10 関数の定義 —処理の分割—

(幾つかのパラメータを除いて) 全く同一の処理を1つのプログラムの中で複数回行いたいこともある。この様な場合、それら各々の細かな処理を別々に手順の中に書き込むと、プログラムが長く読みにくくなってしまう。

補足：
本質的に同じ処理がプログラムのあちこちで繰り返されていない場合でも、長い処理手順は広い範囲で共有される変数を含むので概して分かりにくくなる。

⇨ プログラムを複数の機能単位 (ここでは関数) に分けて構成する方法について説明する。

補足：
ここで考えている「機能単位」とは、printf(), scanf() や数学関数の様な関数のことで、1つの処理を関数として定義する方法を述べる。

10–1 関数定義の例

例題10. 1 (二項係数の計算)　n 個のものから k 個を選ぶ組合せの数 $\binom{n}{k}$ は

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

と計算できる。2つの正整数データ n と k を読み込みこの計算式に基づいて組合せの数 $\binom{n}{k}$ を計算して出力するCプログラムを作成せよ。

(考え方)

計算式に階乗計算が3箇所もあるので、main()関数とは別に、階乗計算を行う関数 factorial() を定義するのが自然であろう。

引数として整数値を受け取りその階乗値を返す関数 factorial() が記述されていれば、 組合せの数 $\binom{n}{k}$ の計算は 、数学関数と同じ様に factorial() を呼び出して

$$\binom{n}{k} = \frac{\texttt{factorial(n)}}{\texttt{factorial(k)}\,\texttt{factorial(n-k)}}$$

という風に行うことができる。

関数 factorial() に与える 引数データは 、我々が入力する正整数、およびそれらの差であるので、そのデータ型は int とするのが妥当である。

また、factorial() の 関数値は 本来整数であるが、int 型で表せる範囲を越えてしまう危険性もあるので、そのデータ型を実数型にして階乗値も組合せの数も近似計算する方が無難である。

⇨ 関数の型は `double factorial(int k);`

階乗計算については 、例題 7.6 と同じ風に行えば良い。

(プログラミング)

```
[motoki@x205a]$  nl binomial-coeff.c   Enter
     1 /* 2つの正整数データ n と k を読み込み                    */
     2 /* 二項係数 n!/(k!*(n-k)!) を出力するCプログラム */

     3 #include  <stdio.h>

     4 double factorial(int k);
```

関数プロトタイプ

一般に、関数プロトタイプは

次のような構造をしている。

関数値のデータ型 関数名 (データ型 名前, ... , データ型 名前);

または

関数値のデータ型 関数名 (データ型 , ... , データ型);

```
 5 int main(void)
 6 {
 7   int  n, k;   ←22行目のkとは別領域

 8   printf("It will compute a binomial coefficient.\n"
 9          "Input two positive integers n and k(<=n):  ")
10   scanf("%d%d", &n, &k);

11   printf("\nThe number of the combinations of\n"
12      "      n objects taken k at a time = %20.14g\n",
13      factorial(n)/(factorial(k)*factorial(n-k)));
14   return 0;    ↑              ↑              ↑
15 }              実引数
```

```
16 /*-------------------------------------------------------
17 /* 階乗値を計算してその結果を返す関数                    */
18 /*-------------------------------------------------------
19 /*    (仮引数) k : 何の階乗を計算するかを表す整数        */
20 /*    (関数値)   : k! の値をdoubleで                      */
21 /*-------------------------------------------------------
22 double factorial(int k)
23 {
24    int    i;
25    double fact;
26
27    fact = 1.0;
28    for (i=2; i<=k; ++i)
29      fact *= (double)i;
30    return fact;
31 }
```

仮引数, 7行目のkとは別領域

自動変数

計算結果を呼出し元に返す
(関数値、戻り値、返却値)

```
[motoki@x205a]$  gcc binomial-coeff.c  Enter
```

```
[motoki@x205a]$  ./a.out  Enter
It will compute a binomial coefficient.
Input two positive integers n and k(<=n):   50 25   Enter

The number of the combinations of
      n objects taken k at a time =  1.264106064377 5e+14
[motoki@x205a]$
```

関数の仕様を記述することの利点：

各々の関数に仕様が書かれていると、作り上げた関数の処理内容を理解する際、その関数から呼び出す別の関数については処理内容を詳しく見る代わりに仕様を見るだけで良いので、一度に把握するプログラムの範囲が小さくて済む。

⇨ ◇ プログラムを理解し易くなる。

◇ 多数の関数が複雑に絡み合った大きなプログラムを作る場合もしっかりとしたプログラムを作ることが可能となる。

関数仕様の書き方について：

仕様としては、関数を使う側に対してどういう機能を提供するのかを書く。
それゆえ、

◇ 与えられた引数に対して、どういう関数値が返されるかを書く。

◇ 関数の外側で確保された変数等の値を変える作用、いわゆる副作用がある場合は、どういう副作用があるかも書く。

◇ 関数の内部の細かな変数や、処理手順についての記述は控えるべきである。

10–2 名前の有効範囲，局所変数，大域変数

大域的な変数や配列はプログラムを分かりにくくする原因にもなる。

それゆえ、C言語においては、関数定義

関数値のデータ型　関数名（ データ型 名前，. . . ，データ型 名前 ）　. . . 頭部

{

　宣 言

　⋮

　宣 言

　文

　⋮

　文

}　. . . 本体

の中で宣言され確保される変数や配列は、その関数定義の本体の中だけで使える局所的(local)なものとして扱われ、この関数の外部からはアクセスできない様になっている。

実際には、関数定義の本体部 (i.e. 仮引数列に続く { と } で囲まれた部分) は、ブロックの一種と考えられる。

ブロックと複合文： C 言語においては次の構造のものを複合文と呼び、そのうち実際に宣言が 1 個以上含まれているものをブロックと呼ぶ。

```
{
    宣 言
     ⋮
    宣 言
    文
     ⋮
    文
}
```

複合文／ブロックは、1 つの文が書ける所であればどこにでも置くことが出来るので、ブロックの中により小さなブロックが入り、その内側のブロックの中にまた別のブロックが入り、..... という入れ子構造 も可能である。

プログラムの中に出来るこの「ブロックの入れ子構造」に基づいて、変数等に付けた名前の有効範囲が次の様に決まる。

(規則1) どの名前 (の領域) も、それが宣言されたブロックの中だけでアクセスできる。

(規則2) 外側のブロックで宣言された名前を内側のブロックで再定義すると、外側の名前の領域 (i.e. その名前を持った外側の変数) は内側のブロックからはアクセスできなくなる。

(規則1) の理由 :

ブロック内で宣言された変数や配列の領域は、ブロック内の宣言の場所に制御が移ると自動的に確保され、ブロックの出口に制御が移ると解放される。

ブロックの中で宣言され局所的に使われるこれらの変数を自動変数という。

大域的な名前：

- 関数名はそのファイルのどの場所からでもアクセスできる。
- 関数の外で宣言された変数や配列はそのファイルのどの場所からでもアクセスできる。（外部変数, 外部配列という。）
  $\Longrightarrow$ ファイル全体をブロックの一種と見なすことも出来る。

> **注意：**
> 外部変数を使い過ぎると、副作用のために関数同士の独立性がなくなり、分かりにくいプログラムになる恐れがあります。

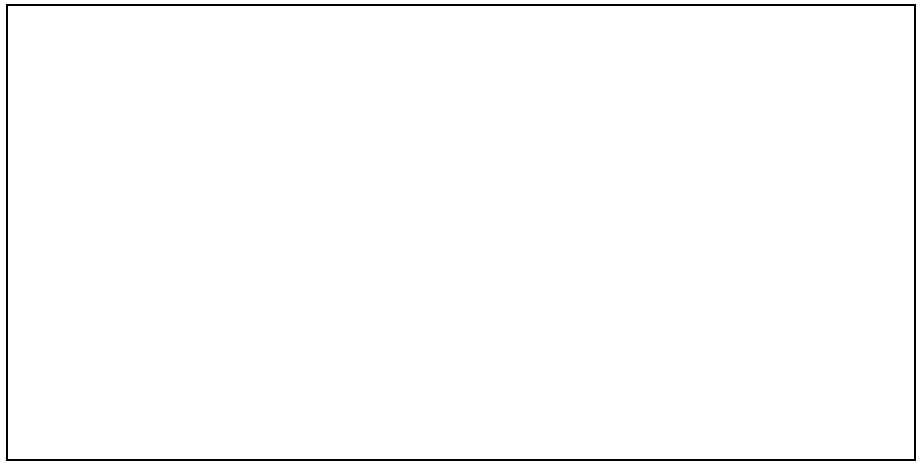
次の例題は、C言語における名前の有効範囲の規則を例示するものである。

例題10. 8 (名前の有効範囲, 外部変数) 次のCプログラムを実行するとどういう出力が得られるか？ 下の ＿＿＿＿ の部分に予想される出力文字列を入れよ。但し、ここでは空白は␣と明示せよ。

```
[motoki@x205a]$  nl scope-of-name.c  Enter
     1  /* 名前の有効範囲、外部変数の理解のためのプロ... */

     2  #include <stdio.h>

     3  void sub(void);

     4  int  a = 1;          /* 外部変数 */

     5  int main(void)
     6  {
     7    int  a = 22;      /* 自動変数 */
     8    printf("(1) %d\n", a);
```

```
 9    {                              /* ブロックの始まり */
10      int  a = 333;
11      printf("(2) %d\n", a);
12    }                              /* ブロックの終わり */

13    printf("(3) %d\n", a);
14    sub();
15    return 0;
16  }

17  void sub(void)
18  {
19    int  b = 4444;

20    printf("(4) %d\n", a);
21    printf("(5) %d\n", b);
22  }
```

```
[motoki@x205a]$  gcc scope-of-name.c  Enter
[motoki@x205a]$  ./a.out  Enter
```



```
[motoki@x205a]$
```

(考え方)　このプログラムの中に出来る「ブロックの入れ子構造」を明示すると次の様になる。

```
1  /* 名前の有効範囲、外部変数の理解のためのプログラム例 */
2  #include <stdio.h>
3  void sub(void);
4  int  a = 1;          /* 外部変数 */
5  int main(void)
6  {
7    int  a = 22;     /* 自動変数 */
8    printf("(1) %d\n", a);
9    {                          /* ブロックの始まり */
10     int  a = 333;
11     printf("(2) %d\n", a);
12   }                          /* ブロックの終わり */
13   printf("(3) %d\n", a);
14   sub();
15   return 0;
16 }
17 void sub(void)
18 {
19   int  b = 4444;
20   printf("(4) %d\n", a);
21   printf("(5) %d\n", b);
22 }
```

(実行結果)　結局、プログラムの

$$\left\{\begin{array}{l} 8\text{行目の a は　7行目で確保された a として、} \\ 11\text{行目の a は 10行目で確保された a として、} \\ 13\text{行目の a は　7行目で確保された a として、} \\ 20\text{行目の a は　4行目で確保された a として、} \\ 21\text{行目の b は 19行目で確保された b として} \end{array}\right.$$

解釈されることになるから、実行結果は 次の様になる。

```
[motoki@x205a]$  ./a.out  Enter
(1) 22
(2) 333
(3) 22
(4) 1
(5) 4444
[motoki@x205a]
```

10–3 再帰計算

例えば、漸化式

$$f_i = \begin{cases} 1 & \text{if } i=1 \\ i \times f_{i-1} & \text{if } i \geq 2 \end{cases}$$

が与えられていれば、f_i の値は次の様に計算できる。

$$f_i = i \times f_{i-1} = i \times (i-1) \times f_{i-2} = i \times (i-1) \times (i-2) \times f_{i-3} = \cdots = i!$$

⇨ この漸化式で f_i の計算式の中に f_{i-1} が出て来るのと同じ様に、関数定義の中に自分自身 (i.e. 定義しようとしている関数) の呼び出しを書くことができれば、漸化式に相当する関数定義を行い、漸化式による計算と同等の計算をその関数定義に基づいて行うことが、原理的にできるはずである。

一般に、関数定義の中で自分自身を呼び出すことを再帰呼び出しと言い、再帰呼び出しを伴う関数の実行を再帰計算と言う。

C言語においては、関数定義の中で自分自身を呼び出す、いわゆる再帰呼び出しが許されており、またその様な関数を実行する機構も備わっているので、漸化式による計算と同等の計算をプログラム上で行うことができる。

例題10. 10 (二項係数;階乗の再帰計算) 漸化式

$$f_i = \begin{cases} 1 & \text{if } i=1 \\ i \times f_{i-1} & \text{if } i \geq 2 \end{cases}$$

によって $f_i = i!$ と定まる。これを考慮に入れて、整数を1個引数として受け取りその階乗値をdouble型で計算して返す関数 factorial() を再帰的に定義せよ。 そして、この関数を用いて例題10.1と同じことを行うCプログラムを作成せよ。すなわち、正整数データ n と k を読み込み、n 個のものから k 個を選ぶ組合せの数を

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

と計算して出力するCプログラムを作成せよ。

(考え方)　ここで定義する関数factorial()の仕様(呼び出す側に対して提供する機能) 自体は例題10.1の場合と変わらない。

⇨ main()関数については例題10.1のものをそのまま流用できる。

⇨ 例題10.1で示したプログラムの中で、
関数 factorial() を指示に従って再帰的に定義し直すだけでよい。

(プログラミング)

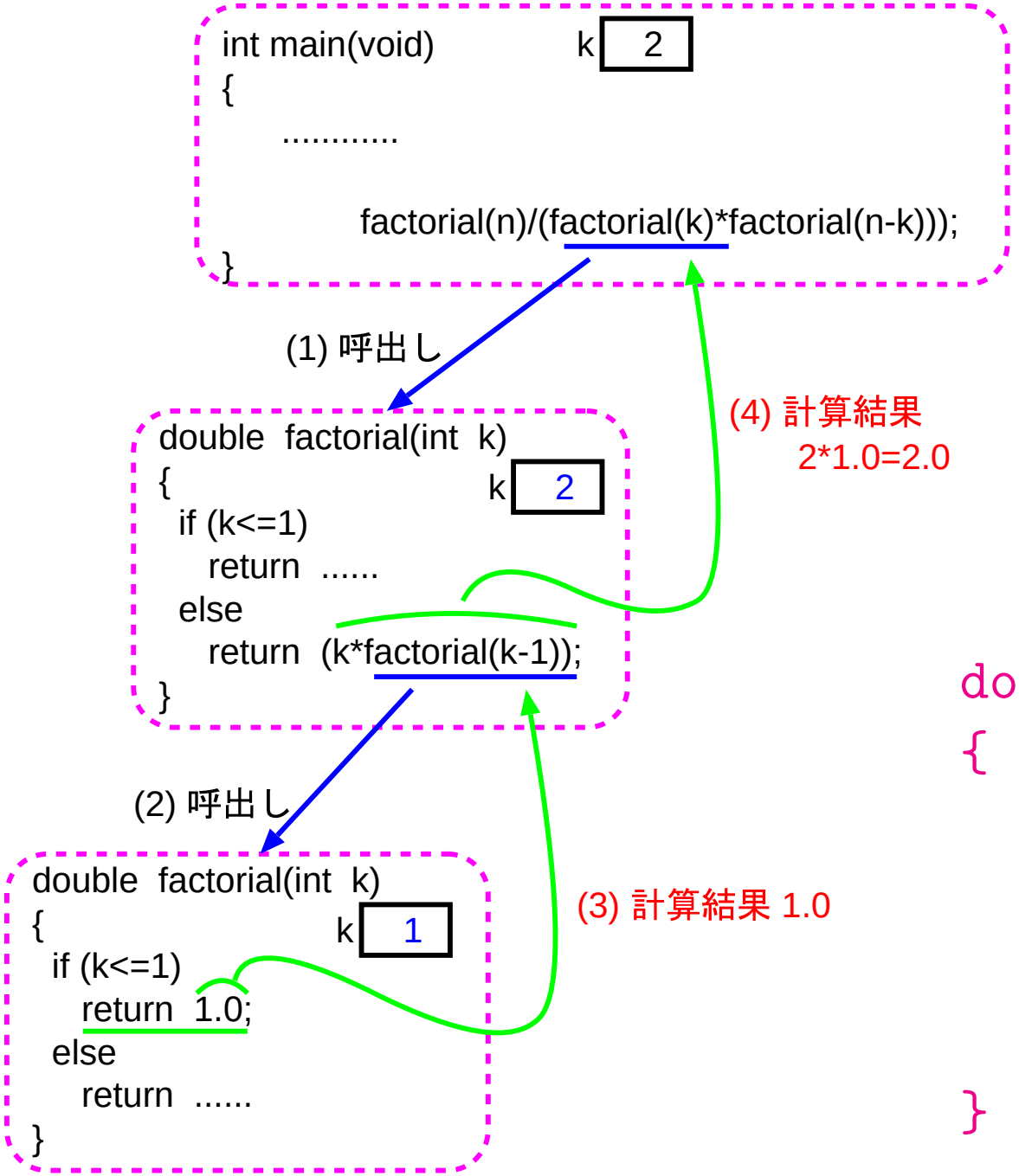
```
[motoki@x205a]$  nl binomial-coeff-using-rec-fatorial.c   Ente
     1 /* 2つの正整数データ n と k を読み込み                    */
     2 /* 二項係数 n!/(k!*(n-k)!) を出力するCプログラム */
     3 /*  (階乗値を再帰的に計算する関数を用意する。)     */

     4 #include <stdio.h>
```

```
 5 double factorial(int k);

 6 int main(void)
 7 {
 8   int  n, k;

 9   printf("It will compute a binomial coefficient.\n"
10          "Input two positive integers n and k(<=n):  ")
11   scanf("%d%d", &n, &k);

12   printf("\nThe number of the combinations of\n"
13          "      n objects taken k at a time = %20.14g\n
14          factorial(n)/(factorial(k)*factorial(n-k)));
15   return 0;
16 }
```

```
17 /*-------------------------------------------------------
18 /* 階乗値を計算してその結果を返す関数（再帰版）          */
19 /*-------------------------------------------------------
20 /*   （入力引数）k : 何の階乗を計算するかを表す整数      */
21 /*   （関数値）    : k! の値をdoubleで                    *
22 /*-------------------------------------------------------
23 double factorial(int k)
24 {
25   if (k <= 1)
26     return 1.0;
27   else
28     return (k * factorial(k-1));   再帰呼び出し
29 }
[motoki@x205a]$  gcc binomial-coeff-using-rec-fatorial.c   Ent
[motoki@x205a]$  ./a.out   Enter
It will compute a binomial coefficient.
Input two positive integers n and k(<=n):   50 25   Enter
```

```
The number of the combinations of
      n objects taken k at a time =  1.2641060643775e+14
[motoki@x205a]$
```

11行目でkの値として 2 が入力された場合のfactorial(k) の処理の様子




```
int main(void)
{
   int n, k;
   ......
   printf(......
          ......,
          .../(factorial(k)*
   return 0;
}
double factorial(int k)
{
   if (k <= 1)
      return 1.0;
   else
      return k*factorial(k-1);
}
```

10–4 パラメータの受渡し方法 —値呼出し vs. 参照呼出し

実引数と仮引数の対応付け：

関数呼び出しの際には、関数呼び出し側の実引数(または実パラメータ)と関数定義側の仮引数(または仮パラ メータ)の結合／対応付けが行われる。 引数結合の方式としては 次の2つが一般によく用いられている。

- 値呼出し(call by value)
  実引数として与えられた式が評価／計算され、その値が仮引数の変数の初期値として使われる。
- 参照呼出し(call by reference)
  実引数として与えられた変数の記憶領域と仮引数の変数領域を同一視する。従って、呼び出された関数が直接呼出し側の変数を操作することになる。

これらの内C言語で行えるのは値呼出しのみであるが、変数の主記憶内での番地(ポインタという)を関数に引き渡すことにより参照呼出しと同等のことも行える。(例10.14)

関数実行のプロセス：

関数呼出しがあると、その処理は次のような順序で進む。

（１）各々の実引数を評価。

（２）（１）の結果を対応する仮引数のデータ型に変換

（３）（２）の結果を対応する仮引数(変数)に代入。

（１）〜（３）：(値呼出し)

（４）関数の本体を実行する。実行の途中に、

(場合1) `return;` という文に出会うと 、
制御を呼出し元に戻す。(関数値なし)

(場合2) 本体の実行が終了すると 、
制御を呼出し元に戻す。(関数値なし)

(場合3) `return` 式`;` という文に出会うと 、
式 の値を評価し、その値をその関数が本来返すべきデータ型に変換する。 そして、その結果を関数値として制御を呼出し元に戻す。

次の例題は、

①C言語においては引数結合が値呼出しによって行われていること、そして

②値呼出しを用いて参照呼出しと同等のことも行えること

を説明している。

例題10. 14 (値呼出し,参照呼出し) 次のCプログラムを実行するとどういう出力が得られるか？ 下の □□□□□ の部分に予想される出力文字列を入れよ。但し、ここでは空白は␣と明示せよ。

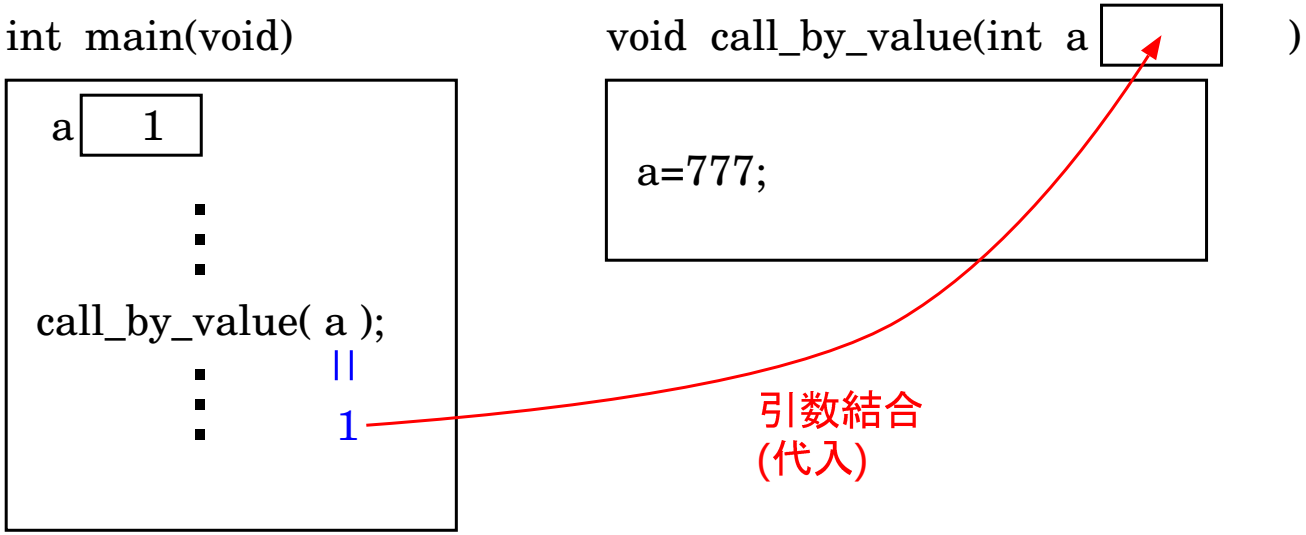
```
[motoki@x205a]$  nl func-binding-parameters.c   Enter
     1 #include <stdio.h>
     2 void call_by_value(int);
     3 void call_by_reference(int *);

     4 int main(void)
     5 {
```

```
 6   int  a=1;

 7   printf("%d\n", a);
 8   call_by_value(a);          /* 値呼出し*/
 9   printf("%d\n", a);          /* aの値は不変！*/

10   call_by_reference(&a); /* 参照呼出し*/
11   printf("%d\n", a);          /* aの値は変わる！*/
12   return 0;
13 }

14 void call_by_value(int a)
15 {
16   a = 777;
17 }

18 void call_by_reference(int *a)
19 {
20   *a = 777;
21 }
```

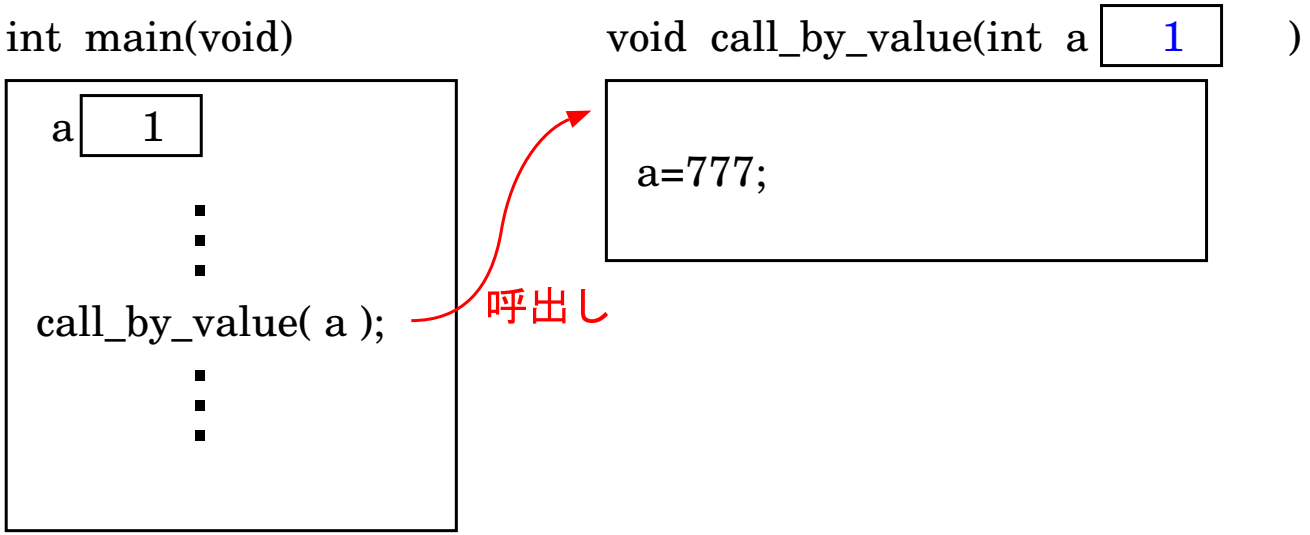
```
[motoki@x205a]$  gcc func-binding-parameters.c  Enter
[motoki@x205a]$  ./a.out  Enter
```

(考え方)　C言語では、
関数引数の結合が値呼出しによって行われる
から、

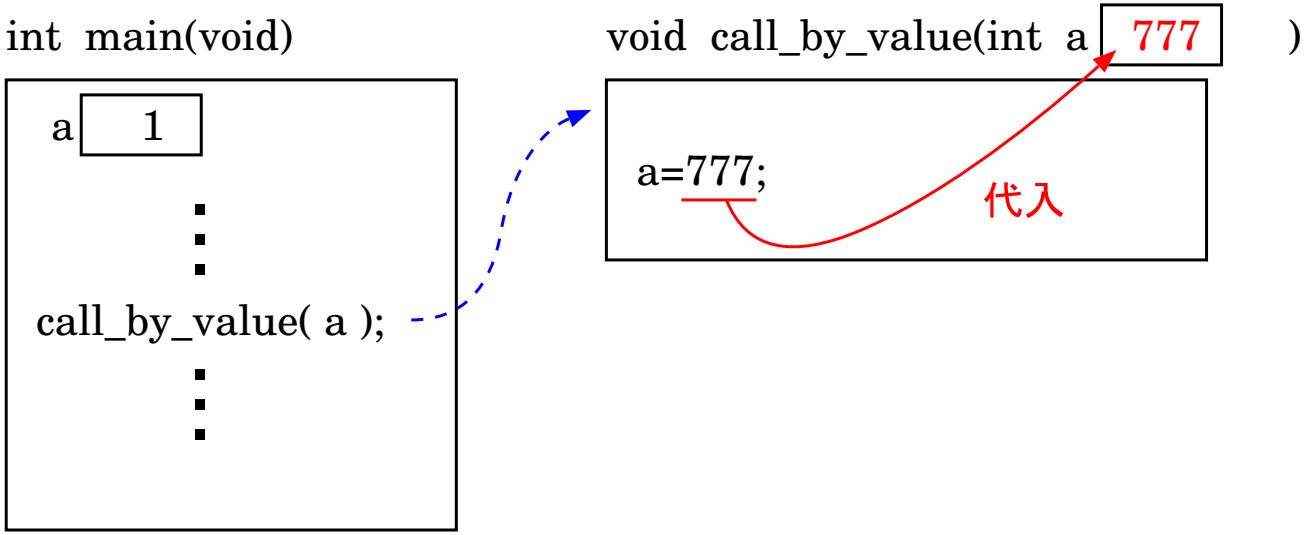
- もし実行が8行目に移りcall_by_value()が次に実行されれば、
  次の様に実行が進む。
  (8行目,引数結合)



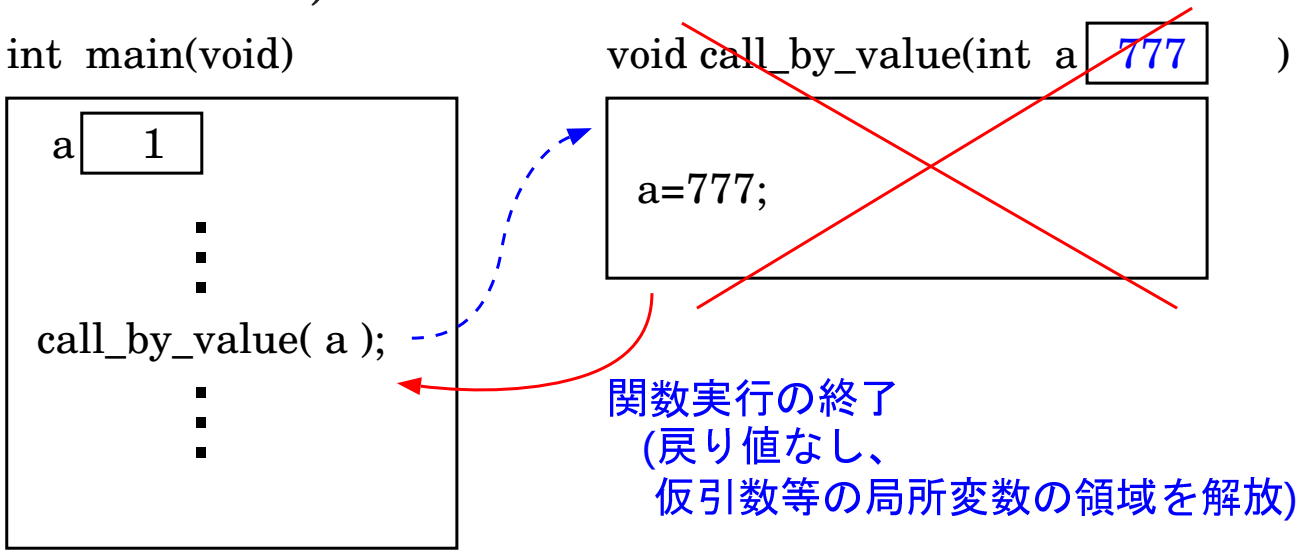
(8行目,関数呼び出し)



(16行目,実行後)



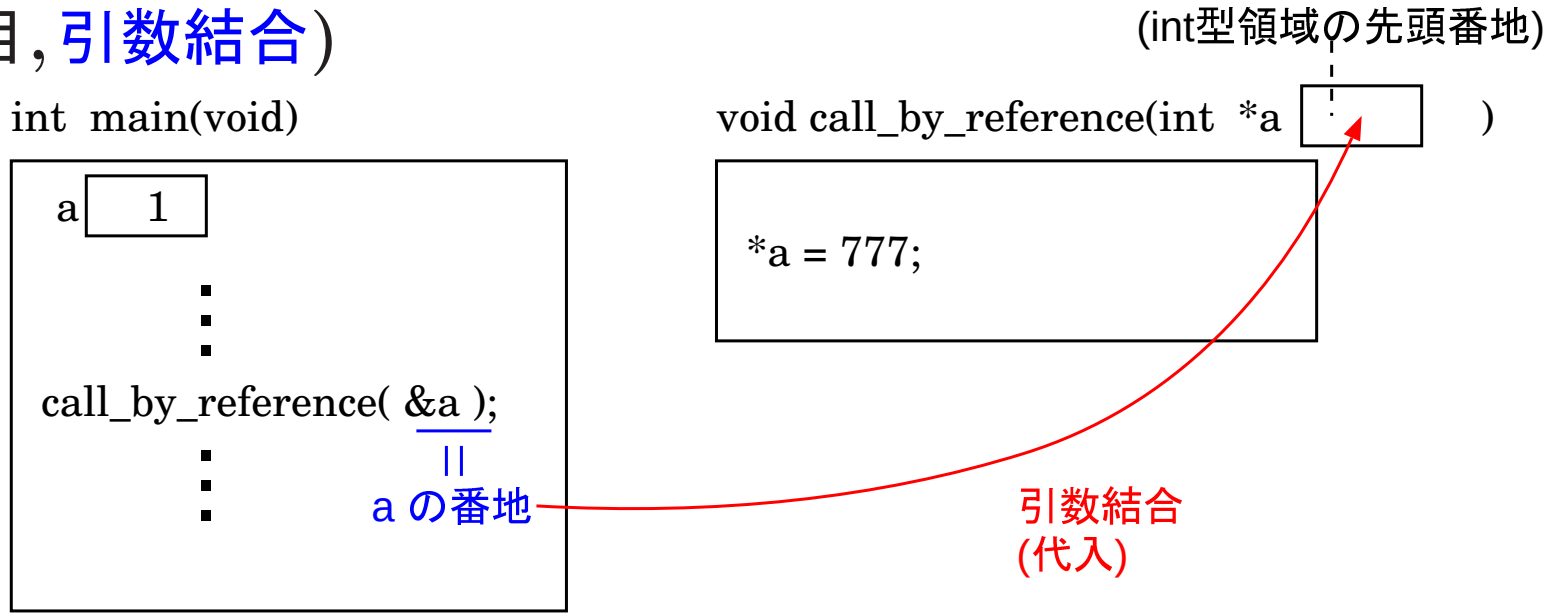
(17行目,関数実行終了)



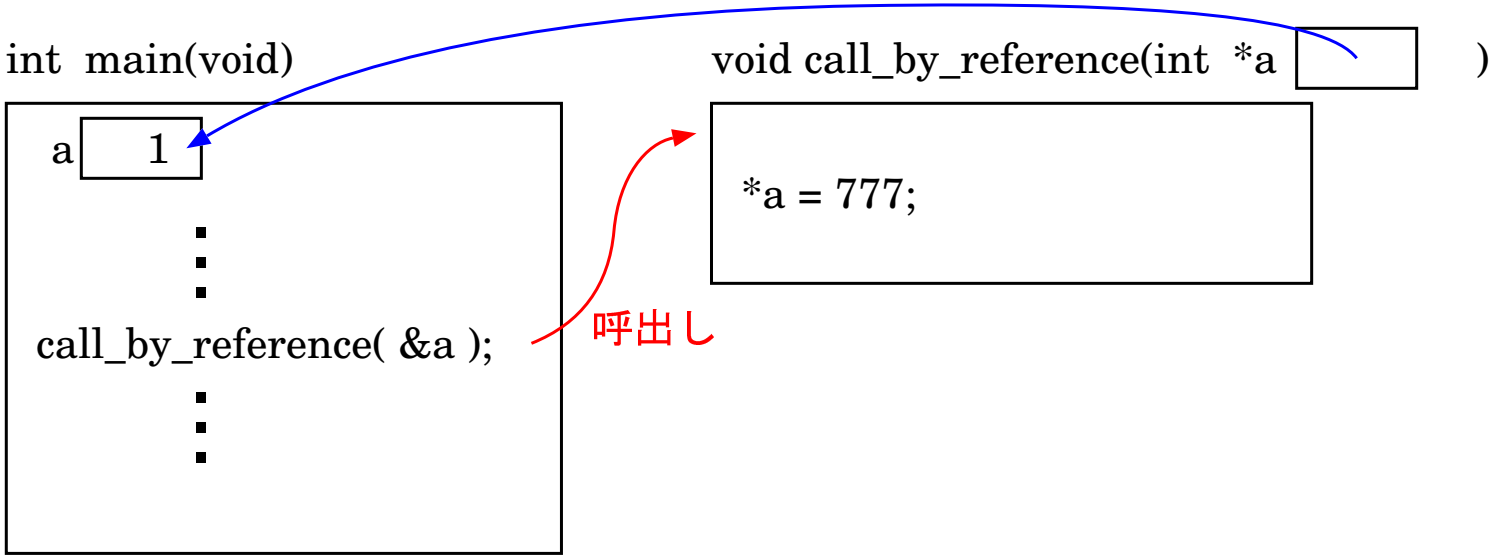
$\Longrightarrow$ 8行目の関数実行終了後も6行目の a の値は 1 のまま変わらない。

- もし実行が10行目に移りcall_by_reference()が次に実行されれば、次の様に実行が進む。

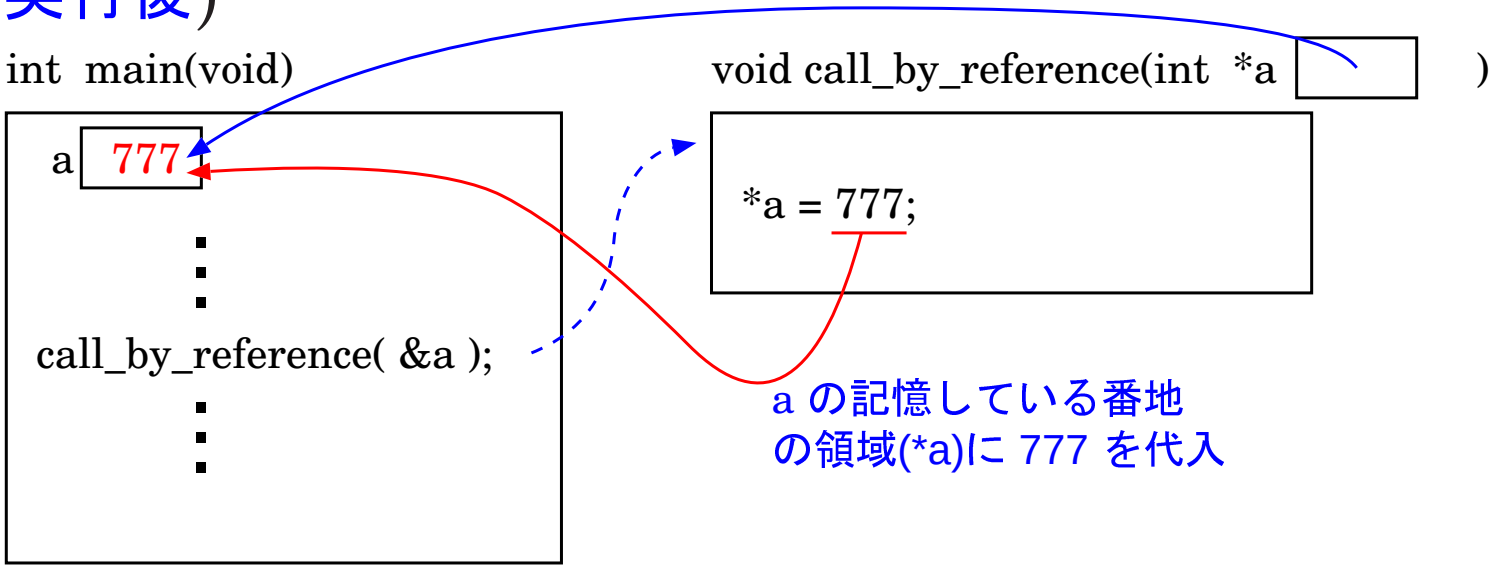
(10行目,引数結合)



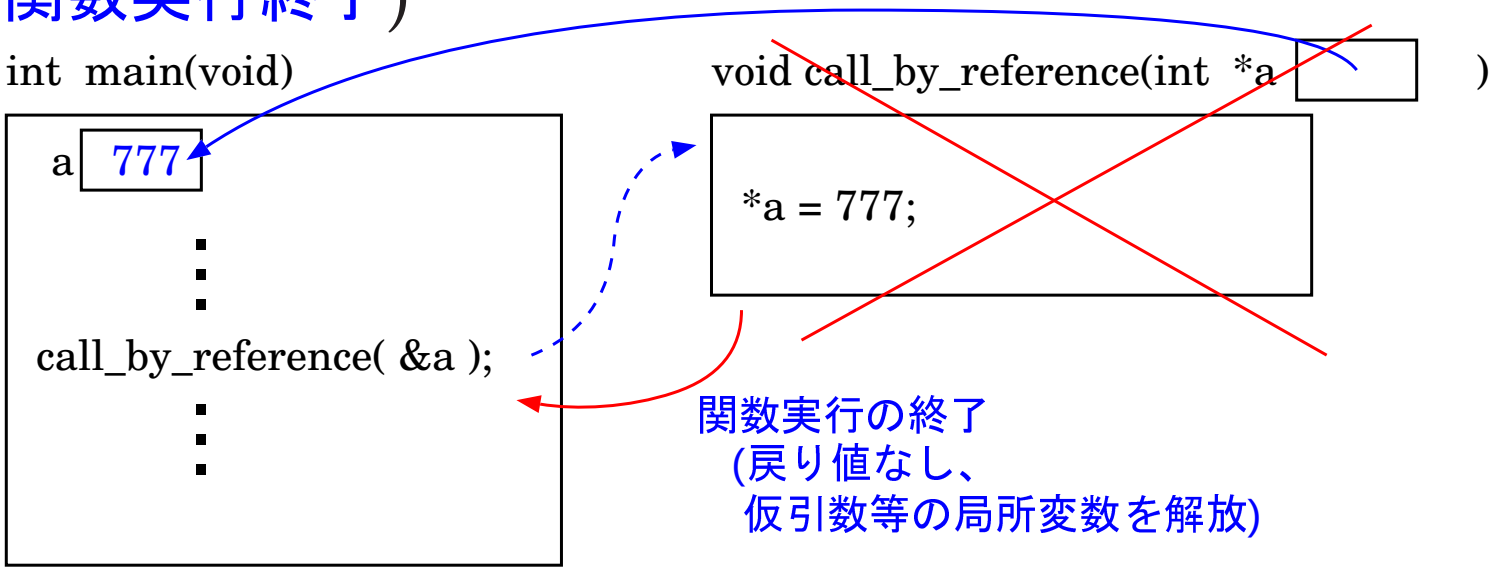
(10行目,関数呼び出し)



(20行目,実行後)



(21行目,関数実行終了)



$\Longrightarrow$ 10行目の関数実行によって6行目の a の値は 777 に変わる。

(実行結果)　結局、プログラムの

$$\begin{cases} \text{7行目では a の値は} \quad 1, \\ \text{9行目では a の値は} \quad 1, \\ \text{11行目では a の値は } 777 \end{cases}$$

になるから、実行結果は次の様になる。

```
[motoki@x205a]$  ./a.out  Enter
1
1
777
[motoki@x205a]
```

番地演算子 & と間接演算子 * :

&v ··· 変数 v へのポインタ (≈番地)。

*p ··· ポインタ p の指す記憶領域、
すなわち、p番地の記憶領域。

⇨ 変数 v があった時、*(&v) と v は同等。

参照呼出しと同等のことを行なう方法 :

- 参照呼出しの仮引数は、ポインタとして宣言する。
- 関数の本体部では、参照呼出しの仮引数は間接演算子 * を付けて使う。
- 関数を呼ぶ時、参照呼出しの実引数として変数等へのポインタ (i.e. 番地) を与える。

10–5 配列を関数パラメータとして受け渡す

配列データの受渡しを行いたい場合、配列要素毎に値呼出しによる引数結合を行っていたのでは引数結合に相当の時間がかかってしまう。

⇨ C言語では、配列データの受渡しを行いたい場合には、
その配列(の先頭要素)へのポインタを呼び出し先の関数
に引き渡す様にする。

一次元配列 a を関数の引数として受渡しする方法：

- 仮引数側では、次のいずれかの書き方をする。

  データ型　配列名 []

  データ型 *配列名

  データ型　配列名 [大きさ]

補足：
配列の大きさを明示する必要はない。
明示したとしても捨てられる。

- 関数本体の中では、仮引数の配列要素の参照はこれまでと全く同じ様な書き方をする。

- 実引数側では、次のいずれかの書き方をする。

  a

  &a[0]

補足：
配列名は、計算機内部では
先頭要素を指す定数ポインタ
として扱われている。

例題10. 16 (一次元配列の一部を関数パラメータとして受け渡す)

double型一次元配列の部分要素列についての情報を引数として受け取り、その部分配列内の要素の総和を計算して返す関数 sum() を定義せよ。 そして、

$$\mathtt{v[k]} = 2^{\mathbf{49\text{-}k}} \quad (\mathbf{k{=}0\sim\ 49})$$

という風に値の設定された大きさ**50**の**double**型一次元配列について、この関数を用いて、

v[0]+v[1]+v[2]+···+v[49],

v[40]+v[41]+v[42]+···+v[49],

v[20]+v[21]+v[22]+···+v[39]

の値を計算して出力するCプログラムを作成せよ。

(考え方)　素直に考えるなら、部分配列内の要素の総和を計算する関数 sum() には

- ①配列の名前 (i.e. 先頭要素の番地),
- ②総和の始めとなる配列要素の添字番号,
- ③総和を締めくくる配列要素の添字番号

の3つを引数として引き渡すことが頭に浮かぶ。もちろん、これは妥当な考えで、関数 sum() も使い易くなる。

しかし、呼ばれる関数側としては、受け渡されるポインタが指す領域以降に然るべき型のデータ領域が十分に長く確保されていれば良いだけである。 従って、逆に、これさえ守れば良い訳で、もし

> double型配列の名前 a と大きさ size を引数として受け取り、
> a[0]+a[1]+···+a[size-1] を計算して返す関数
> double sum(double a[], int size)

が定義できているなら、この関数を sum(&v[from], size) という風に使うことも許されるはずで、この呼び出しによって部分配列の総和 v[from]+v[from+1]+···+v[from+size-1] が計算されることになる。

確認：

配列要素 v[from]~ v[from+size-1] が sum() を呼び出す側で確保されていれば、確かに、

◇ &v[from] は配列要素を指すポインタで、

◇ &v[from] 番地以降にも同じ型のデータが十分長く続いている。

呼び出された側の関数が実際にメモリ確保された領域だけを使うようにするのはプログラマの責任である。

(プログラミング)　(部分)配列の要素の総和を計算する関数 sum() は、引数として配列の名前(先頭要素の番地) a と大きさ size を受け取り、a[0]+···+a[size-1] を計算して返すものとする。

また、main() 関数の中では、配列要素 v[0]~ v[49] に対する値の設定は数学関数 pow() を使うのではなく

$$\begin{aligned} &\texttt{v[49]} \longleftarrow \mathbf{1}, \\ &\texttt{v[48]} \longleftarrow \texttt{v[49]} \times \mathbf{2}, \\ &\texttt{v[47]} \longleftarrow \texttt{v[48]} \times \mathbf{2}, \\ &\cdots\cdots\cdots\cdots\cdots \end{aligned}$$

という風に行うことにして、プログラムを構成した。

```
[motoki@x205a]$  nl func-bind-part-of-array-Kelley.c   Enter
     1 #include <stdio.h>

     2 double sum(double a[], int size);
```

```
 3 int main(void)
 4 {
 5   int    i;
 6   double v[50];

 7   v[49] = 1.0;
 8   for (i=48; i>=0; --i)
 9     v[i] = v[i+1] * 2.0;

10   printf("v[0] +v[1] + ... +v[49] = %16.0f\n",
            sum(v, 50));
11   printf("v[40]+v[41]+ ... +v[49] = %16.0f\n",
            sum(&v[40], 10));
12   printf("v[40]+v[41]+ ... +v[49] = %16.0f\n",
            sum(v+40, 10));
13   printf("v[20]+v[21]+ ... +v[39] = %16.0f\n",
            sum(v+20, 20));
```

```
14   return 0;
15 }

16 /*-------------------------------------------
17 /* double型配列(もしくは配列の断片)の要素の総和を計算し...
18 /*-------------------------------------------
19 /*   (仮引数)     a : double型配列
20 /*                size : double型配列 a の大きさ
21 /*   (関数値) : a[0]+a[1]+a[2]+...+a[size-1]
22 /*-------------------------------------------
23 double sum(double a[], int size)
24 {
25   int    i;
26   double sum=0.0;

27   for (i=0; i < size; ++i)
28     sum += a[i];
```

```
    29   return sum;
    30 }
[motoki@x205a]$  gcc func-bind-part-of-array-Kelley.c  Enter
[motoki@x205a]$  ./a.out  Enter
v[0] +v[1] + ... +v[49] = 1125899906842623
v[40]+v[41]+ ... +v[49] =                1023
v[40]+v[41]+ ... +v[49] =                1023
v[20]+v[21]+ ... +v[39] =          1073740800
[motoki@x205a]$
```

多次元配列を関数の引数として受渡しする方法：

- 仮引数側では、1次元目を除く全ての次元の大きさを指定して、

  データ型 配列名 [] [大きさ] ･･･ [大きさ]

  または　データ型 配列名 [大きさ] ･･･ [大きさ]

  または　データ型 (*配列名) [大きさ] ･･･ [大きさ]

  という書き方をする。

2次元目以降の次元の大きさを指定する理由：

そうしないと、コンパイラが配列の添字の値からその配列要素の番地を割り出せないからである。

また、1次元目の配列の大きさについては明示する必要はない。 明示したとしても捨てられる。

例10. 19 (多次元配列を関数の仮引数とする場合) 3次元配列

int a[7][9][2] を引数として受渡したい時には、仮引数部は次のように書く。

いずれか
- int a[][9][2]
- int a[7][9][2]
- int (*a)[9][2]　←― 明示的な書き方

- 関数本体の中では、仮引数の配列要素の参照はこれまでと全く同じ様な書き方をする。

- 実引数側では、
  a または &a[0]
  という書き方をする。

例えば

aが3次元配列の場合、配列名aは計算機内部では2次元配列 a[0] を指す定数ポインタとして扱われている。

例題10. 20 (行列の積を計算する関数)　例題9.8では、3×3実数値行列 $A=[a_{ij}]$, $B=[b_{ij}]$ の要素を読み込み、これらの行列の積 AB を計算してその結果を表示するプログラムを提示した。このプログラムの中の行列の積を計算する部分を関数化し、プログラム全体を再構成してみよ。

(考え方)　計算結果 (単一の数値ではない) を関数の戻り値とするのは無理がある。

⇨ 乗算対象の a[][], b[][]　だけでなく、計算結果の格納場所 productAB[][] も関数引数として受け渡す、

```
void matrix_multiplication(double x[][3],
                           double y[][3],
                           double productXY[][3]);
```

という関数を考え、この中で次の処理を行なう。

(productXYの表す行列) ⟵ (xの表す行列) × (yの表す行列)

(プログラミング)　次の通り。

```
[motoki@x205a]$  nl matrix-multiplication-function.c  Enter
 1 /* 3×3実数値行列 A=[a_ij], B=[b_ij] の要素を読み込み、 */
 2 /* 行列の積 AB を計算してその要素を2次元状に見易く出力する  */
 3 /* Cプログラム

 4 #include <stdio.h>

 5 #define  N  (3)

 6 void matrix_multiplication(double x[][N], double y[][N],
 7                            double productXY[][N]);

 8 int main(void)
 9 {
10   double a[N][N], b[N][N], productAB[N][N];
```

```
11    int  i, j;

12    /* 行列 A,B の各要素を入力する */
13    for (i=0; i<N; i++) {
14      printf("行列 A の %d 行目の要素を順に入力して下さい:  ", i
15      for (j=0; j<N; j++)
16        scanf("%lf", &a[i][j]);
17    }
18    printf("\n");
19    for (i=0; i<N; i++) {
20      printf("行列 B の %d 行目の要素を順に入力して下さい:  ", i
21      for (j=0; j<N; j++)
22        scanf("%lf", &b[i][j]);
23    }

24    /* 行列の積 AB の各要素を計算して配列 productAB に記録する *
25    matrix_multiplication(a, b, productAB);
```

```
26    /* 行列の積 AB の結果を表示する */
27    printf("\n行列の積 AB の結果:\n");
28    for (i=0; i<N; i++) {
29      printf("    ");
30      for (j=0; j<N; j++)
31        printf("  %12.5g", productAB[i][j]);
32      printf("\n");
33    }
34    return 0;
35  }

36  /*-------------------------------------------------
37  /* 行列の積を計算
38  /*-------------------------------------------------
39  /*   (仮引数)     x : double型2次元配列
40  /*                y : double型2次元配列
```

```
41 /*        productXY : double型2次元配列, 計算結果の格納場所
42 /*    (関数値) : なし
43 /*    (機能) : (x[][]の表す行列)×(y[][]の表す行列) の計算を
44 /*                行ないその結果を productXY[][] に格納する。
45 /*-------------------------------------------------------
46 void matrix_multiplication(double x[][N], double y[][N],
47                                 double productXY[][N])
48 {
49   int i, j, k;

50   for (i=0; i<N; i++) {
51     for (j=0; j<N; j++) {
52       productXY[i][j] = 0.0;
53       for (k=0; k<N; k++)
54         productXY[i][j] += x[i][k]*y[k][j];
55     }
56   }
```

```
57 }
[motoki@x205a]$
```

10–6 段階的詳細化

複雑な仕事内容を計算機で処理する際は、
処理のモジュール化、すなわち、

> 小さなプログラムを部品 (module) として構成し、
> それらの部品を使うことによってより大きなプログラムを構築する、

というソフトウェア構築法が有効である。

補足：
プログラムの大きさが2倍になった場合、分かり難さは2倍では済まない。

モジュール化の利点

- プログラムの構造化、系統化

  ⇨ プログラムが分かり易く修正し易くなる。

  補足：
  各モジュールの仕様が明らかになっていることが大切。

- 大きなプログラムを何人かで分担して作るのに都合が良い。

- 同一の処理単位を何回も重複してプログラムに組む込む必要がなくなる。

  ⇨ プログラムの簡素化

何をモジュールと考えるか：

通常のプログラミング言語では、機能的にまとまりのあるプログラム断片を１つの関数または手続きとして定義／登録でき、また、これらの関数や手続きを必要に応じて呼び出せる様になっている。

⇨ 一般にはプログラムを設計する際に
関数や手続きをモジュールと考えるのが最も素朴で自然。

モジュール化を実際に進めるための手法としては 、
段階的詳細化、すなわち、
　　処理手順を少しずつ詳細化していくことによってプログラム／アルゴリズムを作り上げてゆく、
という考え方がよく用いられている。

このプログラム設計方針に従えば 、
処理手順は大雑把なものから (十分に) 細かなものへと少しずつ詳細化されていくことになるから、
　詳細化の途中に現われる処理単位のうち
　機能的にまとまりのあるものをモジュールに
すれば、元の大きな処理は比較的きれいに分割され、プログラムのモジュール化が自然に進むことになる。

10–7 付録 関数についてのまとめ —C文法のまとめ

C言語における関数の扱い：

- 値を返さない関数を手続きと考える。

  ⇨ 手続きを定義する手段は用意されていない。

- 関数の定義を並べたものがプログラムになる。

- 全ての関数は同一水準にある。

- プログラムの起動は関数`main`の実行で始まる。
  (`main`が主プログラム。)

- 全ての関数は、使用する前にその引数の型、関数値の型、すなわち関数プロトタイプを宣言しておかなければならない。例えば、

  ```
  double pow(double x, double y);
  ```

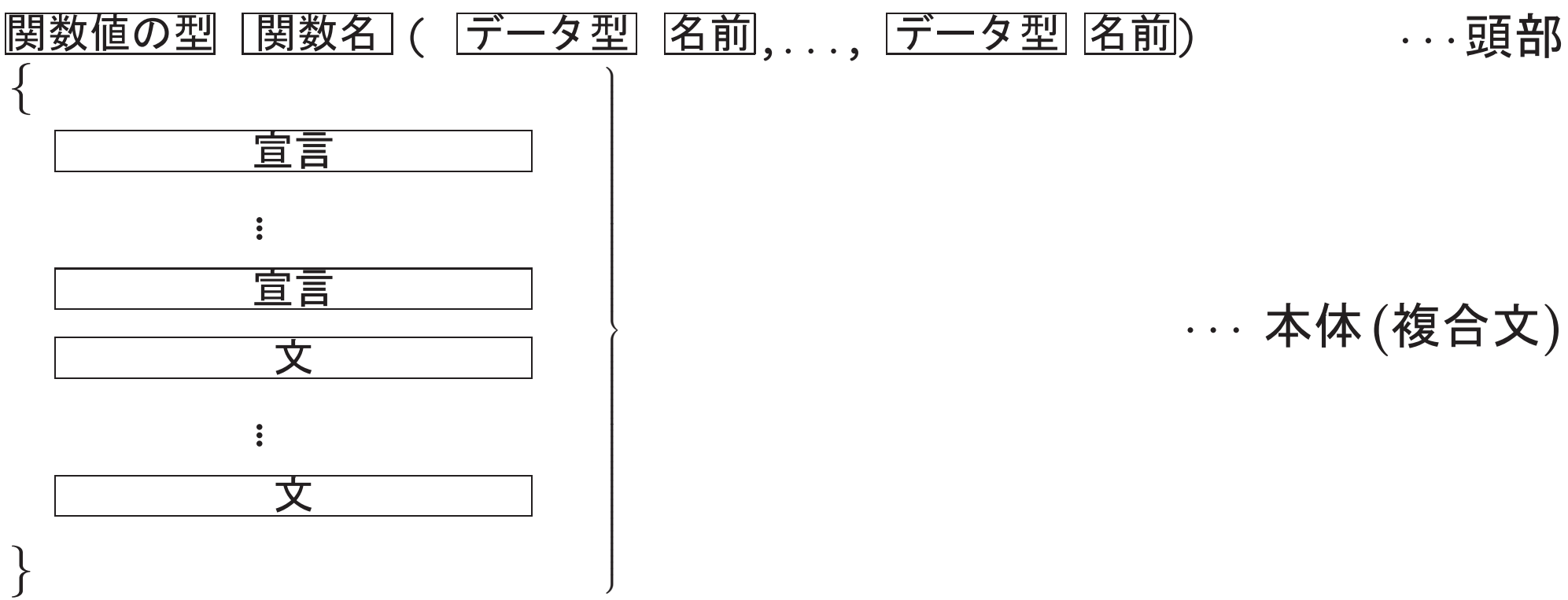
  または `double pow(double, double);`

- 標準ライブラリ関数については、関数プロトタイプの宣言は`<stdio.h>`等のヘッダファイルの中に置かれている。

- 関数呼び出しの際の引数結合は常に値呼出しで行われる。(但し、`&`演算子を用いれば、参照呼出しと同等のことも行える。)

関数定義：

- 一般形は次の通り。



- 値を返さない関数を定義する場合は 関数値の型 の部分は `void` とする。

- 関数値の型の部分を省略すると `int` が暗黙に仮定される。
  (しかし、これを当てにして省略するのは良くない。)

return文：

- 構文は次のいずれか。

```
return;
return 式 ;
```

- return文に出くわすと、その関数の実行は終了する。(呼出し元に戻る。)

- 式が指定されていると、その値(を指定されたデータ型に変換したもの)が関数値になる。

- return文に出会わないまま関数の本体部の処理が終わった場合も、その関数の実行は終了する。(当然、関数値はない。)

関数プロトタイプ :

- 構文は次のいずれか。

関数値の型 関数名 (データ型 名前 , . . . , データ型 名前) ;

または

関数値の型 関数名 (データ型 , . . . , データ型) ;

- 関数の引数の個数と型、および関数値の型をコンパイラに知らせるための宣言。

- 関数を呼び出す前に、その関数を定義するかプロトタイプを宣言しないといけない。 [この情報が分からないと、コンパイラは例えば戻って来た計算結果 (ビット列) をどう解釈してよいか分からない。]

- 標準ライブラリ関数のプロトタイプは <stdio.h>, <stdlib.h>, ... に入っている。

10–8 付録 標準ライブラリ関数のまとめ

- 標準ライブラリ関数については、関数プロトタイプの宣言は次のいずれかの標準ヘッダファイルの中に置かれている。

```
<assert.h> <limits.h> <signal.h> <stdlib.h>
<ctype.h>  <locale.h> <stdarg.h> <string.h>
<errno.h>  <math.h>   <stddef.h> <time.h>
<float.h>  <setjmp.h> <stdio.h>
```

⇨ 標準ライブラリ関数を使いたければ、
その関数のプロトタイプが入っている標準ヘッダファイル
をインクルードしなければならない。

- 標準ヘッダファイルの中には、用途別に関数プロトタイプだけでなくマクロ定義なども入っている。各々の内容は次の通り。

標準ヘッダファイル	内容
<assert.h>	プログラムが思惑通に働いているかをチェックするための、引数付きマクロの定義が入っている。講義ノート16.7節を参照。
<ctype.h>	文字の種類 (e.g. 制御文字, 印字可能文字, 数字, 小文字,...) をテストしたり変換したりするための関数のプロトタイプが入っている。
<errno.h>	ライブラリ関数がエラーを検出したとき、その報告をするのに使うマクロ等が定義されている。
<float.h>	浮動小数点数型 (e.g. float, double) の各々の特性と限界を定めるマクロ、例えば表せる正の最小値を定めたマクロ等が入っている。
<limits.h>	整数型 (e.g. char, short, int, long) の各々の特性と限界を定めるマクロ、例えば表せる最大値を定めたマクロ等が入っている。
<locale.h>	地域化処理のためのデータ型、マクロ、関数プロトタイプが入っている。
<math.h>	数学関数に関するマクロ、関数プロトタイプが入っている。講義ノート4.1節を参照。

標準ヘッダファイル	内容
<setjmp.h>	非局所的分岐: 関数の実行環境を保存したり復元したりするためのデータ型、マクロ、関数プロトタイプが入っている。
<signal.h>	実行時のエラー, 外部からの割り込みといった、実行時に起こる例外状態を処理するためのマクロ、関数プロトタイプが入っている。
<stdarg.h>	可変引数リストを持つ関数 (e.g.printf) の引数を処理するためのデータ型、マクロが定義されている。
<stddef.h>	共通に使われるデータ型、マクロの定義が入っており、その中にはコンパイラに固有のものもある。
<stdio.h>	入出力に関するデータ型、マクロ、関数プロトタイプが入っている。
<stdlib.h>	記憶域確保,疑似乱数発生,強制終了,文字列を数値に変換,など、いわゆるユーティリティ関数のプロトタイプが入っている。
<string.h>	文字列を操作するための関数のプロトタイプが入っている。
<time.h>	日付と時刻を扱うためのデータ型、マクロ、関数プロトタイプが入っている。

以下、標準ヘッダファイルの中で定義されているデータ型,マクロ,関数プロトタイプの内で、有用そうなものを簡単に紹介する。

省略。

必要になった時点で、各自で調べる。

文字種類テストの関数/引数付きマクロ <ctype.h> :

関数プロトタイプ	説明
int isalnum(int c)	cが英数字か?
int isalpha(int c)	cが英字か?
int iscntrl(int c)	cが制御文字か?
int isdigit(int c)	cが数字か?
int isgraph(int c)	cが空白以外の印字可能文字か?
int islower(int c)	cが小文字か?
int isprint(int c)	cが印字可能文字(空白も含む)か?
int ispunct(int c)	cが区切り文字か?
int isspace(int c)	cが空白類か?
int isupper(int c)	cが大文字か?
int isxdigit(int c)	cが16進数字か?

文字種類変換の関数 <ctype.h> :

関数プロトタイプ	説明
`int tolower(int c)`	c を小文字に変換
`int toupper(int c)`	c を英大文字に変換

共通に使うデータ型, マクロ <stddef.h> :

名前	説明
`ptrdiff_t`	「2つのポインタの差」を表すデータ型
`size_t`	sizeof 演算の結果を表すデータ型で、 `typedef unsigned int size_t;` と定義されている。
`NULL`	ヌルポインタを表すマクロ
`wchar_t`	「多バイト文字の番号」を表すデータ型を

入出力に関するデータ型,マクロ <stdio.h> :

名前	説明
FILE	ファイルのアクセス状況を記録した構造体のデータ型。入力用か出力用か、次の読み込み文字の位置、ファイル終端が起きたかどうか、などの情報から成る。
EOF	「ファイルの終わり」の値を表すマクロ
NULL	空ポインタを表すマクロ
stdin	標準入力を表すマクロ
stdout	標準出力を表すマクロ
stderr	標準エラー出力を表すマクロ

ファイルをオープン・クローズする関数 <stdio.h> :

関数プロトタイプ … 説明
`FILE *fopen(const char *filename, const char *mode)` … ファイルをオープンし、そのファイルポインタを返す。オープンに失敗すると NULL を返す。ここで、filenameはオープンするファイルの名前（文字列）へのポインタである。modeが"r" の時は読み込みを、"w" の時は書き出しを、"a" の時は追加書き出しを、"rb" の時はバイナリファイルの読み込みを、"r+" の時はテキストファイルを読み書き両用にオープンすることを表す。
`int fclose(FILE *fp)` … ファイルをクローズする。ここで、fpはファイルポインタ。
`FILE *freopen(const char *filename, const char *mode, FILE *f` … ファイルポインタfpに付随するファイルをクローズし、代わりに新しくファイルをオープンしfpに結びつける。

書式付き入出力の関数 <stdio.h> :

関数プロトタイプ … 説明
`int printf(const char *cntrl_string, ...)` … 標準出力への書式付き出力。講義ノート節を参照。
`int fprintf(FILE *fp, const char *cntrl_string, ...)` … 指定した出力ストリームへの書式付き出力。
`int sprintf(char *s, const char *cntrl_string, ...)` … 指定したchar型配列への書式付き出力。最後に空文字\0 も出力して、出力結果を文字列とする。
`int scanf(const char *cntrl_string, ...)` … 標準入力からの書式付き入力。講義ノート節を参照。
`int fscanf(FILE *fp, const char *cntrl_string, ...)` … 指定した入力ストリームからの書式付き入力。
`int sscanf(char *s, const char *cntrl_string, ...)` … 指定した文字列(char型配列)からの書式付き入力。 注意： 実行する度に、指定した配列の先頭から入力作業を開始する。

１文字入出力の関数 <stdio.h> :

関数プロトタイプ … 説明
int getchar(void) … 標準入力のストリームから1文字だけ(空白も可)読み込んで、その文字コードの値を返す。但し、ファイルの終りまたはエラーを検出した時はEOFを返す。
int fgetc(FILE *fp) … 指定した入力ストリームから1文字だけ(空白も可)読み込んで、その文字コードの値を返す。ファイルの終りまたはエラーを検出した時はEOFを返す。
int ungetc(int c, FILE *fp) … 指定した入力ストリームにcという文字コードを戻す。
int putchar(int c) … 標準出力ストリームに文字コードcの文字を書き出す。成功すると (int)(unsigned char)c を返し、失敗すると EOF を返す。
int fputc(int c, FILE *fp) … 指定した出力ストリームに文字コードcの文字を書き出す。

1行入出力の関数 <stdio.h> :

関数プロトタイプ … 説明
char *gets(char *s) … 標準入力ストリームから改行コード又はファイルの終りまでの文字の並びを読み込み、char型配列 s に格納する。その際、改行コードは空文字\0 に置き換えられる。通常は s が返されるが、ファイル終了又はエラー発生時には NULL が返される。 セキュリティ上の問題(バッファオーバーラン)があり使うべきでない関数とされ、2011年の言語仕様改定でC11の標準Cライブラリから廃止された。gccでは使うと警告が出るらしい。
char *fgets(char *line, int n, FILE *fp) … 指定した入力ストリームから、改行コード又はファイルの終りまでの文字の並び(但し長くなっても n-1 文字で打ち切り)を読み込み、最後に空文字 \0 を付けてchar型配列 line に格納する。通常は line の値が関数値として返されるが、ファイル終了又はエラー発生時には NULL が返される。
int puts(const char *s) … 標準出力ストリームに文字列 s を書き出す。但し、文字列の最後の空文字 \0 の代わりに改行コードを書き出す。成功すると非負の値を返し、失敗すると EOF を返す。

関数プロトタイプ … 説明
int fputs(const char *s, FILE *fp) … 指定した出力ストリームに文字列 s を書き出す。但し、文字列の最後の空文字 \0 は出力しない。[puts と違って、代わりに改行コードを書き出すこともしない。]

バイナリファイルの入出力を行う関数 <stdio.h> :

関数プロトタイプ … 説明
size_t fread(void *a_ptr, size_t el_size, size_t n, FILE *fp) … 指定した入力ストリームから、1要素el_sizeバイトのデータをn個(但しファイル終了になるとそこまで)、a_ptr が指す配列に格納する。関数値は読み込んだ要素数である。
size_t fwrite(const void *a_ptr, size_t el_size, size_t n, FILE … a_ptr が指す配列から、1要素当たりel_sizeバイトのデータをn個取り出し、指定した出力ストリームに書き出す。関数値は書き出しに成功した要素数である。

ファイルの読み込み位置/書き込み位置を設定する関数 <stdio.h> :

- オープンしたファイルは、通常、前から順に処理しますが、ファイルの先頭や末尾からの距離を指定して、(原理的には) 任意の場所にアクセスすることが出来る。また、現在見ている場所 (先頭からのバイト数) を知ることも出来る。

- 内部的には、ファイル中の現在処理している場所は、ファイル位置指示子と呼ばれる記憶領域の中に記録される。　これはファイルポインタの指すFILE型構造体のメンバで、通常は、この値がファイルの先頭場所から始まって少しずつ大きくなる。

関数プロトタイプ … 説明
int fseek(FILE *fp, long offset, int place) … ファイル位置指示子の値を place から offset バイト離れた所に設定する。ここで、place としては SEEK_SET (ファイルの先頭を表す;通常0), SEEK_CUR (現在位置を表す;通常1), SEEK_END (ファイルの末尾を表す;通常2) のいずれかを指定する。成功すると 0 を返し、失敗すると 0以外の値 を返す。
void rewind(FILE *fp) … ファイル位置指示子をファイルの先頭に設定する。
long ftell(FILE *fp) … ファイル位置指示子の現在の値(先頭からのバイト数)を返す。但し、エラーを検出した時は -1 を返す。

一時ファイルをオープンする関数 <stdio.h> :

関数プロトタイプ ··· 説明
FILE *tmpfile(void) ··· 一時的な使用目的のための(バイナリ)ファイルを "wb+" という利用モードでオープンし、そのファイルポインタを返す。オープンに失敗すると NULL を返す。この一時ファイルは、クローズまたはプログラム終了時に削除される。

入出力に関するその他の関数 <stdio.h> :

関数プロトタイプ … 説明
`int fflush(FILE *fp)` … fpで指定したストリームが出力用の時、そのストリーム向けに溜ったバッファデータを実際にストリームに吐き出す。
`int feof(FILE *fp)` … 指定したストリームにファイル終了の標識が立っているかどうかを調べ、立っていれば0以外、立っていなければ0を返す。
`int remove(const char *filename)` … 指定したファイルを削除する。
`int rename(const char *from, const char *to)` … ファイルの名前を変更する。

記憶域を動的に確保する関数 <stdlib.h> :

関数プロトタイプ … 説明
`void *malloc(size_t size)` … sizeバイトの記憶域を(ヒープ領域から)確保し、その先頭へのポインタを返す。記憶域確保に失敗すれば空ポインタNULLを返す。
`void *realloc(void *ptr, size_t size)` … ptrの指す記憶域の内容を保存したまま、その大きさをsizeに変更する。成功すれば、変更後の記憶域の先頭へのポインタを返し、失敗すれば空ポインタNULLを返す。
`void *calloc(size_t n, size_t el_size)` … 1要素がel_sizeバイトで要素数がn個の配列のための連続領域を(ヒープ領域から)確保し、全てのビットを0にクリアした後、その先頭へのポインタを返す。失敗すれば空ポインタNULLを返す。
`void free(void *ptr)` … ptrが指す記憶域を解放する。ptrがNULLの時は何も起きない。

疑似乱数発生のためのマクロ,関数 <stdlib.h> :

関数プロトタイプ/マクロ名 ··· 説明
RAND_MAX ··· 関数rand()が返すint型疑似乱数の最大値を表すマクロ
int rand(void) ··· 区間$[0, \mathrm{RAND_MAX}]$の間の疑似整数乱数を返す。
int srand(unsigned seed) ··· 関数rand()の生成する疑似乱数の種を seed に設定する。デフォルトでは seed=1 である。

プログラムを強制終了するためのマクロ,関数 <stdlib.h> :

関数プロトタイプ/マクロ名 ··· 説明
void exit(int status) ··· プログラムを正常終了させ、status を主ルーチンの関数値として呼び出し元(OS)に返す。呼び出し元は、status=0 の時にプログラムが正常終了したと判断する。
EXIT_SUCCESS ··· 関数exit()の引数として使うマクロで、通常は 0 と定義されている。成功終了を表す。
EXIT_FAILURE ··· 関数exit()の引数として使うマクロで、通常は 1 と定義されている。異常終了を表す。

環境変数へのアクセス,OSコマンド実行ための関数 <stdlib.h> :

関数プロトタイプ ··· 説明
`char *getenv(const char *name)` ··· 指定した環境変数の値(文字列)へのポインタを返す。
`int system(const char *s)` ··· 指定したコマンドをOSが提供するコマンドインタープリタに実行してもらう。

文字列を数値に変換するための関数 <stdlib.h> :

関数プロトタイプ … 説明
`double atof(const char *s)` … sの指す文字列を実数と見て、それをdouble型の内部表現形式に変換して返す。
`int atoi(const char *s)` … sの指す文字列を整数と見て、それをint型の内部表現形式に変換して返す。
`int atol(const char *s)` … sの指す文字列を整数と見て、それをlong int型の内部表現形式に変換して返す。

検索, 整列のための関数 <stdlib.h> :

関数プロトタイプ … 説明

```
void *bsearch(const void *key_ptr, const void *a_ptr, size_t n
        size_t el_size, int (*compar)(const void *, const void
```

… 昇順に並んだ1次元配列の中から key_ptr が指すものと等しい要素を探し出し、そこへのポインタを返す。見つからなければ NULL を返す。ここで、a_ptr は昇順に並んだ1次元配列(の先頭要素)を指すポインタ、n は配列の大きさ、el_size は配列要素1個の占めるバイト数、compar は2つの要素の大小を判定する関数(比較関数という)へのポインタである。比較関数の2つの引数は大小を比較する要素へのポインタであり、これらの引数を基に比較関数は

(第1引数の指す要素) < (第2引数の指す要素) なら 負、
(第1引数の指す要素) = (第2引数の指す要素) なら 零、
(第1引数の指す要素) > (第2引数の指す要素) なら 正

の値を返す。データ型の所で指定されたconstは引数の値が変えられないことを宣言している。また、引数の型として指定されている (void *) は総称的なポインタ型で、この型のポインタはどんなポインタ変数にも代入可能である。

(続く)

関数プロトタイプ … 説明

`void *qsort(const void *a_ptr, size_t n, size_t el_size,`
`int (*compar)(const void *, const void *) )`
… 1次元配列の要素を比較関数 compar の基準に従って昇順に並べ換える。ここで、a_ptr は昇順に並んだ1次元配列(の先頭要素)を指すポインタ、n は配列の大きさ、el_size は配列要素1個の占めるバイト数である。

整数の絶対値,商と剰余のペアを求める関数 <stdlib.h> :

関数プロトタイプ/データ型 ··· 説明
div_t ··· 関数div()が返す構造体(int型のペア)のデータ型名
ldiv_t ··· 関数ldiv()が返す構造体(long int型のペア)のデータ型名
int abs(int i) ··· i の絶対値(int型)を返す。
long labs(long i) ··· i の絶対値(long int型)を返す。
div_t div(int number, int denom) ··· number を denom で割った時の商と剰余の組を返す。
ldiv_t ldiv(long number, long denom) ··· number を denom で割った時の商と剰余の組を返す。

文字列の長さを測るための関数 <string.h> :

関数プロトタイプ ··· 説明
`size_t strlen(const char *s)` ··· 文字列 s の長さを返す。

文字列の連接,コピーをするための関数 <string.h> :

関数プロトタイプ … 説明
`char *strcat(char *s1, const char *s2)` … 文字列 s2 を文字列 s1 の末尾にコピーし、s1 の値を返す。
`char *strncat(char *s1, const char *s2, size_t n)` … 文字列 s2 を文字列 s1 の末尾にコピーし、s1 の値を返す。但し、s2 の長さが n を越える場合は最初のn文字だけをコピーし、その後に空文字\0 を追加する。
`char *strcpy(char *s1, const char *s2)` … 文字列 s2 を末尾の空文字 \0 も含めてs1 の指す領域にコピーし、s1 の値を返す。
`char *strncpy(char *s1, const char *s2, size_t n)` … 文字列 s2 をs1 の指す領域にコピーし、s1 の値を返す。但し、s2 の長さがn 以上の時は最初のn文字だけをコピーする。(空文字 \0 は追加しない。) s2 の長さが n 未満の時はn-(s2の長さ)個の空文字をコピーの末尾に埋めておく。

2つの文字列を比較するための関数 <string.h> :

関数プロトタイプ ··· 説明
`int strcmp(const char *s1, const char *s2)` ··· 2つの文字列 s1 と s2 を辞書式順序で比較する。その結果、s1 が s2 より小さければ負、等しければゼロ、大きければ正の値を返す。
`int strncmp(const char *s1, const char *s2, size_t n)` ··· 2つの文字列 s1 と s2 の最初の n 文字の部分を辞書式順序で比較する。その結果、s1 が s2 より小さければ負、等しければゼロ、大きければ正の値を返す。

文字列の中で文字を探索する関数 <string.h> :

関数プロトタイプ … 説明
`char *strchr(const char *s, int c)` … 文字(コード) c を文字列 s の最初から探す。見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。
`char *strrchr(const char *s, int c)` … 文字(コード) c を文字列 s の最後から逆向きに探す。見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。
`char *strpbrk(char *s1, const char *s2)` … 文字列 s2 内に含まれる文字を文字列 s1 の最初から探す。 見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。

文字列を探索する関数 <string.h> :

関数プロトタイプ … 説明
`char *strstr(char *s1, const char *s2)` … 文字列パターン s2 を文字列 s1 の最初から探す。見つかればその最初の部分文字列の先頭位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。
`size_t strspn(char *s1, const char *s2)` … 文字列 s1 の先頭からの部分文字列で、文字列 s2 内に含まれる文字だけで構成される部分の長さを返す。
`size_t strcspn(char *s1, const char *s2)` … 文字列 s1 の先頭からの部分文字列で、文字列 s2 内に含まれない文字だけで構成される部分の長さを返す。
`char *strtok(char *s1, const char *s2)` … 文字列 s2 内の各文字を区切り記号と見て文字列 s1 を走査し、s1 の中に現れる字句 (i.e. 区切り記号以外から成る文字の並び) を探す。字句が見つかればその直後の文字が空文字に書き換えられた上でその字句の先頭位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。引き続き、s2 を空ポインタにしてこの関数が呼び出されると、前回の走査の続きの位置から走査が始まる。

バイト列をコピーするための関数 <string.h> :

関数プロトタイプ … 説明
`void *memcpy(void *to, const void *from, size_t n)` … from の指す長さ n のバイト列 (i.e. 文字の並び; 空文字\0が最後に来る保証はない) を to の指す領域にコピーし、to の値を返す。領域が重なっている場合の動作は定義されない。
`void *memmove(void *to, const void *from, size_t n)` … from の指す長さ n のバイト列 (i.e. 文字の並び; 空文字\0が最後に来る保証はない) を to の指す領域にコピーし、to の値を返す。領域が重なっていても正しくコピーされる。
`void *memset(void *p, int c, size_t n)` … p の指す領域に1バイトデータ (unsigned char)c を続けて n 個格納し、p の値を返す。

2つのバイト列を比較するための関数 <string.h> :

関数プロトタイプ … 説明
`int memcmp(const void *p1, const void *p2, size_t n)` … p1 と p2 の指す2つのバイト列の最初の n バイトの部分を辞書式順序で比較する。その結果、p1の方が p2 のバイト列より小さければ負、等しければゼロ、大きければ正の値を返す。

バイト列の中で文字を探索する関数 <string.h> :

関数プロトタイプ … 説明
`void *memchr(const void *p, int c, size_t n)` … バイトデータ (unsigned char)c を p の指すバイト列の最初から高々 n バイト探す。見つかればその最初の位置へのポインタを返し、見つからなければ空ポインタ NULL を返す。

日付と時間に関するデータ型,マクロ <time.h> :

名前	説明
CLOCKS_PER_SEC	関数clock()の想定している１秒当たりのクロック数を表すマクロ。
clock_t	各々の計算機で独自に設定されている時間量(クロック数) 表すためのデータ型で、clock()の返す関数値もこのデータ型を持つ。
time_t	暦上の日付,時刻を表すためのデータ型で、time()の返す関数値もこのデータ型を持つ。
struct tm	日付と時間の情報をまとめた構造体

時間計測の関数 <time.h> :

関数プロトタイプ ··· 説明
`clock_t clock(void)` ··· プログラム実行のためにそれまでにプロセッサを使用した時間 (クロック数) を返す。
`double difftime(time_t t2, time_t t1)` ··· 2つのカレンダ時刻 `t2` と `t1` の差 `t2-t1` を計算し、それに相当する秒単位の時間を **double** 型で返す。

現在の時刻を知るための関数 <time.h> :

関数プロトタイプ … 説明
`time_t time(time_t *tp)` … 現在のカレンダー時間として、1970年1月1日0時0分0秒からの経過秒数を返す。
`struct tm *localtime(const time_t *t_ptr)` … t_ptrが指すカレンダ時間をローカル時間に変換し、その時間に相当する struct tm 型データへのポインタを返す。
`char *asctime(const struct tm *tp)` … tpが指す struct tm 型データを例えば Sun Feb 24 17:30:27 2002 といった文字列に変換し、その文字列へのポインタを返す。
`char *ctime(const time_t *t_ptr)` … asctime(localtime(t_ptr)) と同等。すなわち、t_ptrが指すカレンダ時間をローカルな時刻を表す Sun Feb 24 17:30:27 2002 といった文字列に変換し、その文字列へのポインタを返す。

(続く)

関数プロトタイプ … 説明

```
size_t strftime(char *s, size_t n,
                const char *format, const struct tm *tp)
```

… tp が指す struct tm 型時刻データを format が指す書式に従って変換し、得られた文字列を s が指す領域に格納する。但し、n 文字を越えた文字列が得られた場合は最初の n 文字だけを格納する。関数値は、格納された文字の長さである。