる。
	（３）汎用的能力
	・・・・・・・・・・・・・・・・・・・・・・・・・・・・
	（４）態度・姿勢
	・・・・・・・・・・・・・・・・・・・・・・・・・・・・

0–3 教科書、参考書

教科書：

講義ノート等をpdfの形でWebに配置

⇨ 各自でdownloadを行い必要な箇所を印刷

参考書：

⇨ 講義ノート参照

- A.ケリー&I.ポール「CのABC (上)」(1993年,アジソン・ウェスレイ/星雲社, 2718円＋税)
- A.ケリー&I.ポール「CのABC (下)」(1993年,アジソン・ウェスレイ/星雲社, 1942円＋税)
- P.S.Wang「ANSI C & UNIX 上 下」(1993~ 4年,共立出版)
- 浦&原田(編)「C入門」(1994年,培風館)
- B.W.カーニハン&D.M.リッチー「プログラミング言語C 第2版」(1989年,共立出版)

0–4 授業予定

	1限	2限
1回	●処理の選択と繰り返し…プログラム内部の変数の状態遷移を基に処理手順を構築する例題； ●関数(その1)…関数定義, どの様にコンパイルが進むか, 再帰	実習課題1: 変数の状態遷移を理解した上で、しっかりした構造のCプログラムを作る課題に取り組む。 最後は、LaTeXでレポートを完成させる。
2回	●関数(その2)…関数パラメータの受渡し,関数呼出しの実装,再帰計算vs.反復計算； ●モジュール化について…モジュール化, 静的外部変数等を利用した関数以外のモジュール	
3回	●構造体,共用体,typedef； ●動的データ構造…自己参照的構造体	実習課題2: モジュール化、配列の使い方に慣れるための課題に取り組む。 最後は、LaTeXでレポートを完成させる。
4回	●動的データ構造…線形連結リスト, 2分木	

<table>
<tr><td>5回</td><td>●2分木,push-downスタック,待ち行列
...配列を用いた2分木の実装, push-downスタックの実装</td><td rowspan="2">実習課題3: 必要に応じてデータ構造を定義したり、動的データ構造を扱ったりするための能力を養うための課題に取り組む。
最後は、LaTeXでレポートを完成させる。</td></tr>
<tr><td>6回</td><td>●UNIXプログラミング環境...プログラムの計算時間の測定</td></tr>
<tr><td>7回</td><td>●UNIXプログラミング環境...Makefileを用いた分割コンパイル, ライブラリファイルの作成・利用;
●プリプロセッサ...ヘッダファイルの構成・利用, マクロ, 引数付きマクロ</td><td rowspan="2">実習課題4: Makefileを用いたプログラムの保守・管理を経験するための課題に取り組む。
最後は、LaTeXでレポートを完成させる。</td></tr>
<tr><td>8回</td><td>ターム末試験</td></tr>
</table>

以下、第 1~4 節では C 言語の復習も兼ねて、アルゴリズムを如何に構築していくか に重点をおいて説明する。

$\Longrightarrow$ プログラムを説明する際の参考にもなります。

1 復習 Cの基本構文

1–1 自習 基本的なプログラム例

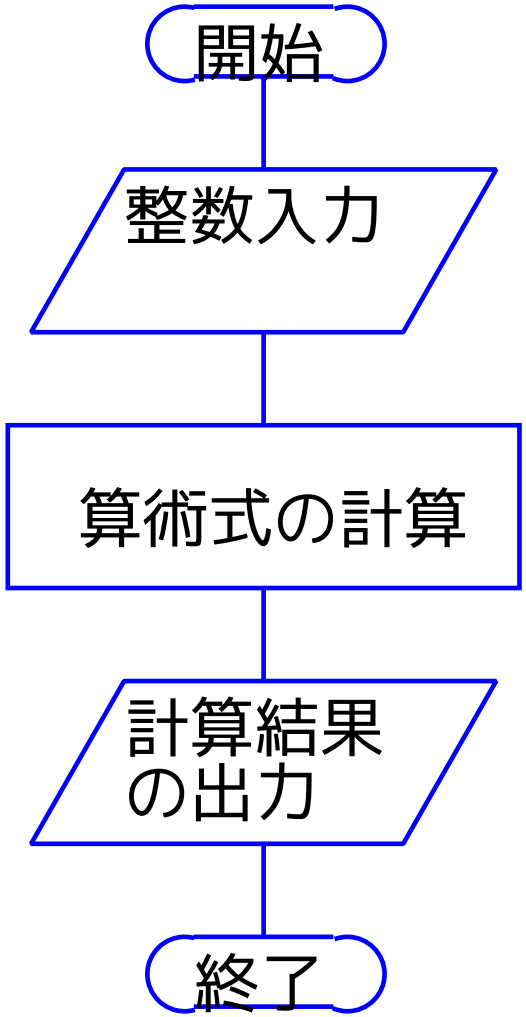
ここでは、

① 整数データの入力,

② それを基に算術式計算,

③ 計算結果の出力

という単純な処理の流れがC言語でどう表されるか例示する。



例題1．1 (四則演算)　2つの整数データを読み込み、それらの 和, 差, 積, 商, 除算の際の余り を出力するCプログラムを作成せよ。

この処理のためには、

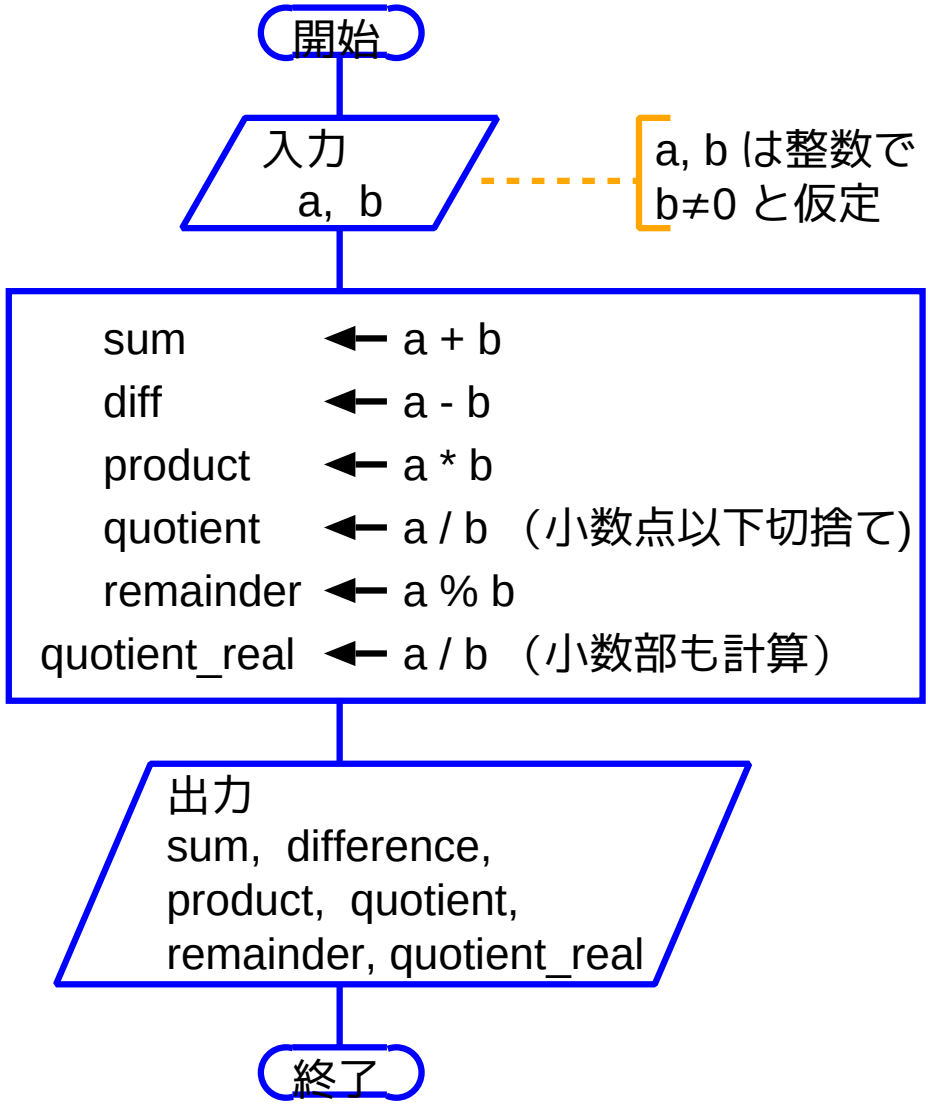
読み込むデータを格納する場所が必要

$\Longrightarrow$ a, b という名前の記憶領域

計算した結果を格納する領域も必要

$\Longrightarrow$ sum, diff, product, quotient, remainder, quotient_real という名前の記憶領域

⇨ コンピュータが行うべき処理は右図の通り。



```
[motoki@x205a]$  nl fundamentals-arith.c   Enter
     1  /* 2つの整数データを変数 a と b に読み込み、それらの和，*/
     2  /* 差，積，商，除算の際の余りを出力するCプログラム       */

     3  #include <stdio.h>

     4  int main(void)
     5  {
     6    int    a, b, sum, diff, product, quotient, remainder;
     7    double quotient_real;

     8    scanf("%d%d", &a, &b);

     9    sum       = a+b;  /* 和 */
    10    diff      = a-b;  /* 差 */
    11    product   = a*b;  /* 積 */
    12    quotient = a/b;  /* 商 */
```

```
13      remainder= a%b;   /* 除算の際の余り */
14      quotient_real = (double)a / (double)b;

15      printf("\nInput data: %d, %d\n\n"
16             "Sum:          %d\n"
17             "Difference: %d\n"
18             "Product:    %d\n"
19             "Quotient:   %d\n"
20             "Remainder:  %d\n"
21             "Quotient (over real): %g\n",
22             a, b, sum, diff, product, quotient, remainder,
23             quotient_real);

24      return 0;
25  }
```

```
[motoki@x205a]$  gcc fundamentals-arith.c   Enter
[motoki@x205a]$  ./a.out   Enter
```

11 3 Enter

```
Input data: 11, 3

Sum:        14
Difference: 8
Product:    33
Quotient:   3
Remainder:  2
Quotient (over real): 3.66667
[motoki@x205a]$
```

注目点：

- 式のそれぞれがデータ型をもつ。

- C言語においては、”=” は「代入」という副作用を持った演算子として扱われる。

  ⇨ 変数 = 式　という形のものは代入式と呼ばれ、
  値を持つ。

- `scanf( )` や `printf( )` もC言語の構文ではなく関数呼び出し。
  従って、関数値を持つ式として扱われる。

- 式の後にセミコロン(`;`)を付けたものが文。

例題1. 2 (円錐の体積；float型, double型, マクロ名)　2 つの実数データ r, h を読み込み、

　　底面の半径が r、高さが h の円錐の体積

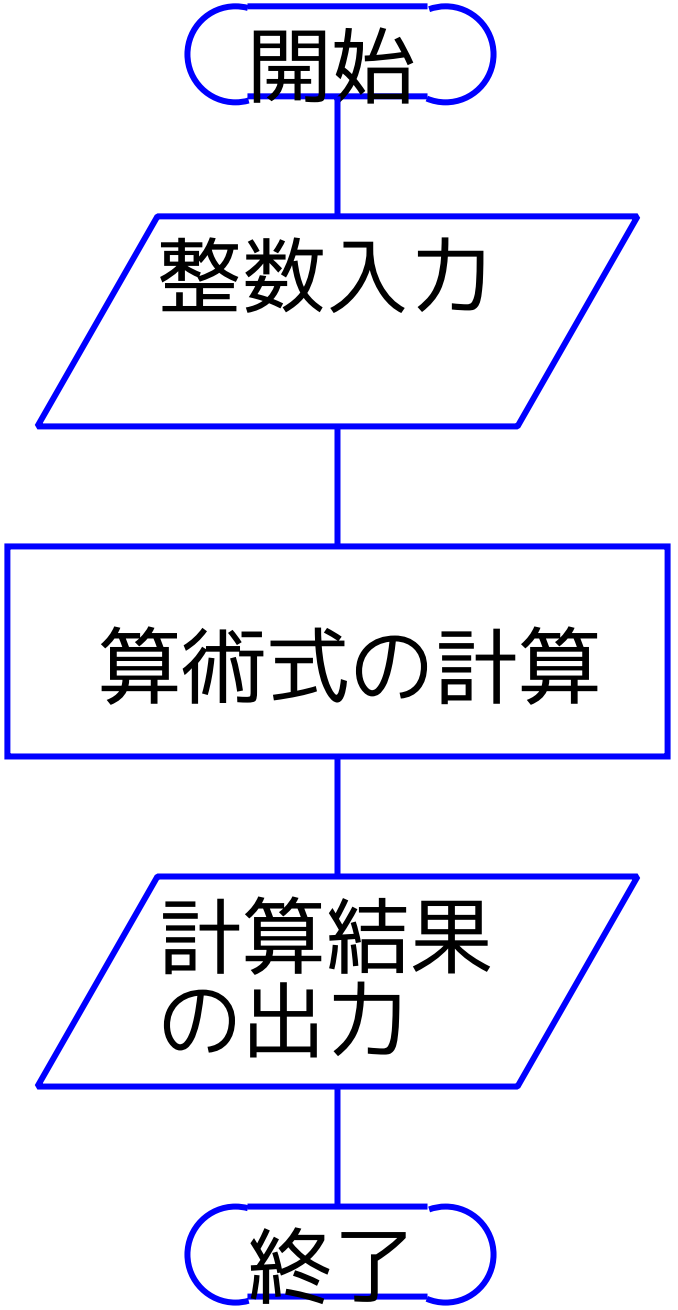
を出力するCプログラムを作成せよ。

(考え方)
処理の流れは例題1.1 で考えた右図と同じである。

違いは、
計算対象が整数データではなく実数データであるということと、
計算式が

$$\text{体積}=\frac{\pi \mathbf{r^2 h}}{\mathbf{3}}$$

ということだけである。



(プログラミング)

実数データを表すためのデータ型として、C言語では

- float型,
- double型,
- long double型

(浮動小数点数型)

の3つが用意されている。

このうち、良く使われるのは float型 と double型 の2つ

⇨ float型 と double型で処理したプログラムをそれぞれ例示する。

double型で処理するプログラム：

```
[motoki@x205a]$  nl volume-of-cone-double.c   Enter
     1 /* 2つの実数データ r と h を読み込み、                    */
     2 /*       底面の半径が r、高さが h の円錐の体積   */
     3 /* を出力するCプログラム                                 */
     4 /*    ---double型で計算する版---                        */

     5 #include <stdio.h>
     6 #define  PI    (3.1415926535897932)    /* 円周率 */

     7 int main(void)
     8 {
     9   double r, h;

    10   scanf("%lf%lf", &r, &h);
    11   printf("底面の半径が %f, 高さが %f の円錐の体積\n"
    12          "      = %f\n",
```

```
    13          r, h, PI*r*r*h/3.0);

    14    return 0;
    15 }
[motoki@x205a]$  gcc volume-of-cone-double.c  Enter
[motoki@x205a]$  ./a.out  Enter
2.0 5.0  Enter
底面の半径が 2.000000, 高さが 5.000000 の円錐の体積
     = 20.943951
[motoki@x205a]$
```

float型で処理するプログラム：

```
[motoki@x205a]$  nl volume-of-cone-float.c   Enter
     1 /* 2つの実数データ r と h を読み込み、                    */
     2 /*       底面の半径が r、高さが h の円錐の体積   */
     3 /* を出力するCプログラム                                 */
     4 /*    ---float型で計算する版---                          */

     5 #include <stdio.h>
     6 #define  PI     (3.1415926f)    /* 円周率 */

     7 int main(void)
     8 {
     9   float r, h;

    10   scanf("%f%f", &r, &h);
    11   printf("底面の半径が %f, 高さが %f の円錐の体積\n"
    12          "      = %f\n",
```

```
    13          r, h, PI*r*r*h/3.0f);

    14    return 0;
    15 }
[motoki@x205a]$  gcc volume-of-cone-float.c  Enter
[motoki@x205a]$  ./a.out  Enter
2.0 5.0  Enter
底面の半径が 2.000000, 高さが 5.000000 の円錐の体積
     = 20.943950
[motoki@x205a]$
```

注意：

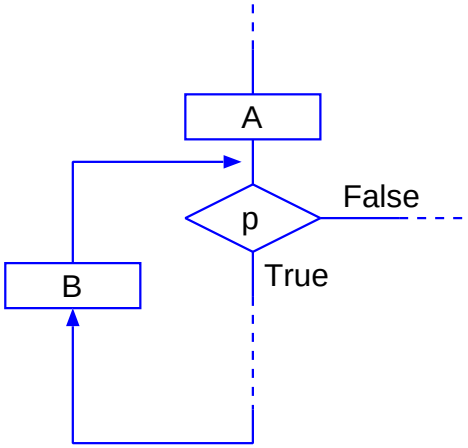
平成13年度実習室においては、ソースプログラムの文字コードがEUC以外だと、コンパイラが文字コードをちゃんと認識しないためにエラーとして処理されていました。

⇨ `printf` がらみの不可解なエラーになったら、
文字コードを別なものに変換してみる。

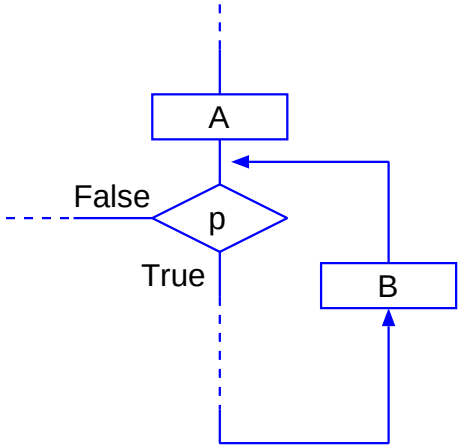
1–2 自習 処理の規則的な繰り返し

繰り返しの箱:

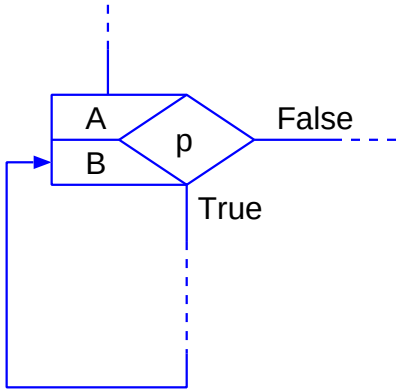
流れ図を用いて処理手順を表した場合、処理の繰り返しを表すために...



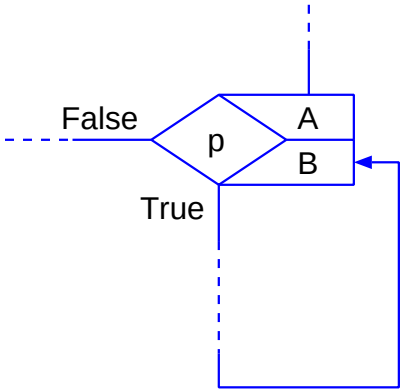
,

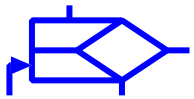


$\Longrightarrow$ 略記法



,



この  という形の箱をここでは繰り返しの箱と呼ぶ。

例題1. 3 (階乗;for文の基本形)　正整数データ k を読み込み、その階乗値 $k!=1\times2\times3\times\cdots\times k$ を double 型実数として求めて出力するCプログラムを作成せよ。

(考え方)　我々が手で計算するとしたら、正整数 k が与えられたとき、次の様に計算を進める。

$$\begin{aligned}
&(\text{step } 1)\ \ 1! &&= 1\\
&(\text{step } 2)\ \ 2! &&= 1!\times2 &&= 1\times2 = 2\\
&(\text{step } 3)\ \ 3! &&= 2!\times3 &&= 2\times3 = 6\\
&&&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
&(\text{step } k)\ \ k! &&= (k-1)!\times k &&= \cdots\cdots\cdots
\end{aligned}$$

この計算をコンピュータに行わせれば良い。では、

どんな変数を用意すれば良いのだろうか?

⇨ 次の点に注目：

この計算の途中で出て来る $1!, 2!, 3!, \ldots, (k-1)!, k!$ の値は、計算のいずれかの時点でどこかの変数に記憶しておく必要がある。

(試行案) 1!, 2!, 3!, ... 各々毎に別の変数を用意して、次の様に計算。

(step 1) 1! の値を保持する変数 ⟵ 1

(step 2) 2! の値を保持する変数 ⟵ 1! の値を保持する変数 ×2

(step 3) 3! の値を保持する変数 ⟵ 2! の値を保持する変数 ×3

(step 4) 4! の値を保持する変数 ⟵ 3! の値を保持する変数 ×4

(step 5) 5! の値を保持する変数 ⟵ 4! の値を保持する変数 ×5

.....

(step k) k! の値を保持する変数 ⟵ (k−1)! の値を保持する変数 ×k

⇨ 致命的な欠点

色々な k の値に対処するために際限のない個数の変数が必要。

⇨ 次の点に注目：

次に $\mathbf{i!}$ の値を計算する時点では、
計算に必要な値は $\mathbf{(i-1)!}$ の値と $\mathbf{i}$ の値だけであり、
$\mathbf{1!,2!,\ldots,(i-2)!}$ の値はそれ以降も必要ない。

⇨ $\mathbf{1!,2!,3!,\ldots}$ を保持するために1つだけ共通のデータ格納領域を用意して、次の様に計算すれば良い。

① 最初はそこに $\mathbf{1!}$ の値を保持する，

② $\mathbf{2!}$ が計算できれば保持されていた $\mathbf{1!}$ の値は捨て代わりに $\mathbf{2!}$ の値を保持する，

③ $\mathbf{3!}$ が計算できれば保持されていた $\mathbf{2!}$ の値は捨て代わりに $\mathbf{3!}$ の値を保持する，

. . . . . .

すなわち、$\boxed{\mathbf{1!,2!,3!,\ldots}\text{の値を保持する変数}}$ を用意して、
次の様に計算すれば良い。

(step 0) 正整数 k を読み込む。

(step 1) 1!,2!,3!,...の値を保持する変数 $\longleftarrow 1$

(この時点で 1!,2!,3!,...の値を保持する変数 $= 1!$)

(step 2) 1!,2!,3!,...の値を保持する変数

$\longleftarrow$ 1!,2!,3!,...の値を保持する変数 $\times 2$

(この時点で 1!,2!,3!,...の値を保持する変数 $= 2!$)

(step 3) 1!,2!,3!,...の値を保持する変数

$\longleftarrow$ 1!,2!,3!,...の値を保持する変数 $\times 3$

(この時点で 1!,2!,3!,...の値を保持する変数 $= 3!$)

.....

(この時点で 1!,2!,3!,...の値を保持する変数 $= (k-1)!$)

(step k) 1!,2!,3!,...の値を保持する変数

$\longleftarrow$ 1!,2!,3!,...の値を保持する変数 $\times k$

(この時点で 1!,2!,3!,...の値を保持する変数 $= k!$)

(step k+1) 1!,2!,3!,...の値を保持する変数の値を出力

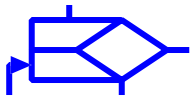
(プログラミング)

上記の手順(step 2)~ (step k)は、

1!,2!,3!,...の値を保持する変数 ⟵ 1!,2!,3!,...の値を保持する変数 × i

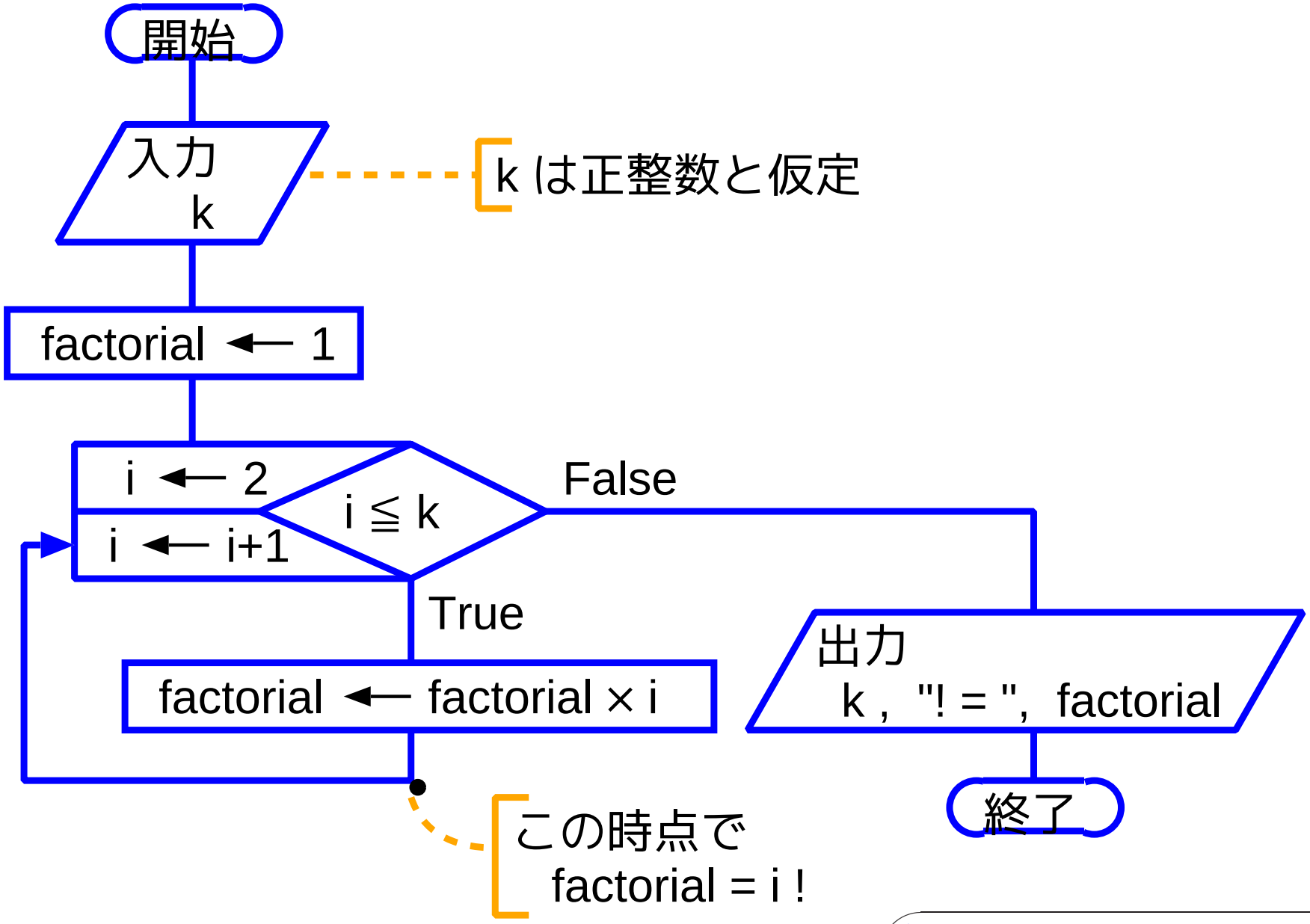
という処理を i=2, 3, 4, ..., k に対して順に行っているだけ。

⇨ 流れ図においてはこの(step 2)~ (step k)の部分を

繰り返しの箱  を用いて表すことができる。

読み込んだ正整数を格納するために k という名前の変数を、
1!,2!,3!,...の値を保持するために factorial という名前の変数を、
i=2~ k の値を記憶するために i という名前の変数を
用意することにすれば、...

⇨ 行うべき処理は次の流れ図の様に書き表すことができる。



補足：
“**factorial**” は階乗という意味の英単語である。
⟹ 階乗と何の関係のない計算に “**factorial**” という名前の変数を用いてはならない。

実際には、整数型 **32** ビットでは **13!** は記憶できないので、上記プログラムのように **int** 型で階乗値を正確に求めたい場合は、入力変数 **k** は **12** 以下でなければならない。

```
[motoki@x205a]$  nl factorial-double.c   Enter
     1 /* 正整数データを読み込み、その階乗値を          */
     2 /* double型実数として求めて出力するCプログラム */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   int    k, i;
     7   double factorial;

     8   printf("何の階乗を求めますか?: ");
     9   scanf("%d", &k);

    10   factorial = 1.0;
    11   for (i=2; i<=k; ++i){
    12     factorial *= (double) i; /* この時点でfactorial=i!
```

```
13   }

14   printf("%d! = %21.16g\n", k, factorial);
15   return 0;
16 }
```

```
[motoki@x205a]$  gcc factorial-double.c  Enter
[motoki@x205a]$  ./a.out  Enter
何の階乗を求めますか?:  53  Enter
53! = 4.27488328406002 5e+69
[motoki@x205a]$
```

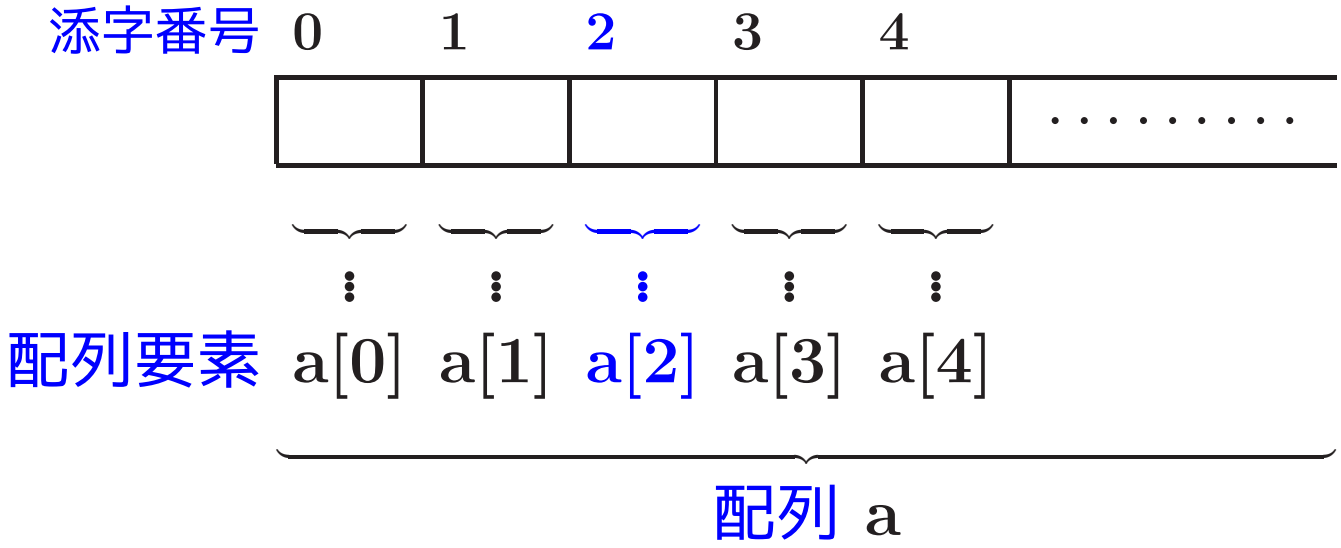
1–3 自習 一次元配列

同じデータ型の領域を連続的に並べたものを一般に配列と呼び、その中の個々のデータ領域を配列要素と呼ぶ。

配列を使えば 大量の同種のデータを規則的に並べて格納し、その中の各々のデータに対して同じ処理を繰り返すことが簡単にできるので、大抵のプログラミング言語で配列が使えるようになっている。

特にC言語においては、

配列の先頭に位置する要素から順に 0, 1, 2, 3, ... という添字番号が各々の配列要素に割り振られ、配列 a の中の添字番号が k の配列要素は a[k] と表される。



例題1. 4 (平均と分散)　50個の実数データ $x_0, x_1, x_2, \cdots, x_{49}$ を読み込み、それらの平均 μ と分散 V を定義式

$$\mu = \frac{1}{50}(x_0 + x_1 + x_2 + \cdots + x_{49})$$

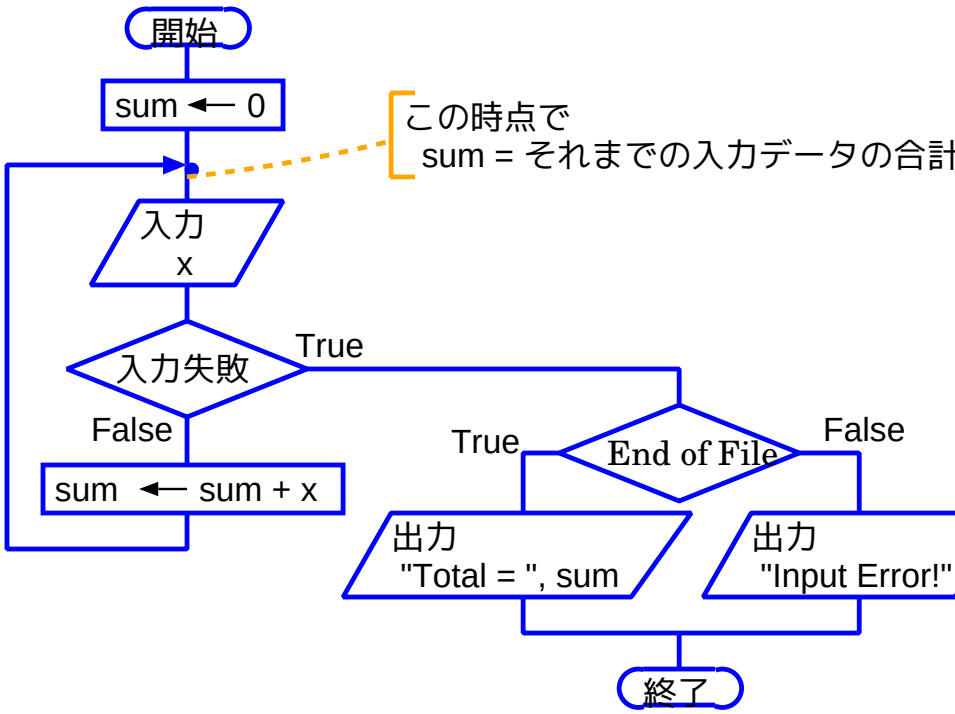
$$V = \frac{1}{50}\sum_{i=0}^{49}(x_i - \mu)^2$$

に従って求めて出力するCプログラムを作成せよ。

(考え方)　平均 μ を計算するだけなら、読み込んだデータを保持する変数を1個だけ用意し、

- そこへのデータ読み込みと、
- 読み込んだデータを別の累算値を保持する変数に加える作業

を交互に繰り返せばよい。



$$\mu = \frac{1}{50}(\mathbf{x_0} + \mathbf{x_1} + \mathbf{x_2} + \cdots + \mathbf{x_{49}})$$
$$\mathbf{V} = \frac{1}{50}\sum_{\mathbf{i=0}}^{\mathbf{49}}(\mathbf{x_i} - \mu)^{\mathbf{2}}$$

しかし、
指定された式に従って分散 $\mathbf{V}$ を計算するなら 、
$\mathbf{V}$ の計算には平均 μ の計算結果が必要になるので、分散 $\mathbf{V}$ の計算で指定された累算をデータの読み込みと並行して行うわけにはいかない。

⇨ 読み込んだ50個の実数データ $\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{49}}$ は
全て保持しておく必要がある。

⇨ これらのデータ保持に配列を用いる。

(プログラミング)

```
[motoki@x205a]$  nl average-variance.c   Enter
```

(注釈は省略)

```
 6  #include <stdio.h>

 7  int main(void)
 8  {
 9    int      i;
10    double   x[50], ave, var;

11    ave = 0.0;
12    for (i=0; i<50; ++i) {
13      scanf("%lf", &x[i]);
14      ave += x[i];
15    }
16    ave /= 50.0;
```

```
17      var = 0.0;
18      for (i=0; i<50; ++i)
19        var += (x[i]-ave)*(x[i]-ave);
20      var /= 50.0;

21      printf("\nInput data:\n");
22      for (i=0; i<50; i+=5)
23        printf("%14.5e%14.5e%14.5e%14.5e%14.5e\n",
24               x[i], x[i+1], x[i+2], x[i+3], x[i+4]);
25      printf("\nAverage  = %14.6g\n"
26             "Variance = %14.6g\n", ave, var);
27      return 0;
28  }
```

```
[motoki@x205a]$  cat fundamentals-ave-var.data   Enter
1.0000  1.0001  1.0002  1.0003  1.0004
1.0005  1.0006  1.0007  1.0008  1.0009
```

```
1.0010  1.0011  1.0012  1.0013  1.0014
1.0015  1.0016  1.0017  1.0018  1.0019
1.0020  1.0021  1.0022  1.0023  1.0024
1.0025  1.0026  1.0027  1.0028  1.0029
1.0030  1.0031  1.0032  1.0033  1.0034
1.0035  1.0036  1.0037  1.0038  1.0039
1.0040  1.0041  1.0042  1.0043  1.0044
1.0045  1.0046  1.0047  1.0048  1.0049
[motoki@x205a]$  gcc fundamentals-ave-var.c  Enter
[motoki@x205a]$  ./a.out < fundamentals-ave-var.data  Enter

Input data:
   1.00000e+00   1.00010e+00   1.00020e+00   1.00030e+00   1.0
   1.00050e+00   1.00060e+00   1.00070e+00   1.00080e+00   1.0
   1.00100e+00   1.00110e+00   1.00120e+00   1.00130e+00   1.0
   1.00150e+00   1.00160e+00   1.00170e+00   1.00180e+00   1.0
   1.00200e+00   1.00210e+00   1.00220e+00   1.00230e+00   1.0
```

```
   1.00250e+00   1.00260e+00   1.00270e+00   1.00280e+00   1.0
   1.00300e+00   1.00310e+00   1.00320e+00   1.00330e+00   1.0
   1.00350e+00   1.00360e+00   1.00370e+00   1.00380e+00   1.0
   1.00400e+00   1.00410e+00   1.00420e+00   1.00430e+00   1.0
   1.00450e+00   1.00460e+00   1.00470e+00   1.00480e+00   1.0

Average  =         1.00245
Variance =     2.0825e-06
[motoki@x205a]$
```

1–4 付録 基本文法のまとめ

⇨ 軽く目を通しておいて下さい。

少くとも、何処に何が書いてあるか把握しておく。

1–4–1 Cプログラムの構成

Cプログラムの基本形式：

- Cプログラムは次のような形をしている。

プリプロセッサ指令の列
(`#include ...` や `#define ...`)

大域変数の宣言

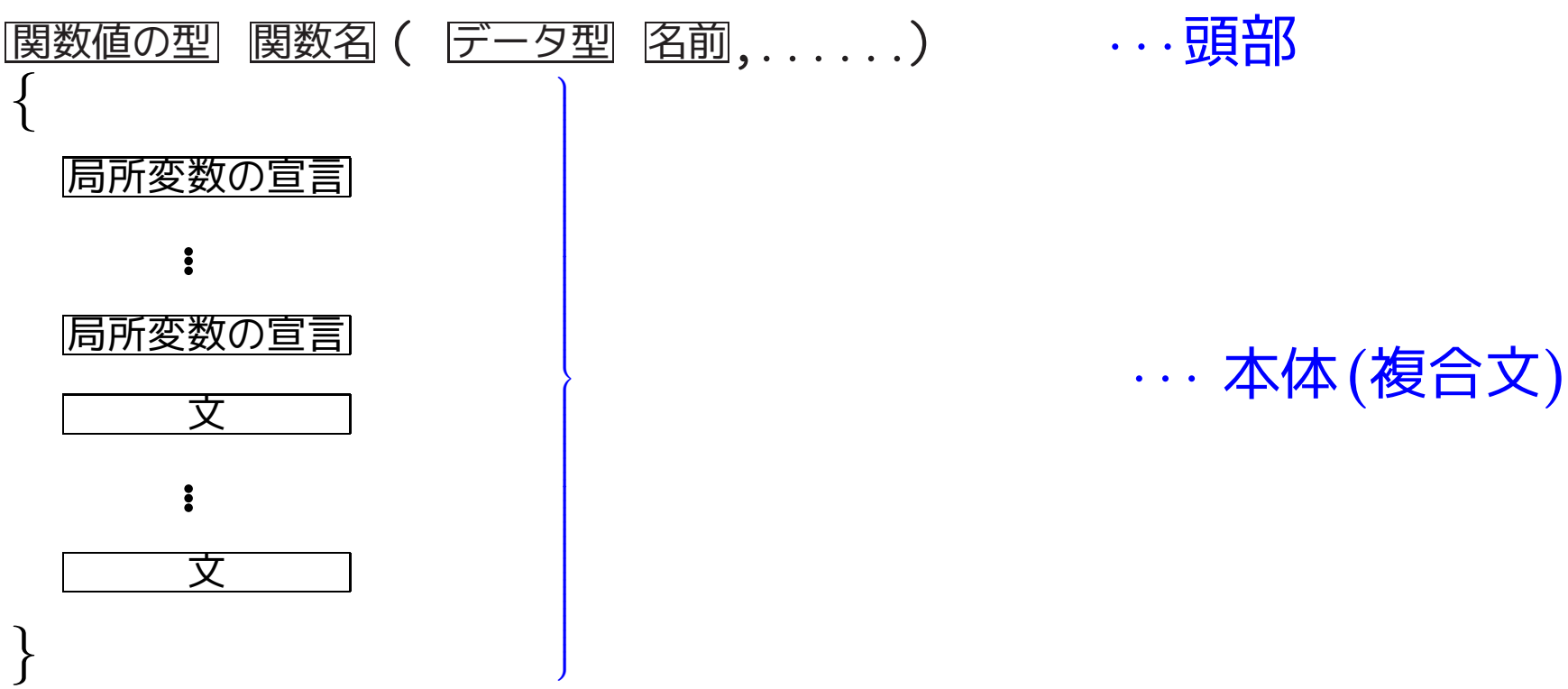
関数定義の列

- プログラム起動の際は `main` という名前の関数から実行が開始される。

- プログラム内の `/*` と `*/` で囲まれた部分は注釈として扱われる。

関数定義の形式：

- Cプログラムの関数定義は次のような形をしている。



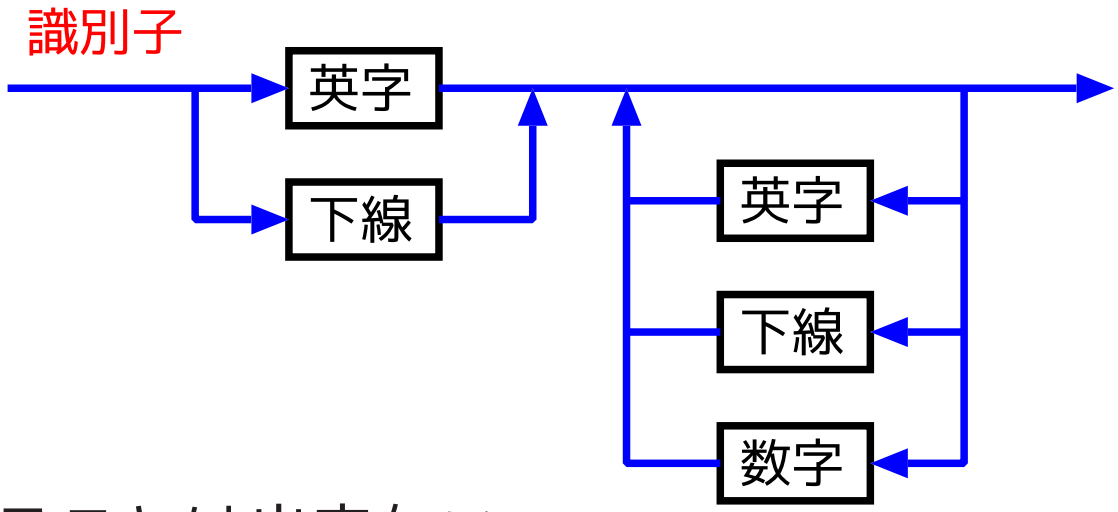
- 関数値の型 の部分は省略可能で、省略すると `int` と見なされる。
- 名前を表す文字列の途中を除いて、どこで改行してもよいし、どこに空白を挿入してもよい。
  ⇨ 字下げ等

1–4–2 宣言、式、代入

変数や関数の名前の付け方：変数や関数(, 配列, ... など)の名前としては、

英字または下線で始まり、
それに英数字または下線が続いた文字の並び

を使うことが出来る。



但し、

- 複数のものに同じ名前を付けることは出来ない。
- 英字の大文字と小文字は区別される。
- プログラムを読み易くするために、変数や関数の役割に応じた名前を

付けることが大切である。

- Cプログラムの中では、次の文字列(キーワードと言う) は特別な役割を果たすので、変数や関数等の名前として使うことは出来ない。

```
auto     double int      struct
break    else   long     switch
case     enum   register typedef
char     extern return   union
const    float  short    unsigned
continue for    signed   void
default  goto   sizeof   volatile
do       if     static   while
```

- ANSI(American National Standards Institute)規格のC言語では、先頭の少なくとも31文字を識別することになっている。
- 下線で始まる識別子はシステムプログラムで使われたものと衝突することがある。

変数の宣言：変数を使う時は、

データ型　変数名 , 変数名 , … , 変数名 ;

という風に宣言する。

- 関数定義の最初に置く。(実行文の前。)
- メモリ領域の確保のため。
- 指定した演算を正しく行うため。

例えば、
整数型の加算と浮動小数点数型の加算では機械語命令コードが違うので、確保したメモリにどんな種類のデータを入れるかは処理系側が知っておかなければならない。

代入文：変数に値をセットしたい場合は、次の様に書く。

変数等 = 式 ;

但し、

- 式 は定数、変数、関数呼出し等を演算子でつないだものである。
- 算術演算子としては次のものがある。

算術演算子	機能
+	加算。但し、単項演算の場合は恒等変換を表す。
-	減算。但し、単項演算の場合は符号反転を表す。
*	乗算。
/	除算。
%	剰余。”a % b” は a を b で割った時の余りを表す。

- 整数定数としては、例えば

  `17` (**10**進), `017` (**8**進), `0x17` (**16**進)

  といった表記のものを使うことが出来る。

- セミコロン (;) を付けると式が文になる。

算術計算の際の自動型変換：

- int型,float型,double型の間では、四則演算 a+b, a-b, a*b, a/b はどれも次の様に行われる。

	b (int)	b (float)	b (double)
a (int)	そのまま演算	floatに揃えて演算	doubleに揃えて...
a (float)	floatに揃えて演算	そのまま演算	doubleに揃えて...
a (double)	doubleに揃えて...	doubleに揃えて...	そのまま演算

- 実数→整数 間の型変換が実際にどう行われるかについては計算機に依存する。 [切捨て、切り上げ、四捨五入のいずれか。]

代入演算子:

- C言語では、代入を表す = は構文の一部ではなく演算子。

  $\Longrightarrow$ a=b+c　は式。

  セミコロンの付いた a=b+c;　は文。

- 代入式は通常の算術式と同様に値を持っている。例えば、代入文

  ```
  a = (b=2) + (c=3);
  ```

  は、次の代入文の列と同等。

  ```
  b=2;
  c=3;
  a = b + c;
  ```

- 代入演算子には、= だけでなく

  ```
  +=   -=   *=   /=   %=   ......
  ```

  というのもある。 一般に、

  変数等 op = 式

  は次の式と同等。

  変数等 = 変数等 op 式

例えば、j *= k+3 は j = j * (k+3) と同等。

代入の際の自動型変換：

- 代入　変数等 = 式　において両辺の型が違えば、式 の値は 変数等 の型に強制的に変換される。

キャスト演算子：

- 明示的に型変換を行うことが出来る。
- 式の値をデータ型という型に変換したければ、次の様に書く。
  （ データ型 ） 式
- キャストは単項演算子。
- 他の単項演算子 (e.g. 符号反転の-,++) と同じ優先順位、結合性 (右から左) を持つ。

例 1. 5 (キャスト演算の優先順位)

`(float) i+3` は `((float)i) + 3` と同等である。

増分演算子と減分演算子:

++変数等 ... 副作用として変数等の値を $+1$ する。
その結果を値とする。

--変数等 ... 副作用として変数等の値を -1 する。
その結果を値とする。

変数等++ ... 変数等の値を式の値とする。
副作用として変数等の値を $+1$ する。

変数等-- ... 変数等の値を式の値とする。
副作用として変数等の値を -1 する。

1–4–3 コンパイラの作業

プログラムのコンパイルと実行： **UNIX/Linux**上においては、**Emacs**等のエディタを使って作られた**C**プログラムをコンパイルするには、一般に、cc や gcc といったコマンドが用いられる。例えば、prog1.c という名前の**C**プログラムが出来ている時、これをコンパイル・実行するには次の様にすればよい。

(例1) `gcc prog1.c` .....(コンパイル)
`./a.out` .....(実行)

(例2) `gcc -o prog1 prog1.c` .....(コンパイル)
`./prog1` .....(実行)

いずれの場合も、コンパイル直後にメッセージが出されたらそれはエラーメッセージで、よく読んでプログラムを修正した上で再度コンパイルする必要がある。

($\Longrightarrow$ **2.2**節を参照)

一般に、cc, gcc といった**C**言語処理系は翻訳の前に前処理を行う。#で始まる行はその前処理で何を行うか指示をしている。

コンパイラの実際の作業手順について：一般に、cc, gcc といったC言語処理系は、実際には次のような手順でコンパイル作業を進める。

(1) 前処理 (ヘッダファイル、すなわち .h で終わるファイルの読み込み、等を行う。)

(2) プログラムを構成する文字の列を字句、すなわち
コンパイルの際に意味のある最小単位
の列に変換する。

補足：
字句には次の６種類がある。

キーワード	···	int, while, ...
識別子	···	変数名, 関数名, ...
定数	···	77, 12.3e+5, 'a', ...
文字列定数	···	"abc", ...
演算子	···	+, -, *, /, %, 関数名の次の括弧, ...
句切り記号	···	( ), {}, ;, ...

(3)
(4)
⋮
} 構文解析、翻訳コード生成、など

プリプロセッサ(前処理を行う部分):

- Cコンパイラの翻訳作業の前にヘッダファイルの読み込み等を行う。

- # で始まる行はプリプロセッサへの指令。

(普通、1カラム目に # を置く。)

例:

```
#include <stdio.h>                    ···(/usr/include/stdio.h)
#include "ファイル名"                  ···(ファイル名は普通 .h で終わる。)
#define  PI  3.14159  ···(マクロ名には普通英大文字を使う。)
```

- 標準のヘッダファイル <stdio.h>, <stdlib.h> , ... の中には関数プロトタイプの宣言等が入っている。

前処理作業の具体例：

前処理はCプログラム中の # で始まる行 (前処理指令) の指示に従って行われる。例えば、

- Cプログラム中に

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

  という行があれば、プログラムのその場所に `/usr/include/stdio.h` というファイルの中身が挿入されたものとして、コンパイル作業が続けられる。

- Cプログラム中に

  ```
  #include "mylib.h"
  ```

  という行があれば、自分で別に作成した `./mylib.h` というファイルの中身がプログラムのその場所に挿入されたものとして、コンパイル作業が続けられる。

- Cプログラム中に

```
#define PI 3.1415926535897932
```

という行があれば、それ以降は (空白等で区切られた) `PI` という文字列は自動的に `3.1415926535897932` という文字列に置き換えられる様になる。

- Cプログラム中に

```
#define square(x) ((x)*(x))
```

という行があれば、それ以降は自動的に `square(a)` という文字列は `((a)*(a))` と置き換えられ、`square(a+b)` という文字列は `((a+b)*(a+b` と置き換えられる様になる。

ヘッダファイルの中身は?:

C 言語においては、入出力を始めとした基本動作を行うために色々な関数が用意され、プログラムの中からそれらの関数を適宜呼び出す様になっている。例えば、printf() や scanf() もこういった関数で、プログラムの中で printf(...); と書くことによってprintf関数の呼び出しを表している。これらの関数は、予めコンパイルされ標準のライブラリの中に蓄えられていて、適切に呼び出されるのを待っている状態にある。ところが、コンパイラはこれらのライブラリ関数がどういう引数を取りどういう型の値を関数値とするのかについての情報を全く持っていないので、これらの情報をコンパイル時にコンパイラに知らせる必要がある。これを行っているのが #include <stdio.h> 等の行である。

すなわち、標準のヘッダファイル<stdio.h>, <stdlib.h>, ... の中にはそれぞれの標準ライブラリ関数がどんな型のデータを引数に取りどんな型の値を返すかの情報をコンパイラに知らせるための文(関数プロトタイプと言う)、などが入っている。

#defineで始まる行について：

- マクロ定義という。
- これを用いれば、プログラムのパラメータとなる定数、物理定数などに記号の名前 (マクロ名 または 記号定数という) を付け、以降のプログラム内で自由に使うことが出来る。
- 習慣的に、マクロ名には英大文字列を使う。
- マクロ定義によってパラメータ付きの任意の文字列に名前を付けることが出来る。 例えば、

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

但し、この場合は `max(i++,j++)` とすると駄目。

- マクロを定義する場合、マクロ名の右側の置換テキストは全体を丸括弧で囲むのが無難。 何故なら、例えば

```
#define square(x) (x)*(x)
```

とマクロ定義した場合は、

```
4/square(2) ⇨ 4/(2)*(2)
```

と展開されてしまう。

- パラメータ付きマクロを定義する場合、マクロ名の右側の置換テキストにおいては各パラメータを丸括弧で囲むのが無難。 何故なら、例えば

```
#define square(x) (x*x)
```

とマクロとマクロ定義した場合は、

```
square(z+1) ⇨ (z+1*z+1)
```

と展開されてしまう。

#includeで始まる行について：

- `#include "  .h  "` の形の指令

  ⇨ 自分で用意したインクルードファイル./ `  .h` の中身を挿入

- `#include <  .h  >` の形の指令

  ⇨ 標準に用意されたインクルードファイル/usr/include/ `  .h` の中身を挿入

- ファイルの先頭に置くのが普通。

  (⇨ 挿入指示のファイルをヘッダファイル ともいう。)

- ヘッダファイルの拡張子は習慣的に `.h`

- ヘッダファイルの中に `#include` や `#define` で始まる行があってもよい。

- 標準のヘッダファイルの中には、ライブラリ関数がどんな型のデータを引数に取りどんな型の値を返すかの情報をコンパイラに知らせるための文、などが入っている。

標準ライブラリ：

- C言語では、入出力は関数の呼出しによって行うので言語自体は軽くなっている。 (printf もscanf もライブラリ関数。)

- ライブラリ関数は豊富に用意されている。

- C言語のコンパイラ本体は、各ライブラリ関数のプロトタイプを予め知っている訳ではない。

  ⇨ 必要なプロトタイプ宣言はプログラマが責任を持って行う。
  (#include 等を使う。)

1–4–4 字句要素、演算子

字句の認識：実際のコンパイル作業はプログラムを構成する字句を認識することから始まる。例えば、プログラム

```
 1  /* ２つの整数を読み込み、和を出力 */

 2  #include <stdio.h>

 3  int main(void)
 4  {
 5    int   a, b, sum;

 6    printf("Input two integers:  ");
 7    sacnf("%d%d", &a, &b);
 8    sum = a + b;
 9    printf("%d + %d = %d\n", a, b, sum);

10    return 0;
11  }
```

の場合、コンパイラは次の表に示される様な字句を認識する。

	キーワード	識別子	定数	文字列定数	演算子	句切り記号
1行目						注釈部
2行目	プリプロセッサ指令					
3行目		main			()	
4行目						{
5行目	int	a b sum				, ;
6行目		printf		"Input two ··· "	()	;
7行目		scanf a b		"%d%d"	() &	, ;
8行目		sum a b			= +	;
9行目		printf a b sum		"%d + %d ··· "	()	, ;
10行目						}

注釈：

- /* と */ で囲まれた部分は注釈として扱われる。
- 注釈は空白類(空白, **Tab**, 改行)と同等に扱われる。
- 注釈を目立たせたい時には、例えば次の様な書き方をする。

```
/*
 *
 *
 */
```

注　釈

```
/**********************************/
/*                                */
/*      注       釈               */
/*                                */
/**********************************/
```

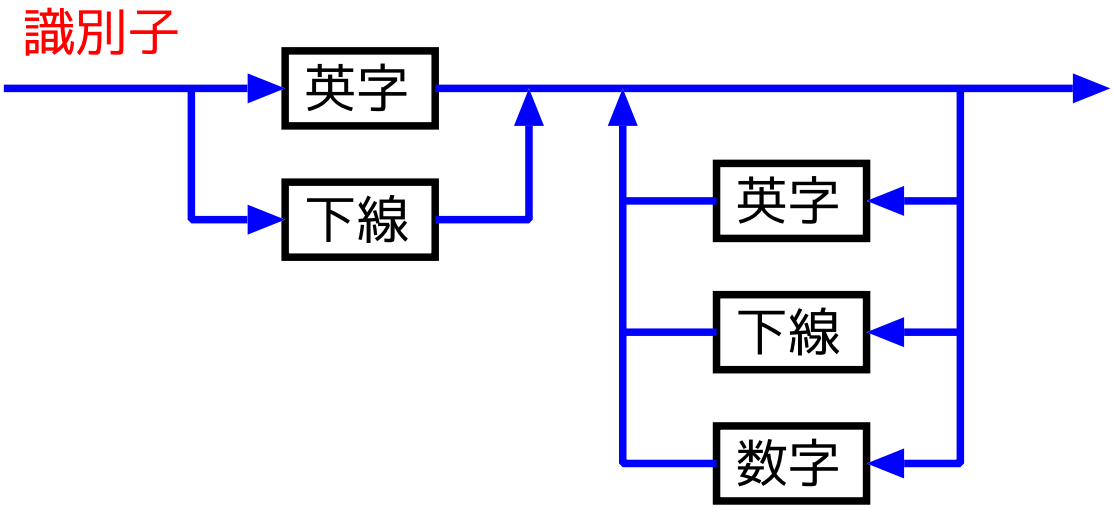
キーワード：

- **Pascal**では「予約語」と呼ばれていた。
- 次のようなキーワードがある。

```
auto     double int      struct
break    else   long     switch
case     enum   register typedef
char     extern return   union
const    float  short    unsigned
continue for    signed   void
default  goto   sizeof   volatile
do       if     static   while
```

識別子：

- 変数, 配列, 関数, ... などに一意的な名前を付けるのに使われる。



- 意味のある名前を選ぶ。
- **ANSI C**では、先頭の少なくとも**31**文字を識別することになっている。
- 下線で始まる識別子はシステムプログラムで使われたものと衝突することがある。

定数：

- 定数
  - 整数定数 … 17 (10進), 017 (8進), 0x17 (16進)
    (注: -17 は定数式)
  - 浮動小数点定数 … 123.4, 123e+5
  - 文字定数 … 'a', 'b', '\n' (改行文字), ' '
  - 列挙定数 … ⟹ 第7章で

文字列定数 (文字リテラル)：

- 文字の列を２重引用符で囲むと文字列定数になる。
- 例えば、次のようなものがある。

```
"abc"
""
"a string with double quote \" within"
"a single backslash \\ is in this string"
"abc"   "def"                          ⟵ "abcdef" と同じ。
```

演算子：

- 次のような種類の演算子がある。

算術演算子	···	+ (単項,恒等変換), - (単項,符号反転), + (加算), - (減算), * (乗算), / (除算), % (剰余)
代入演算子	···	=, +=, -=, *=, /=
増分演算子	···	++変数等, 変数等++
減分演算子	···	--変数等, 変数等--
関係演算子	···	<, <=, >, >=, ==, !=
論理演算子	···	&&, ||, !
条件演算子	···	式 ? 式 : 式
間接演算子	···	* ポインタ
番地演算子	···	& 変数等
sizeof演算子	···	sizeof(オブジェクト)
キャスト演算子	···	(データ型) 式
関数の引数をくくる丸括弧		

.....

- 例えば、次のプログラムでは下線部が演算子。

```
/* ３つの入力データの最大値(その３) */
#include <stdio.h>

main()
{
  int  a, b, c, max;

  scanf("%d%d%d",  &a, &b,  &c);

  if (b<=a &&  c<=a)
    max = a;
  else if (c<=b)
    max = b;
  else
    max = c;

  printf("max = %d\n", max );
  return 0;
```

```
}
```

演算の優先順位と結合性:

{⟹完全な表は11.4節}

優先順位高 ↑

演算子	結合性
関数の引数をくくる丸括弧	左から右
+(単項) -(単項) ++ -- sizeof( ) キャスト	右から左
* / %	左から右
+ -	左から右
= += -= *= /=	右から左

例1. 6 (優先順位)

1+2*3 は 1+(2*3) の意。

-a*b-c は ((-a)*b)-c の意。

((-(a*b))-c ではない。)

例1. 7 (結合性)

a=b=c は a=(b=c) の意。

句切り記号：

- 丸括弧、波括弧、コンマ、セミコロン、など。
- 例えば、次のプログラムでは下線部が句切り記号。

```
/* ３つの入力データの最大値（その３） */
#include <stdio.h>
int main(void)
{
  int  a, b,  c, max;
  scanf("%d%d%d", &a , &b, &c) ;
  if (b<=a && c<=a)
    max = a;
  else if (c<=b)
    max = b;
  else
    max = c;
```

```
    printf("max = %d\n", max) ;
    return 0;
}
```

1–4–5 書式付き出力 —printf—

関数printfの構文：

- 関数printfのデータ型は次の通り。

```
int printf( 書式 , 式 , 式 , ... );
```

- 書式は「(データ)変換指定」や出力したい文字を並べて、2重引用符で囲むことによって指示する。

- 書式に続く 式 は、出力データを表す式である。

- 変換指定は出力値の表示方法を指定したもので、その一般形は

  %[フラグ][最大フィールド幅][.精度]
  [型限定子]変換指定子

  但し、[...] の部分はそれぞれオプションで、省略可。

  となっている。

関数printfの実行の流れ：

- 書式 に書かれた順に出力が為される。

- 「変換指定」以外の部分はそのまま出力される。

  「変換指定」の部分は、第2引数以降から取り出された式の値を変換指定に従って文字列に置き換えて出力される。

  [書式と出力データの列を見比べながら処理が進む。]

- 出力が無事終了した場合は出力した文字の個数が関数値となり、エラーが発生した場合は負の値が関数値になる。

関数printfにおける変換指定子：

次のような変換指定子が用意されている。

変換指定子	説 明
d	指定されたデータがint型の内部表現形式に従っているものと見て、それを10進表記に変換して出力する。
i	
u	指定されたデータがunsigned int型の内部表現形式に従っているものと見て、それを10進表記に変換して出力する。
o	指定されたデータがunsigned int型の内部表現形式に従っているものと見て、それを8進表記に変換して出力する。
x	指定されたデータがunsigned int型の内部表現形式に従っているものと見て、それを16進表記に変換して出力する。(xだとa~ fが、Xだと A~ Fが16進数字として使われる。)
X	
c	int型 (またはchar型) データの下位8ビットを文字コードに持つ文字を出力する。

変換指定子	説 明
f	指定されたデータがdouble型の内部表現形式に従っているものと見て、それを次の形式の10進小数表記(指数部無し) に変換して出力する。 [−] 数字列 . 数字列 出力指定された式がfloat型の場合は、その値がdouble型に変換された後にprintf関数が呼び出される。
e E	指定されたデータがdouble型の内部表現形式に従っているものと見て、それぞれ次の形式の指数部付きの浮動小数点表記に変換して出力する。 [−] 0以外の数字 . 数字列 e ± 2桁以上の数字列 [−] 0以外の数字 . 数字列 E ± 2桁以上の数字列 出力指定された式がfloat型の場合は、その値がdouble型に変換された後にprintf関数が呼び出される。
g G	f変換とe(またはE) 変換の変換結果のうち、短い方の文字列を出力する。

変換指定子	説 明
s	(char *)型の引数データの指す文字から初めて、ヌル文字'\0' が現れるまでの文字列をそのまま出力する。
p	ポインタ型引数データを番地データと見て、それを16進数表示で出力する。
n	文字は出力しない。引数で与えられた (int *) 型ポインタの指す領域に、 このprintf関数で出力された それまでの文字数 を格納する。
%	'%%'という変換指定により1つの%文字を出力する。

例1. 8 (実数データの出力書式の比較) 指数関数 $\mathbf{f(x)=3.14\times10^{x}}$ の $\mathbf{x=-5,\ -4,\ -3,\ -2,\ ...,\ 7,\ 8}$ に対する値がprintf( )に用意されている3つの変換指定子 e, f, g によって実際にどの様に出力されるのかを次に示す。

```
[motoki@x205a]$ nl fundamentals-printf-e-f-g-conversion.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   int    x;
     5   double fx;

     6   printf("-----------------------------------------
     7          "関数 f(x)=3.14*10^x の x=-5,-4,-3, ..., 7,8 に
     8          "e,f,g変換記述子によって実際にどの様に出力...\n"
     9          "-----------------------------------------
```

```
    10          " x     %%12.5e         %%12.5g         %%#12.5g
    11          "--  ------------  ------------  ------------

    12   fx=3.14e-5;
    13   for (x=-5; x<=8; x++) {
    14     printf("%2d  %12.5e  %12.5g  %#12.5g  %12.5f\n",
                  x, fx, fx, fx, fx);
    15     fx *= 10.0;
    16   }
    17   return 0;
    18 }
[motoki@x205a]$ gcc fundamentals-printf-e-f-g-conversion.c
[motoki@x205a]$ ./a.out
-------------------------------------------------------
関数 f(x)=3.14*10^x の x=-5,-4,-3, ..., 7,8 に対する値が
e,f,g変換記述子によって実際にどの様に出力されるかを見る。
-------------------------------------------------------
```

x	%12.5e	%12.5g	%#12.5g	%12.5f
-5	3.14000e-05	3.14e-05	3.1400e-05	0.00003
-4	3.14000e-04	0.000314	0.00031400	0.00031
-3	3.14000e-03	0.00314	0.0031400	0.00314
-2	3.14000e-02	0.0314	0.031400	0.03140
-1	3.14000e-01	0.314	0.31400	0.31400
0	3.14000e+00	3.14	3.1400	3.14000
1	3.14000e+01	31.4	31.400	31.40000
2	3.14000e+02	314	314.00	314.00000
3	3.14000e+03	3140	3140.0	3140.00000
4	3.14000e+04	31400	31400.	31400.00000
5	3.14000e+05	3.14e+05	3.1400e+05	314000.00000
6	3.14000e+06	3.14e+06	3.1400e+06	3140000.00000
7	3.14000e+07	3.14e+07	3.1400e+07	31400000.00000
8	3.14000e+08	3.14e+08	3.1400e+08	314000000.00000

[motoki@x205a]$

例1. 9 (s変換指定子) s変換を用いると(あまり好ましくありませんが)例題1.2のプログラム(double型で処理する版)の11~12行目は次の様に書くことも出来る。

```
printf("%s %f, %s %f %s\n      = %f\n",
        "底面の半径が", r, "高さが", h, "の円錐の体積",
        PI*r*r*h/3.0);
```

関数printfにおける型限定子：

出力データの入った領域の大きさについての認識を調節するために、次の指定が可能。

型限定子	説 明
(なし)	d または i 変換の時、引数のデータ型は int と見なされる。
	u, o, x または X 変換の時、unsigned int
	n 変換の時、(unsigned)int へのポインタ
	e, E, f, g または G 変換の時、double
h	d または i 変換の時、short int
	u, o, x または X 変換の時、unsigned short int
	n 変換の時、(unsigned)short int へのポインタ
l	d または i 変換の時、long int
	u, o, x または X 変換の時、unsigned long int
	n 変換の時、(unsigned)long int へのポインタ
L	e, E, f, g または G 変換の時、long double

関数printfにおける最小フィールド幅の指定：

表の形に揃えて表示したい時のために、出力フィールド(i.e.出力する場所)の大きさの最小値を正整数で、または星印 * で指定することが出来る。[省略も可。]

関数printfにおける「.精度」の指定：

精度は非負整数または星印 * で指定することが出来る。[省略も可。]

関数printfにおけるフラグ部の指定：

必要に応じて自分でよく読む。
場合によっては、浦&原田(編)「C入門」の付録4、ケリー& ポール「CのABC(下)」の第11.2節、等も参照。

1–4–6 書式付き入力 —scanf—

関数scanfの構文：

- 関数scanfのデータ型は次の通り。

```
int scanf( 書式 , 式 , 式 , ... );
```

- 書式 は「(データ)変換指定」や入力中に現れるはずの単語等を並べて、2重引用符で囲むことによって指示する。
- 書式に続く 式 は、入力データを格納するための領域を指す(ポインタ型の)式である。
- 変換指定は文字列で表されている入力データをどのデータ型の内部表現形式に変換するかを指定したもので、その一般形は

  %[代入抑止文字][最大フィールド幅]
  [型限定子]変換指定子

  但し、[...] の部分はそれぞれオプションで、省略可。

  となっている。

関数scanfの実行の流れ：

- 入力ストリームから取り出された個々の入力データは、順番に書式中の「変換指定」に従って内部表現形式に変換され、第2引数以下で指定された番地に1つずつ格納されてゆく。

  [入力ストリーム、書式、入力領域の列の3つを見比べながら処理が進む。]

- 関数値＝「入力に成功したデータの個数」である。但し、途中で入力が無くなった場合は、EOF (マクロ;普通 -1 が割り当てられている) を返す。

- 特殊な場合 (i.e. 書式の中の次の変換が%cまたは'['変換の場合) を除いて、入力ストリーム中の空白類 (i.e. 空白、改行コード、tabコード) は入力データの区切りとして働き読み飛ばされる。

- 書式中(「変換指定」の中を除く)に空白類が現れた場合には、入力中で次に非空白類の文字が現れるまで入力文字が読み飛ばされる。

- 書式中に「変換指定」の一部でも空白類でもない文字が現れた場合には、その文字が次の入力文字になっていなければならない。

  [一致しなければ、データ入力の実行は(途中であっても)終了する。]

関数scanfで可能な変換指定子：

次のような変換指定子が用意されている。

変換指定子	説 明
d	入力文字列を10進整数と見てint型の内部表現形式に変換し、指定された記憶領域に格納する。
i	入力文字列が 0x または 0X で始まっていれば16進整数表記、それ以外で 0 で始まっていれば8進表記、それ以外なら10進表記の整数と見てint型の内部表現形式に変換し、指定された記憶領域に格納する。
u	10進整数表記の文字の並びを unsigned int型の内部表現形式に変換し、指定された記憶領域に格納する。
o	8進整数表記の文字の並びを unsigned int型の内部表現形式に変換し、指定された記憶領域に格納する。[但し、符号付きの入力データもOK。数字部は 0 で始まっていても良い。]
x X	16進整数表記の文字の並びを unsigned int型の内部表現形式に変換し、指定された記憶領域に格納する。[但し、符号付きの入力データもOK。数字部は 0x や 0X で始まっていても良い。]

変換指定子	説 明
e	入力文字列を浮動小数点表記の実数と見てfloat型(型限定子によってdoubleやlong doubleに指定変更可)の内部表現形式に変換し、指定された 記憶領域に格納する。
E	
f	
g	
G	
c	「最大フィールド幅」部で指定された長さ(デフォルトは1)の入力文字列を文字コードのまま(すなわち無変換で)指定された記憶領域に格納する。但し、入力ストリームの途中に空白類が現れても読み飛ばさない。また、格納の際、ヌル文字 '\0' は(最後に)付け加えられない。
s	空白類文字で区切られた入力文字列を次の入力と見て、その文字コードの列を指定された記憶領域に格納する。但し、格納の際、その文字コードの列の最後にヌル文字 '\0' を付け加える。

変換指定子	説 明
[文字列]	文字集合 $\Sigma =$ { 文字列 に現れる文字の集合 **if** 文字列 が ’^’ 以外の文字で始まる 文字列 に現れない文字の集合 **if** 文字列 が ’^’ という文字で始まる 内の文字だけで構成される最長の文字列を入力データとして取り出し、その文字列の最後にヌル文字 ’\0’ を付けた文字コードの列を指定された記憶領域に格納する。
p	ポインタ型データの出力形式 (**i.e.**%p 変換による出力の形式;処理系に依存) をポインタ型の内部表現に変換し、指定された記憶領域に格納する。
n	それまでに読み込まれた文字の数を指定された記憶領域に格納する。
%	次の文字が ’%’ であることを確認する。[単に、書式指定の中では ’%%’ で ’%’ という文字を表すということ。当然、入力ストリームで次の文字が % になっていないと、データ入力は中止 (エラー) となる。]

関数scanfにおける型限定子：

データを格納する記憶領域の大きさについての認識を調節するために、次の指定が可能。

型限定子	説 明
(なし)	d, i またはn変換の時、格納領域のデータ型がint と見なされる。
	u, o, x またはX変換の時、unsigned int
	e, E, f, g またはG変換の時、float
h	d, i またはn変換の時、short int
	u, o, x またはX変換の時、unsigned short int
l	d, i またはn変換の時、long int
	u, o, x またはX変換の時、unsigned long int
	e, E, f, g またはG変換の時、double
L	e, E, f, g またはG変換の時、long double

関数scanfにおける最大フィールド幅の指定：

1個の入力データを表すための最大文字数を正整数で指定できる。これが指定されていない場合は、文字数の上限は考慮されない。

関数scanfにおける代入抑止文字の指定：

星印 * を指定すると、この変換指定子に対応する入力データは読み飛ばされる。

例1. 10 ([変換,代入抑止文字) 行末までのデータを読み飛ばすには、例えば次のように書く。

```
scanf("%*[^\n]");
```

必要に応じて自分でよく読む。
場合によっては、浦&原田(編)「C入門」の付録4、ケリー& ポール「CのABC(下)」の第11.2節、等も参照。

1–4–7 配列

配列について：

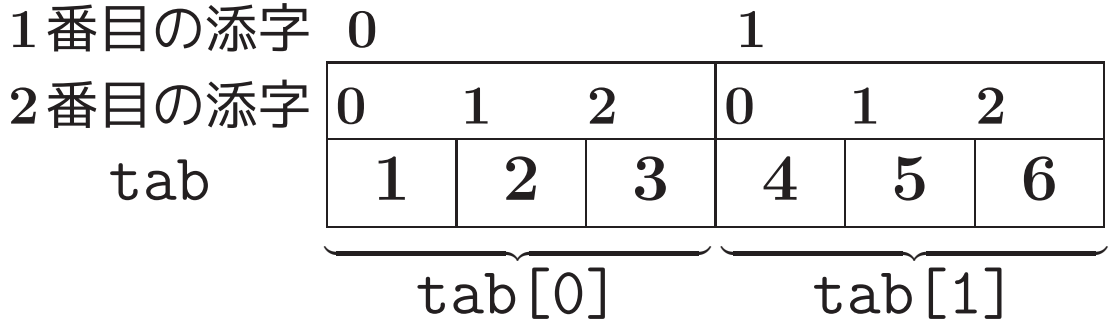
- 配列名が a、大きさが $\mathbf{k_1}\times\mathbf{k_2}\times\cdots\times\mathbf{k_n}$ の配列の宣言／領域確保は次の様に行う。

  データ型　$\mathbf{a[k_1][k_2]\cdots[k_n]}$

- 宣言時の配列の初期化は例えば次の様に行う。

```
int num[12]={31,28,31,30,31,30,31,31,30,31,30,31};

char tab[2][3]={{1,2,3}, {4,5,6}};
```



文字列について：

- **char** 型の配列を使う。

- 文字列の終わりの印として文字列の最後にヌル文字'\0'を置く。

  ⇨ (配列の大きさ) ≥ (文字列の長さ) + 1 でなければならない。

0	1	2	3	4	5	6	
's'	't'	'r'	'i'	'n'	'g'	'\0'	···

- 文字列を 2 重引用符で囲めば文字列定数になる。

- **char** 型配列で文字列を表す場合は、初期設定を次の様に行うことが出来る。

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

(`char s[]={'s','t','r','i','n','g','\0'};` と同等。)