5 整数計算の簡単なプログラム例

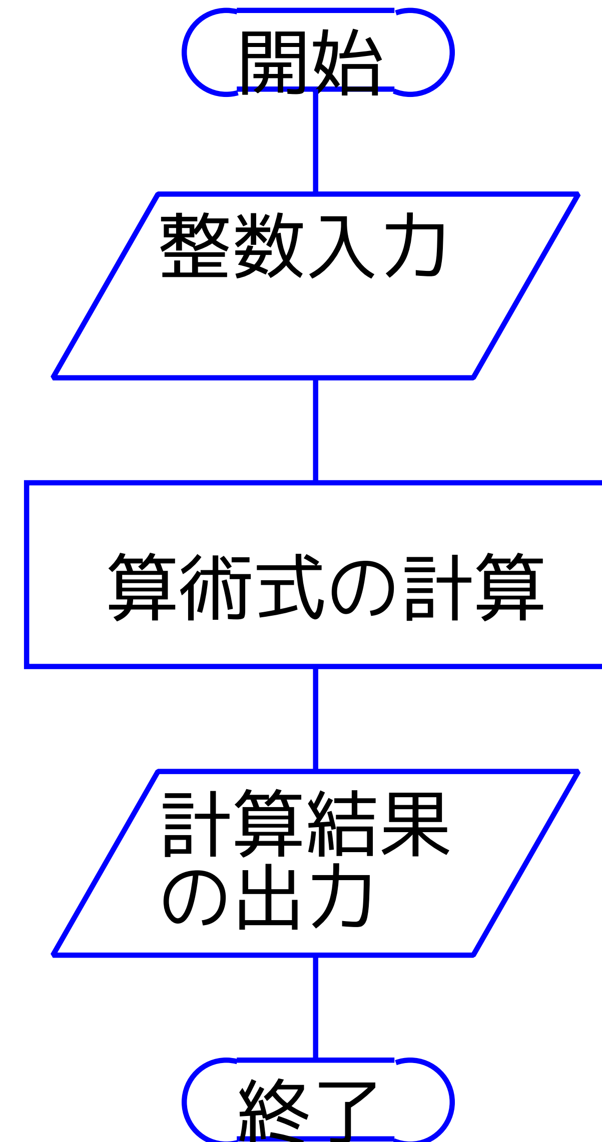
この節では、

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。

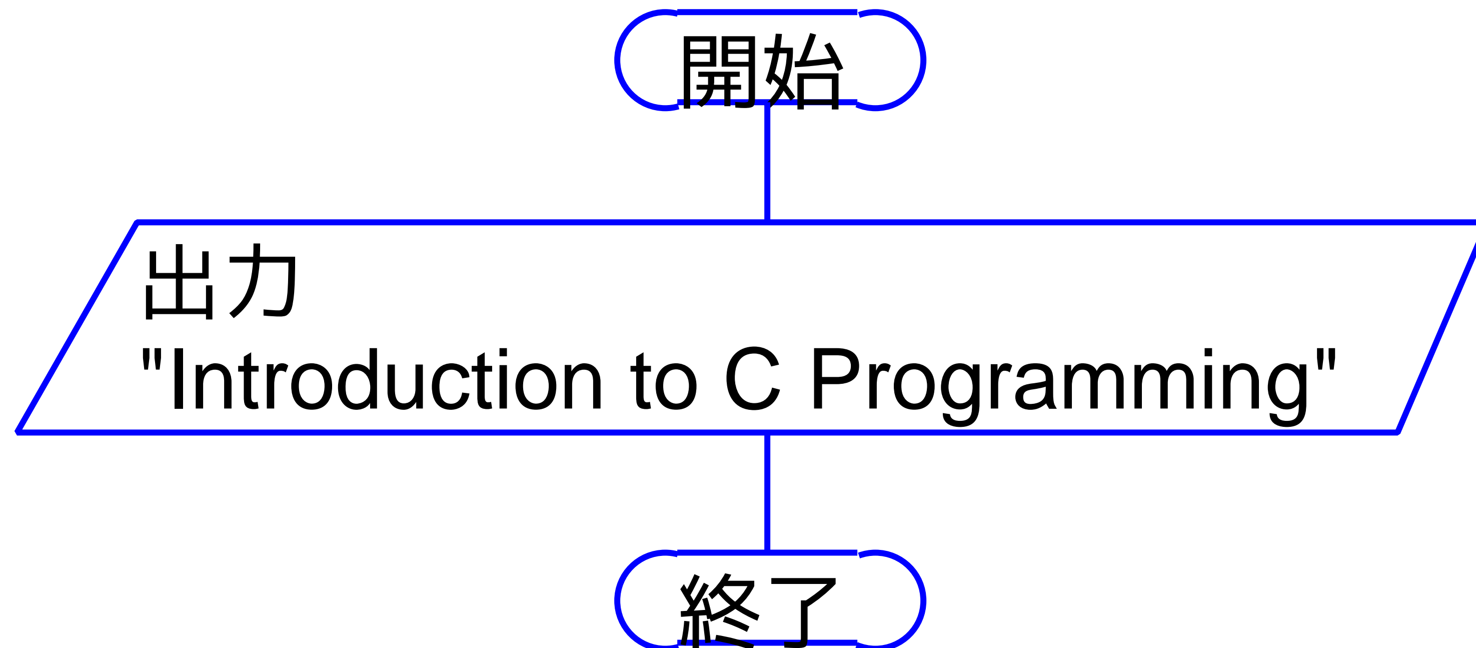


5–1 決められた文字列の出力

例題 5. 1 (決められた文字列を出力) 単に

```
Introduction to C Programming
```

と出力するだけの C プログラムを作成せよ。



```
[motoki@x205a]$  nl output-constant-string.c  Enter
                                          ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                            */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$  gcc output-constant-string.c  Enter
                                          ............. (コンパイル)
[motoki@x205a]$  ./a.out  Enter ....................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$ nl output-constant-string.c  Enter
                                          ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                          */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$ gcc output-constant-string.c  Enter
                                          ............. (コンパイル)
[motoki@x205a]$ ./a.out  Enter ..................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$  nl output-constant-string.c  Enter
                                          ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                          */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$  gcc output-constant-string.c  Enter
                                          ............. (コンパイル)
[motoki@x205a]$  ./a.out  Enter ....................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$ nl output-constant-string.c   Enter
                                     ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                           */

     3 #include <stdio.h>

     4 int main(void)                           字下げ, 空白, tab, 改行
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$
```

```
int main(void){printf("Introduction to C Programming\n");retur
```

```
[motoki@x205a]$  nl output-constant-string.c  Enter
                                          ...........  (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                        */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$  gcc output-constant-string.c  Enter
                                          ..............  (コンパイル)
[motoki@x205a]$  ./a.out  Enter .....................  (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$ nl output-constant-string.c  Enter
                                    ............ (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                          */

     3 #include <stdio.h>

     4 int main(void)                 関数, 関数定義, プログラム
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$ gcc output-constant-string.c  Enter
                                    ............. (コンパイル)
[motoki@x205a]$ ./a.out  Enter ......................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$  nl output-constant-string.c  Enter
                                                .......... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                          */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$  gcc output-constant-string.c  Enter
                                                ............ (コンパイル)
[motoki@x205a]$  ./a.out  Enter ..................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$ nl output-constant-string.c  Enter
                                    ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                         */

     3 #include <stdio.h>    ... /usr/include/stdio.h

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$ gcc output-constant-string.c  Enter
                                    ............. (コンパイル)
[motoki@x205a]$ ./a.out  Enter ..................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$  nl output-constant-string.c  Enter
                                        ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                          */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$  gcc output-constant-string.c  Enter
                                        ............. (コンパイル)
[motoki@x205a]$  ./a.out  Enter ..................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

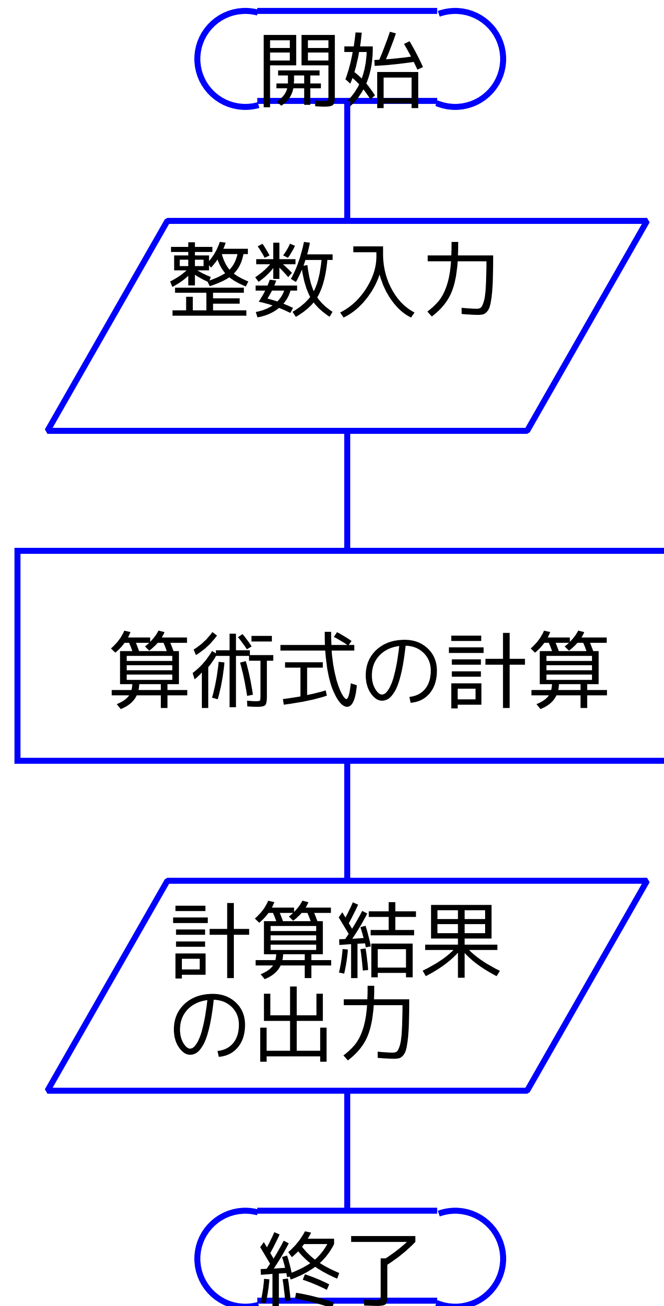
```
[motoki@x205a]$ nl output-constant-string.c  Enter
                                          ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                         */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$ gcc output-constant-string.c  Enter
                                          ............. (コンパイル)
[motoki@x205a]$ ./a.out  Enter ...................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$ nl output-constant-string.c  Enter
                                        ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                          */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$ gcc output-constant-string.c  Enter
                                        ............. (コンパイル)
[motoki@x205a]$ ./a.out  Enter ..................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$  nl output-constant-string.c   Enter
                                        ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                           */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$  gcc output-constant-string.c   Enter
                                        ............. (コンパイル)
[motoki@x205a]$  ./a.out   Enter .................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

```
[motoki@x205a]$ nl output-constant-string.c  Enter
                                        ........... (ファイル表示)
     1 /* 決められた文字列 "Introduction to C Programming" */
     2 /* を出力するだけのCプログラム                            */

     3 #include <stdio.h>

     4 int main(void)
     5 {
     6   printf("Introduction to C Programming\n");
     7   return 0;
     8 }
[motoki@x205a]$ gcc output-constant-string.c  Enter
                                        ............. (コンパイル)
[motoki@x205a]$ ./a.out  Enter ..................... (実行)
Introduction to C Programming
[motoki@x205a]$
```

5–2 四則演算



例題5．3 (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、

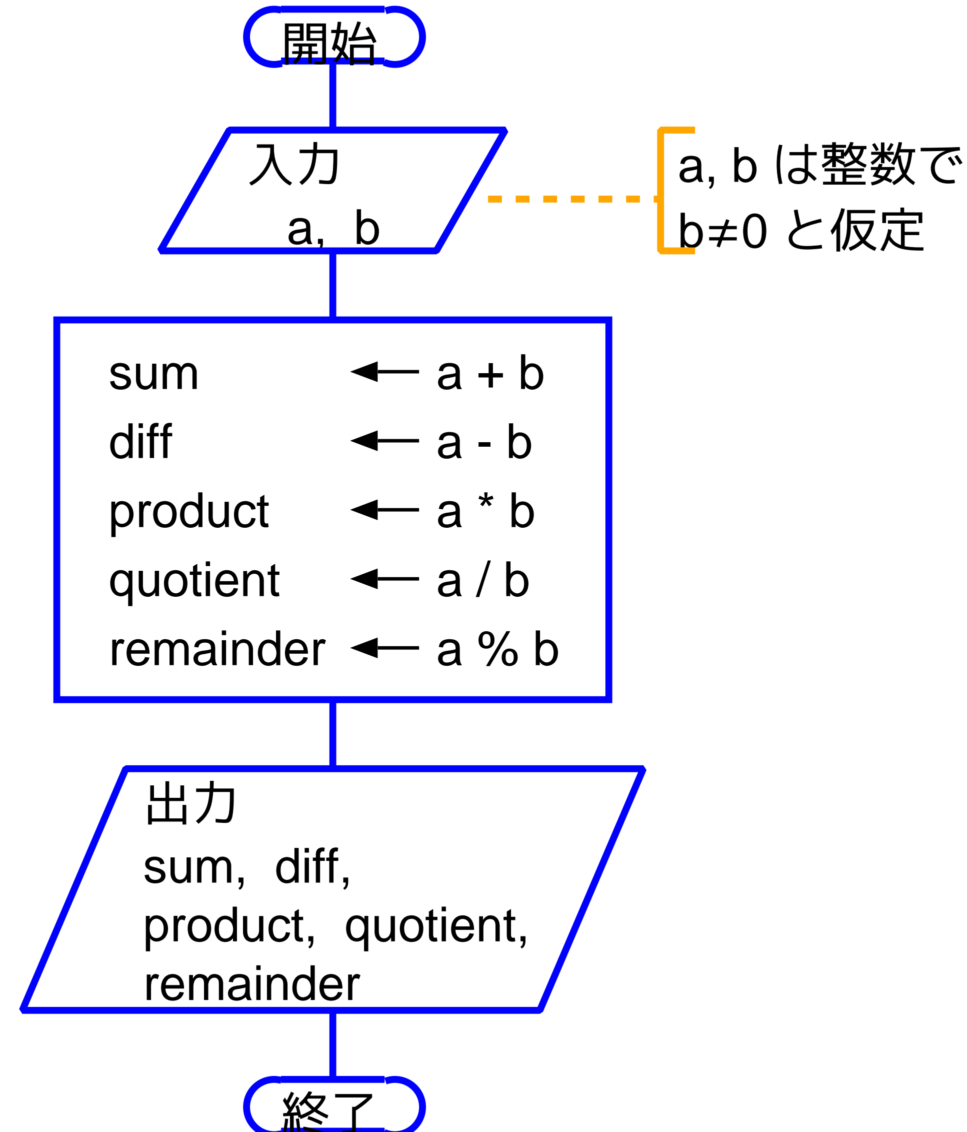
読み込むデータを格納する場所が必要

⟹ 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

⟹ sum, diff, product, quotient, remainder 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 arithmetic-operations-over-int.c  Enter
                                 ..................  (ファイル表示)
     1  /* 2つの整数データを変数 a と b に読み込み、それらの     */
     2  /* 和, 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  {          int型変数
     6    int a, b, sum, diff, product, quotient, remainder;

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

     8    sum      = a+b;
     9    diff     = a-b;
    10    product  = a*b;
    11    quotient = a/b;
```

```
12    remainder= a%b;

13    printf("\nInput data: %d, %d\n\n"
14            "Sum:         %d\n"
15            "Difference:  %d\n"
16            "Product:     %d\n"
17            "Quotient:    %d\n"
18            "Remainder:   %d\n",
19            a, b, sum, diff, product, quotient, remainder);
20    return 0;
21 }
[motoki@x205a]$  gcc -o arith-op arithmetic-operations-over-in
 Enter
                ........ (実行ファイルの名前を指定してコンパイル)
[motoki@x205a]$  ./arith-op  Enter ................... (実行)
11 3  Enter ................................ (データの入力)
```

```
Input data: 11, 3

Sum:        14
Difference: 8
Product:    33
Quotient:   3
Remainder:  2
[motoki@x205a]$
```

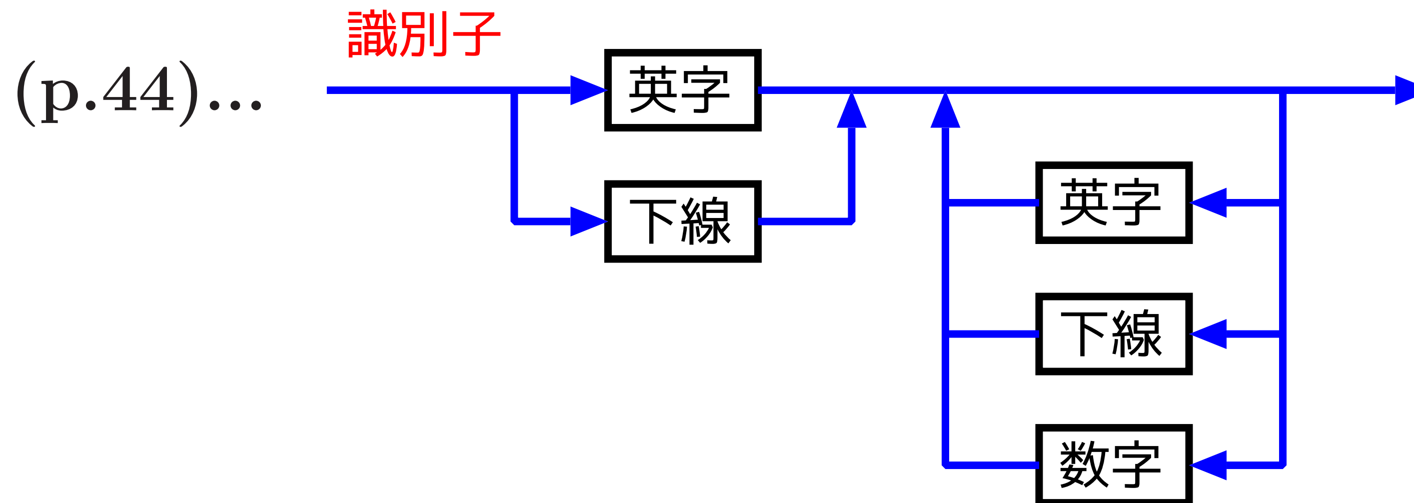
変数の名前の付け方：

C 言語では、変数に付ける名前として、

英字 (または下線) で始まり、

それに英数字や下線が続く文字の並び

を使うことが出来る。 (大文字と小文字は区別する。)



⇨ こういった制約の下で、

使い方/役割に応じた適切な名前を付ける

ことが大切である。

代入文： “sum=a+b;” の様に、プログラムの中で
式の計算をしてその結果を変数に格納する動作を表す部分を一般に代入文と呼ぶ。

特にC言語においては、
変数名 や 123 といった 数を表す文字列 に +, -, *, /, % といった 演算子 や 丸括弧 を組み合わせて色々な 式を構成し、

変数名 = 式 ;

変数名 += 式 ; ....(変数名 = 変数名 + 式 ; と同等)

変数名 -= 式 ; ....(変数名 = 変数名 - 式 ; と同等)

変数名 *= 式 ; ....(変数名 = 変数名 * 式 ; と同等)

変数名 /= 式 ; ....(変数名 = 変数名 / 式 ; と同等)

変数名 %= 式 ; ....(変数名 = 変数名 % 式 ; と同等)

....

という形の代入文を書くことができる。

例5. 4 (代入文) int型変数 sum, x に関して、次の2つの代入文は同じ計算を行う。

```
sum = sum + x;
sum += x;
```

算術式の計算：(コンピュータによる)代入文の実行においては、等号の右側の算術式の計算は通常の数学と同じ順序で1つずつ行われる。すなわち、......

当然、途中で行われる各々の演算の結果はコンピュータの中に(一時的に)保持されなければならず、演算結果を保持する内部表現形式も決まっているはずである。

⇨ 算術式を構成する各々の(部分)算術式に対して、
そのデータ型が定められている。
例えばint型データ同士の演算の場合は、演算結果はint型

例5. 5 (算術式における演算順序の指定) int 型変数 h, m, s, time に関して、次の4つの代入文は同等の計算結果をもたらす。

```
time = h*60*60 + m*60 + s;
time = (((h*60)*60) + (m*60)) + s;
time = (h*60 + m)*60 + s;
time = h*3600 + m*60 + s;
```

例5. 6 (式の計算に関する注意)

int 型同士の除算においては商の小数部は捨てられる。

⇨ 数学的に等しい式であっても、
C 言語の算術式としては異なる値を持つことがある 。

(例) 数学的には 1/2*a=a*1/2 であるが、
大抵の場合2つの代入文

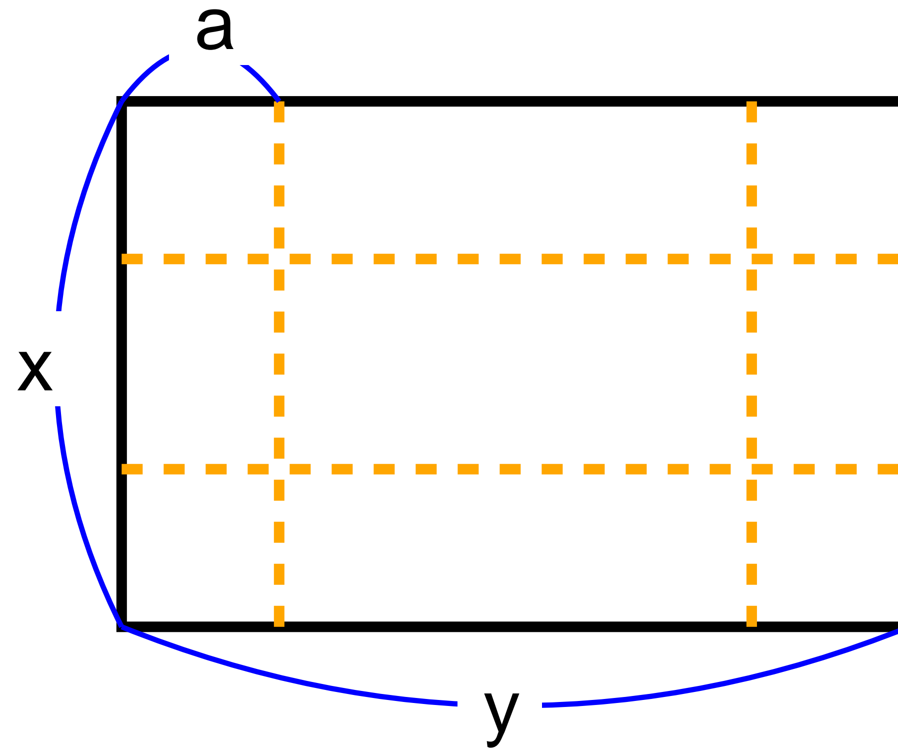
`x=1/2*a;` と `x=a*1/2;`

は異なる実行結果をもたらす。

例題5.7 (直方体の体積) **3**つの整数データ **x, y, a** を読み込み、それらを使って

縦,横が各々 **x cm, y cm**の厚紙の四隅を
a cm 四方だけ切り取ってできる直方体の体積

を計算して出力する**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

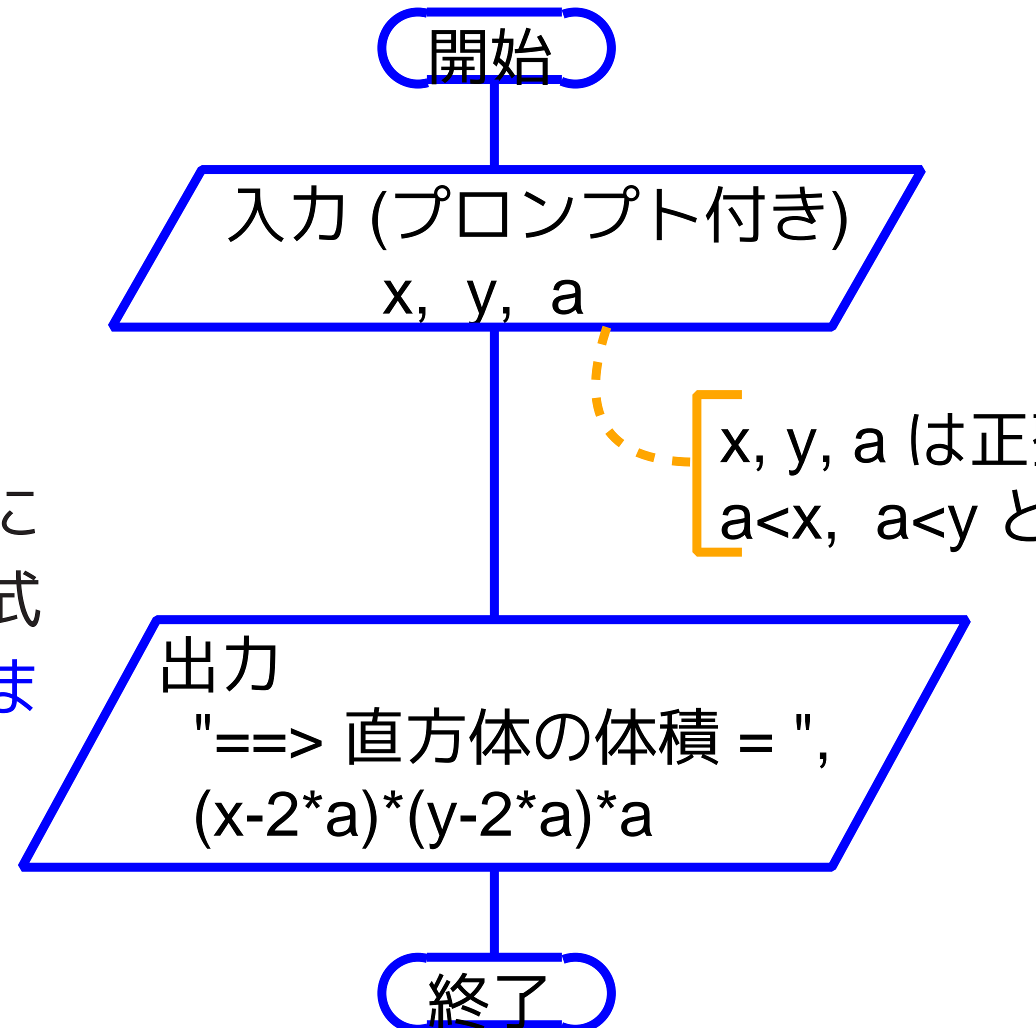
⟹ x, y, a という名前の記憶領域

⇨ 縦が x-2*a cm, 横が y-2*a cm, 高さが a cm の直方体が出来る。

⇨ 求める体積は (x-2*a)*(y-2*a)*a

データ入力を促す文字列も出すことにし、また printf 関数を使って算術式の計算結果を変数に格納せずにそのまま出力することにすれば、

⇨ コンピュータが行うべき処理は右図の通り。



```
[motoki@x205a]$ nl rectangular-parallelepiped.c
     1 /* 3つの整数データ x, y, a を読み込み、それらを使って */
     2 /*      縦,横が各々 x,y の厚紙の四隅を a 四方だけ          */
     3 /*      切り取ってできる直方体                              */
     4 /* の体積を計算して出力するCプログラム                      */

     5 #include <stdio.h>

     6 int main(void)
     7 {
     8   int x, y, a;

     9   printf("厚紙の縦 = ? ");
    10   scanf("%d", &x);
    11   printf("厚紙の横 = ? ");
    12   scanf("%d", &y);
    13   printf("切り取る正方形の一辺 = ? ");
```

```
14    scanf("%d", &a);

15    printf("\n==> 直方体の体積 = %d 立方cm\n",
16             (x-2*a)*(y-2*a)*a);
17    return 0;
18 }
[motoki@x205a]$  gcc rectangular-parallelepiped.c
[motoki@x205a]$  ./a.out
厚紙の縦 = ?  38
厚紙の横 = ?  57
切り取る正方形の一辺 = ?  8

==> 直方体の体積 = 7216 立方cm
[motoki@x205a]$
```

補足：

一般に、プログラムの中でコンピュータの基本動作に相当する部分を**文**と呼ぶ。特に**C**言語においては、セミコロン(;) は文の終わりを表す記号である。

C言語においては、等号 = は + や ∗ と同じ様に**演算子**(**operator**)に分類され、**代入演算子**と呼ばれる。 そして、変数 = 式 というもの自身が値をもつ式として扱われる。 [変数に値が格納されるのは単なる副作用と考える。]

□演習5. 9 (四捨五入) 9桁以下の2つの正整数 $\mathbf{m}$, $\mathbf{n}$ を入力して、

$$\frac{\mathbf{m}}{\mathbf{n}}$$ の小数部を四捨五入して得られる整数値

を出力するプログラムを作成せよ。

Hint：

このプログラムは、例えば $\mathbf{m=7, n=3}$ なら $\mathbf{2}$ を、$\mathbf{m=8, n=3}$ なら $\mathbf{3}$ を出力することになる。そのために、実際にはプログラム内で

$$\left\lfloor \frac{\mathbf{m}}{\mathbf{n}} + \frac{\mathbf{1}}{\mathbf{2}} \right\rfloor = \left\lfloor \frac{\mathbf{2m+n}}{\mathbf{2n}} \right\rfloor = \left\lfloor \frac{\mathbf{m}+\lfloor \mathbf{n/2} \rfloor}{\mathbf{n}} \right\rfloor$$

の計算を int 型算術式で表せばよい。　ここで、$\lfloor \mathbf{x} \rfloor$ は $\mathbf{x}$ を越えない最大整数を表す。

Hintの補足：

C言語で		数学的な意味
`int a, b;`		
`......`		
`...  a/b ...  ;`	$\Longleftrightarrow$	$\left\lfloor \frac{\text{変数 a の保持する値}}{\text{変数 b の保持する値}} \right\rfloor$

5–3 付録 C プログラムの基本的な形 —C 文法の

⇨ 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。

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 関数名 ( データ型 名前,.......)        ···頭部
{
    局所変数の宣言
        ⋮
    局所変数の宣言
    文
        ⋮
    文
}                                            ··· 本体(複合文)
```

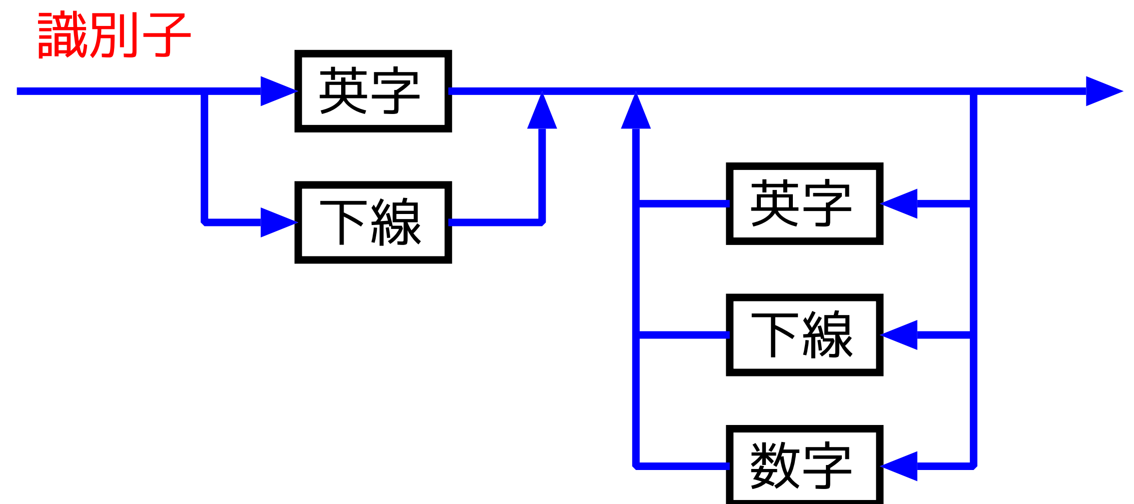
- 関数値の型 の部分は省略可能で、省略すると `int` と見なされる。
- 名前を表す文字列の途中を除いて、どこで改行してもよいし、どこに空白を挿入してもよい。

⇨ 字下げ等

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。

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。

増分演算子と減分演算子:

++変数等 $\cdots$ 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 $\cdots$ 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++ $\cdots$ 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-- $\cdots$ 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。

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

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。

5–4 付録 プログラムのコンパイルと実行

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。

($\Longrightarrow$ **19.1** 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
⋮

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 3.141592653589793
```

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 3.141592653589793 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または
  インクルードファイルという。)

- 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。

5–5 付録 書式付き出力 —printf—

{浦&原田付録4, ケリー&ポール11.2節}

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 i	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。
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 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。)
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5. 11 (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)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。

5–6 付録 書式付き入力 —scanf—

{浦&原田付録4, ケリー&ポール11.2節}

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。]

<table>
<tr><th>変換指定子</th><th>説 明</th></tr>
<tr><td>e</td><td rowspan="5">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。</td></tr>
<tr><td>E</td></tr>
<tr><td>f</td></tr>
<tr><td>g</td></tr>
<tr><td>G</td></tr>
<tr><td>c</td><td>「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。</td></tr>
<tr><td>s</td><td>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。</td></tr>
</table>

変換指定子	説 明

[文字列]	文字集合
	Σ =
	{ 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5. 14 ([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。