4 復習 関数（その1）

4–1 数学的関数の利用

- C言語では、数学的関数は標準ライブラリの中で提供されている。

  ⇨ 数学的関数を使いたければ 、

  ◇ Cプログラムの最初に `#include <math.h>` の宣言をする。

  ◇ コンパイル時には `-lm` オプションを（最後に）付ける。

- 数学的関数の引数、関数値はほとんどが`double`型。

例題4．1（三角関数の表）　**x=0°, 5°, 10°, . . . , 90° に対して sin x, cos x, tan x の値を計算して表の形に見易く出力するCプログラムを作成せよ。**

```
[motoki@x205a]$  nl function-sin-cos-tan.c
     1  /* x=0 deg., 5 deg., 10 deg., ... , 90 deg.    */
     2  /* に対して sin x, cos x, tan x の値を計算して */
     3  /* 表の形に出力するCプログラム                 */

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

     6  #define  PI  (3.1415926535897932)   /* 円周率 */

     7  int main(void)
     8  {
     9    int    x;
```

```
    10    double x_radian, sin_x, cos_x, tan_x;

    11    printf("x(degree)    sin(x)      cos(x)           tan
    12           "-----------------------------------------
    13    for (x=0; x<=90; x+=5) {
    14      x_radian = (double)x * PI / 180.0;
    15      sin_x = sin(x_radian);
    16      cos_x = cos(x_radian);
    17      tan_x = sin_x / cos_x;
    18      printf("%6d     %10.8f  %10.8f  %15.8g\n",
    19             x, sin_x, cos_x, tan_x);
    20    }
    21    return 0;
    22  }
[motoki@x205a]$  gcc function-sin-cos-tan.c -lm
[motoki@x205a]$  ./a.out
x(degree)    sin(x)      cos(x)           tan(x)
```

0	0.00000000	1.00000000	0
5	0.08715574	0.99619470	0.087488664
10	0.17364818	0.98480775	0.17632698
15	0.25881905	0.96592583	0.26794919
20	0.34202014	0.93969262	0.36397023
25	0.42261826	0.90630779	0.46630766
30	0.50000000	0.86602540	0.57735027
35	0.57357644	0.81915204	0.70020754
40	0.64278761	0.76604444	0.83909963
45	0.70710678	0.70710678	1
50	0.76604444	0.64278761	1.1917536
55	0.81915204	0.57357644	1.428148
60	0.86602540	0.50000000	1.7320508
65	0.90630779	0.42261826	2.1445069
70	0.93969262	0.34202014	2.7474774
75	0.96592583	0.25881905	3.7320508

```
    80      0.98480775   0.17364818          5.6712818
    85      0.99619470   0.08715574          11.430052
    90      1.00000000   0.00000000     1.6331239e+16
[motoki@x205a]$
```

-lmオプションを付けないと：

$\Longrightarrow$ 次の様にコンパイルエラーになる。

```
[motoki@x205a]$ gcc throw-a-ball.c
/tmp/cc4TDjRQ.o: In function ‘main’:
/tmp/cc4TDjRQ.o(.text+0x83): undefined reference to ‘sin’
/tmp/cc4TDjRQ.o(.text+0xc9): undefined reference to ‘sin’
collect2: ld returned 1 exit status
[motoki@x205a]$
```

注目点：

- 数学的関数を使う場合、

  `#include <math.h>` という行は数学的関数を呼び出す部分を間違いなく翻訳するために必要となり、

  cc コマンドの`-lm`オプションは数学的関数の翻訳コードも取り込んで完全な実行コードを作るために必要となる。

C言語においては、次のような数学的関数が標準ライブラリに用意されている。

機能	関数名（引数の並び）	引数の型	関数値の型	説明
切捨て	floor(a)	double	double	$\lfloor$ a $\rfloor$
切上げ	ceil(a)	double	double	$\lceil$ a $\rceil$
剰余	fmod(a, b)	double	double	a≥0の時は a-|b|×$\lfloor$ a/|b|$\rfloor$
				a<0の時は a-|b|×$\lceil$ a/|b|$\rceil$
絶対値	fabs(a)	double	double	|a|
平方根	sqrt(a)	double	double	$\sqrt{a}$
べき乗	pow(a, b)	double	double	a^b
	ldexp(a, n)	doubleとint	double	$a \times 2^n$
指数	exp(a)	double	double	e^a
自然対数	log(a)	double	double	$\log_e a$
常用対数	log10(a)	double	double	$\log_{10} a$

機能	関数名（引数の並び）	引数の型	関数値の型	説明
正弦	sin(a)	double	double	sin a，但し aはラジアン
余弦	cos(a)	double	double	cos a，但し aはラジアン
正接	tan(a)	double	double	tan a，但し aはラジアン
逆正弦	asin(a)	double	double	$\sin^{-1}$a $\in [-\pi/2, \pi/2]$
逆余弦	acos(a)	double	double	$\cos^{-1}$a $\in [0,\pi]$
逆正接	atan(a)	double	double	$\tan^{-1}$a $\in [-\pi/2,\pi/2]$
	atan2(a, b)	double	double	$\tan^{-1}$(a/b) $\in [-\pi/2,\pi/2]$
双曲線正弦	sinh(a)	double	double	sinh a
双曲線余弦	cosh(a)	double	double	cosh a
双曲線正接	tanh(a)	double	double	tanh a
整数部と小数部に分離	modf(a, ptr)	doubleと (double *)	double	aの小数部（符号はaと同じ）を返し、aの整数部を ptr の指す領域に格納
仮数部と指数部に分離	frexp(a, ptr)	doubleと (int *)	double	関数呼び出し直後は a＝（関数値）$\times$ 2(ptr の指すint型の値)

補足：

C言語では、数学的関数には分類されていないが次のような関数も標準ライブラリに用意されている。

機能	関数名(引数の並び)	引数の型	関数値の型	説明
乱数	rand()	なし	int	[0, RAND_MAX)の間の疑似乱数
	srand()	unsigned	なし	乱数発生器の状態を初期化
絶対値	abs(a)	int	int	|a|
	labs(a)	long	long	|a|
商と剰余	div(a,b)	int	div_t	aをbで割った時の商と剰余の組
	ldiv(a,b)	long	ldiv_t	aをbで割った時の商と剰余の組

ここで、RAND_MAX は /usr/include/stdlib.h の中で定義されたマクロ名、div_t と ldiv_t は /usr/include/stdlib.h の中で定義された「構造体」の名前である。

4–2 関数定義

例題4．2（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1.3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;   ←21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)));
                  ↑              ↑              ↑
                実引数
14   return 0;
15 }
```

```
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)   仮引数, 7行目のkとは別領域
23 {
24   int    i;     自動変数
25   double fact;

26   fact = 1.0;
27   for (i=2; i<=k; ++i)
28     fact *= (double)i;
29   return fact;
30 }
```

```
[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。

4–3 付録 関数の基本についてのまとめ ---C文法の

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);
```

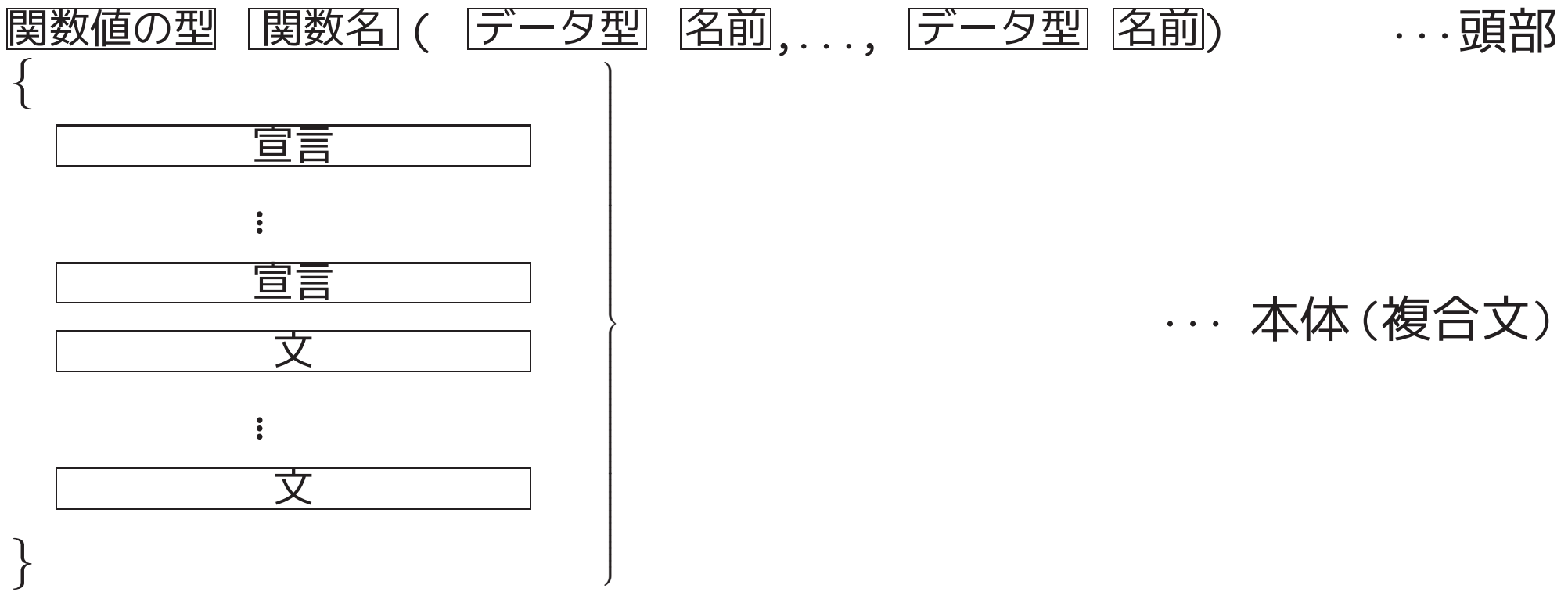
または `double pow(double, double);`

- 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。

4–4 どのようにコンパイル作業が進むのか？

この講義ノートの2.5節でも触れられている様に、ccコマンド／gccコマンドによるCプログラムの翻訳作業は実際には次の順に行われる。

① 前処理（#includeや#defineで始まる行の処理等、すなわちヘッダファイルの取り込み，マクロの展開，注釈の除去など。）

② コンパイル（各々の関数定義を機械語に翻訳，場合によっては最適化も行う。）

③ リンク（各々の関数の翻訳コードを繋げて1つの実行コードを作る。）

これらの作業の様子を図示すると図3の様になる。

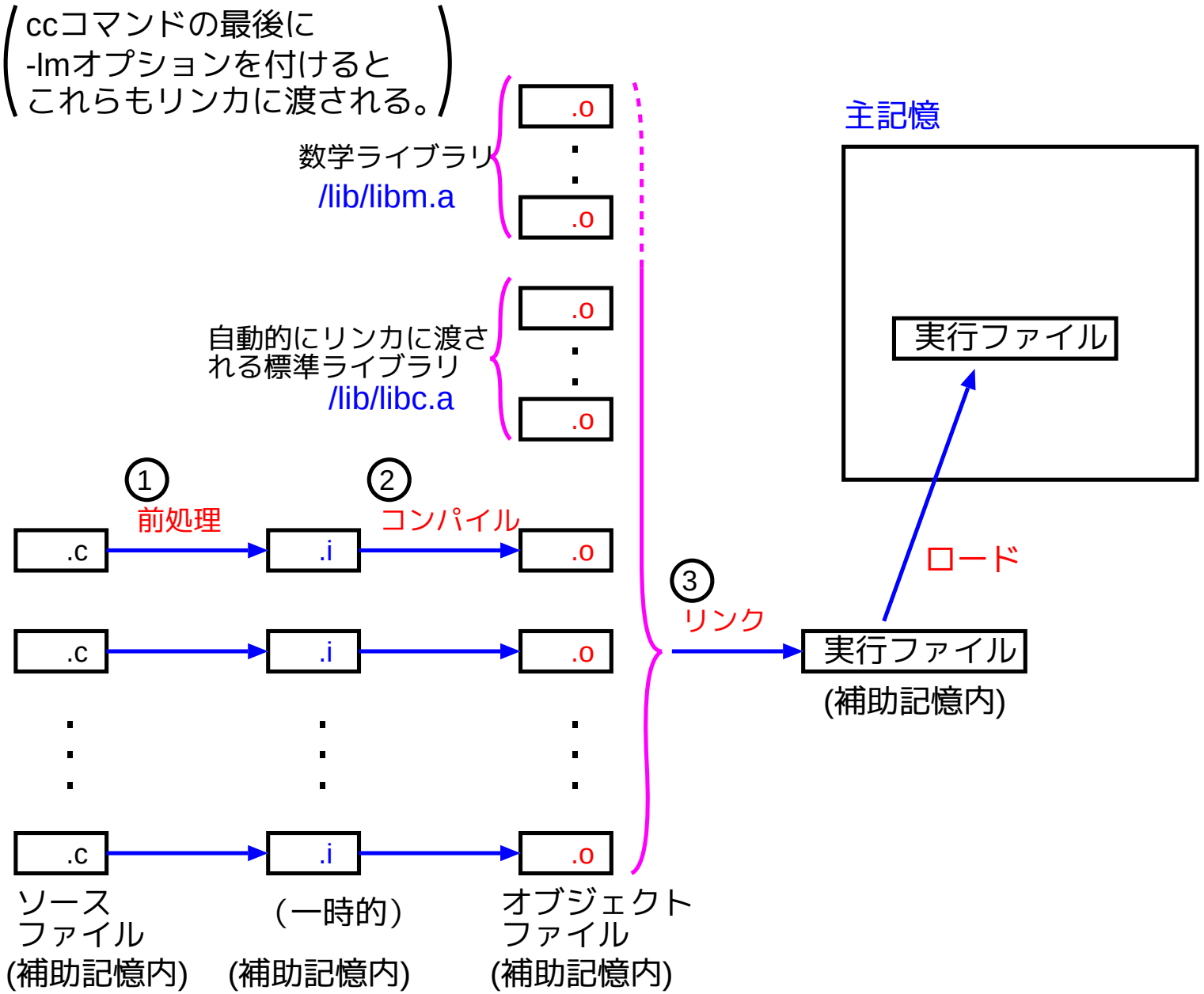


図 3: コンパイル作業の流れ

4–5 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

　⋮

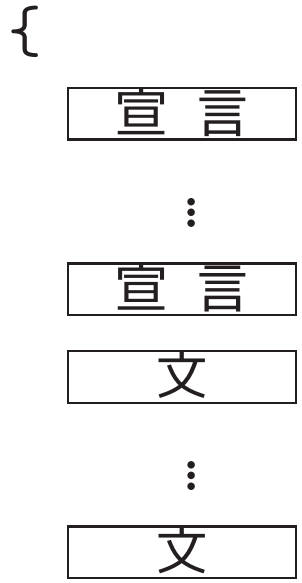
　文

}　　... 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。



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。

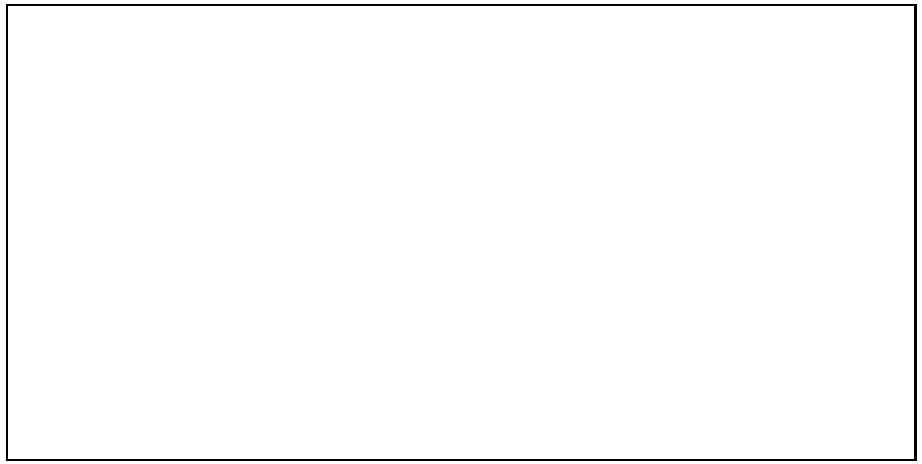
次の例題は、c言語における名前の有効範囲の規則を例示するものである。

例題4．3（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

- 8行目の a は 7行目で確保された a として、
- 11行目の a は 10行目で確保された a として、
- 13行目の a は 7行目で確保された a として、
- 19行目の a は 4行目で確保された a として、
- 20行目の b は 18行目で確保された b として

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]
```

4–6 再帰

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4．4（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4.2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4.2の場合と変わらない。

⇨ main()関数については例題4.2のものをそのまま流用できる。

⇨ 例題4.2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 }
```

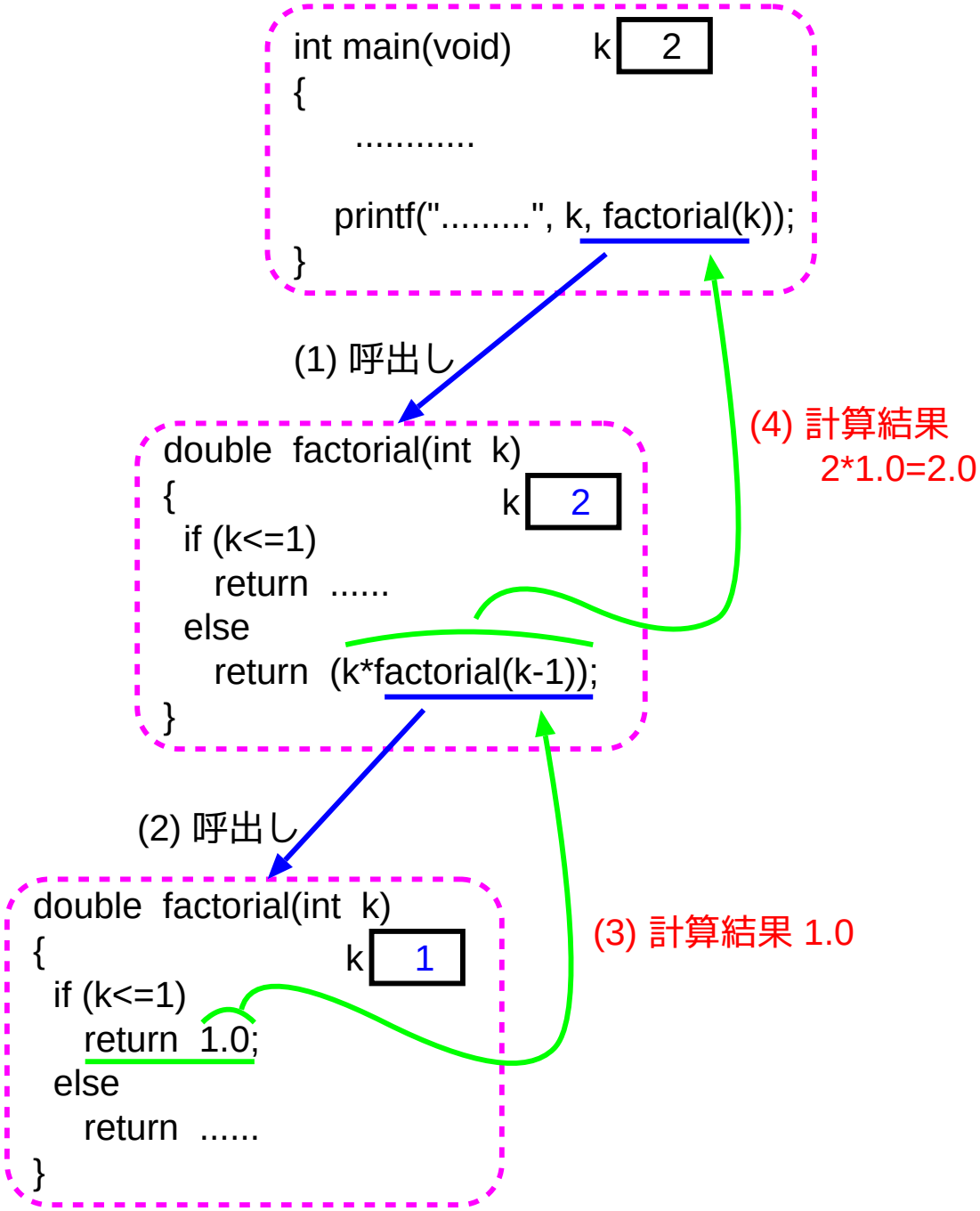
```
17 /*-------------------------------------------------
18 /* 階乗値を計算してその結果を返す関数（再帰版）          */
19 /*-------------------------------------------------
20 /*   （入力引数）k : 何の階乗を計算するかを表す整数      */
21 /*   （関数値）     : k! の値をdoubleで                 *
22 /*-------------------------------------------------
23 double factorial(int k)
24 {
25   if (k <= 1)
26     return 1.0;
27   else
28     return (k * factorial(k-1));
29 }
```

再帰呼び出し

```
[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);
}
```

□演習4．14（McCarthyの91関数）次の漸化式によって定義される整数から整数への関数 $f(x)$ を計算する関数(プログラム)を再帰的に定義してみよ。また、この関数(プログラム)を再帰無しで定義してみよ。
［ 補足： 実は$x>100$の時には $f(x)=x\text{-}10$, $x\leq 100$の時には $f(x)=91$ となる。］

$$f(x) = \begin{cases} x-10 & \text{if } x>100 \\ f(f(x+11)) & \text{otherwise} \end{cases}$$

例題4．5（クイック整列法） 大きさ100のint型配列にランダムに整数を生成し、それらの配列要素を小さい順に並べ替えて出力するCプログラムを作成せよ。

(考え方)　行うべき処理は次の3つの独立した作業から成る。

- 大きさ100のint型配列にランダムに整数を生成する作業。
- 与えられた配列内の要素を小さい順に並べ替える作業。
- 与えられた配列内の要素を順に出力する作業。

それゆえ、これらの作業を行う関数をそれぞれ別個に作り、main()関数からこれらの関数を順に呼び出すことにする。

配列内にランダムに整数を生成する作業はどの様に行えば良いのか？

この例題の場合、良質の(疑似)乱数を生成することが求められている訳ではないので、標準ライブラリ関数の int rand(void) を用いれば十分であろう。
(⟹ p.121を参照。)

補足：

疑似乱数を用いて解を探索したりシミュレーションしたりする場合は、int rand(void) の様な安易な疑似乱数を用いたのでは実験結果そのものの信頼性も疑わしくなる。MT(Mersenne Twister, http://www.math.keio.ac.jp/matu moto/mt.html) あたりを使うべきであろう。

「ランダム」というのは、単に我々の予測がつかないということではない。場合によっては、実験結果をさらに調べるために再実験する必要もある。

次に、配列内の要素を順に出力する作業はどの様に行えば良いのか？

基本的には、1番目の要素，2番目の要素，3番目の要素，... と、順に出力するだけである。 ただその際、1行の文字数が50~ 100文字になる様に1行に出力するデータの個数を決め、その個数のデータ出力で1行が埋まり次第改行(i.e.改行コードを1個出力)した方が良い。

そのためには、 現在の行でデータ出力した個数を保持する変数 count を用意し、

初期設定として	count←−0、
データ出力の度に	count←−count+1、
1行が埋まる度に	改行して count←−0、

とすれば良い。

配列内の要素を小さい順に並べ替える作業はどの様に行えば良いのか？

整列化(sorting)のアルゴリズムは色々なものがこれまでに考案されている。そのうち、ここではクイック整列法(quicksort, クイックソート)を紹介しよう。

具体的には、クイック整列法は

> 整列化の済んでいない部分列 v[f],v[f+1], ..., v[t] が与えられた時、それらの内容を並べ替えて
>
> v[f], ..., v[p-1] (全てv[p]以下), v[p], v[p+1], ..., v[t] (全てv[p]より大)
>
> (但し、pの値, v[p]の値は任意。)
>
> という風にする操作(分割操作 という。また、v[p]を 枢軸 要素という。)

を未整列の部分に繰り返し適用する方法であり、そのアルゴリズムは次のように書き表すことが出来る。

① 与えられたデータ a[0], a[1], ..., a[n-1] に分割操作を適用する。(その結果、枢軸要素が a[p] になったとする。)

② a[0], a[1], ..., a[p-1] を整列化する小問題に対して、このアルゴリズムをさらに適用する。

③ a[p+1], a[p+2], ..., a[n-1] を整列化する小問題に対して、このアルゴリズムをさらに適用する。

(プログラミング)　100個の整数データを保持するために a という名前のint型配列を用意し、

- 配列a内の各要素にランダムに整数を生成する作業,
- 配列a内の全要素を小さい順に並べ替える作業,
- 配列a内の全要素を順に出力する作業

を行うために各々

```
set_an_array_random(...),
quicksort(...),
pretty_print(...)
```

という名前の関数を用意する。

配列 a[] の要素はこれら3つの関数から参照できる必要があるが、現時点では配列を関数の引数として受け渡す方法について説明していないので、ここでは配列 a[] は大域的なものとして宣言することにする。

関数 set_an_array_random(...) の引数と値はどう設定すれば良いのか

配列 a[] を大域的なものにしたので、main()関数から引き渡すデータは何もないし、関数値として知らせてほしいものも何もない。従って、この関数のプロトタイプは次の様にすれば良い。

```
void set_an_array_random(void);
```

関数 pretty_print(...) の引数と値はどう設定すれば良いのか？

この関数についても、set_an_array_random()関数と同様に、main()関数から引き渡すデータは何もないし、関数値として知らせてほしいものも何もない。従って、この関数のプロトタイプは次の様にすれば良い。

```
void pretty_print(void);
```

関数 quicksort(...) の引数と値はどう設定すれば良いのか？

クイック整列法を適用する部分は最初は配列 a[] 全体であるけれども、分割操作を何回か繰り返すことによって未整列の部分は配列内の色々な場所に小区間(i.e. 添字の連続した配列要素の列)として残ることになる。

そこで、関数 quicksort(...) の機能を単に配列 a[] 内の要素全体を小さい順に並べ替えるというものに限定するのではなく、配列 a[] 内の任意の小区間内の要素を小さい順に並べ替えれるものに一般化しておくと、分割操作後に出来る2つの小区間にクイック整列法を適用する処理は単に quicksort(...) を再帰的に呼び出すだけで済む。

並べ替える小区間内の最初と最後の要素番号 from, to を関数 quicksort の引数として使うことにすれば、quicksort(from, to) の処理は...

① 小区間内のデータ a[from], a[from+1], ..., a[to] に分割操作を適用する。(その結果、枢軸要素が a[p] になったとする。)

② quicksort(from,p-1) を再帰的に呼び出す。

③ quicksort(p+1,to) を再帰的に呼び出す。

関数 quicksort( ) の処理結果としてmain()関数が受け取るものは、やはり何もない。従って、この関数のプロトタイプは次の様にすれば良い。

```
void quicksort(int from, int to);
```

以上の様に3つの関数 set_an_array_random(), quicksort(), pretty_print() を構成し、さらに quicksort() 関数の処理の見通しを良くするために小区間 a[from]～a[to] に対して分割操作を行い枢軸要素の添字番号を返す関数

```
int partition(int from, int to);
```

も構成することにする。

主関数main()からこれらの関数を呼び出すことによって

①ランダムに整数を生成して100個の配列要素 a[0]～a[99] を初期設定 、
②ランダムに生成された データの表示 、
③クイック整列法による配列内のデータの 並べ替え 、
④整列後の データの表示 、

を順に行うCプログラムと、.....

```
[motoki@x205a]$  nl function-quicksort.c
     1  /*************************************************
     2  /* Quicksort  : 再帰計算の例
     3  /*-----------------------------------
     4  /*     大きさ100の配列にランダムに整数を生成し、その... */
     5  /*     Quicksortアルゴリズムを使って昇順に並べ替えて出...
     6  /*************************************************

     7  #include <stdio.h>
     8  #include <stdlib.h>      /* 乱数発生のライブラリ関数... *

     9  #define  SIZE   100
    10  #define  WIDTH  10
    11  #define  TRUE    1

    12  void set_an_array_random(void);
    13  void pretty_print(void);
```

```
14  void quicksort(int from, int to);
15  int  partition(int from, int to);

16  int  a[SIZE];              /* 外部配列 */

17  int main(void)
18  {
19    int  seed;

20    printf("Input a random seed (0 - %d):  ", RAND_MAX);
21    scanf("%d", &seed);
22    srand(seed);

23    set_an_array_random();
24    printf("\nbefore sorting:\n");
25    pretty_print();
```

```
26    quicksort(0, SIZE-1);
27    printf("\nafter sorting:\n");
28    pretty_print();
29    return 0;
30  }

31  /*----------------------------------------------
32  /* 引数で与えられた配列の各要素をランダムに設定
33  /*----------------------------------------------
34  /* (仮引数) :  なし
35  /* (関数値) :  なし
36  /* (機能) :  配列要素 a[0]～a[SIZE-1] に 0～999 の間...
37  /*           設定する。
38  /*----------------------------------------------
39  void set_an_array_random(void)
40  {
41    int i;
```

```
42    for (i=0; i<SIZE; ++i)
43      a[i] = rand() % 1000;
44  }

45  /*-----------------------------------------------
46  /* 引数で与えられた配列の要素を順番に全て出力
47  /*-----------------------------------------------
48  /* (仮引数) :  なし
49  /* (関数値) :  なし
50  /* (機能) :  配列要素 a[0]～a[SIZE-1] の値を順番に...  *
51  /*            する。但し、各々の値は横幅7カラムのフ...  */
52  /*            に出力することにし、また、1行にWIDTH...  */
53  /*            を出力する。
54  /*-----------------------------------------------
55  void pretty_print(void)
56  {
```

```
57    int i, count=1;

58    for (i=0; i<SIZE; ++i, ++count) {
59      printf("%7d", a[i]);
60      if (count >= WIDTH) {
61        printf("\n");
62        count = 0;
63      }
64    }
65    if (count > 1)
66      printf("\n");
67  }

68  /*-------------------------------------------
69  /* 引数で与えられた配列要素を小さい順に並べ替える */
70  /*-------------------------------------------
71  /* (仮引数) from : int型配列 a の添字
```

```
72  /*            to    : int型配列 a の添字
73  /* (関数値)  :  なし
74  /* (機能) :  quicksortアルゴリズムを使って、配列要素
75  /*                  a[from],a[from+1],a[from+2], ..., a
76  /*              を値の小さい順に並べ替える。
77  /*-------------------------------------------------------
78  void quicksort(int from, int to)
79  {
80    int  pivot_sub;                  /* pivot subscript の意

81    if (from < to) {
82      pivot_sub = partition(from, to);        /* 分割操作
83      quicksort(from, pivot_sub - 1);
84      quicksort(pivot_sub + 1, to);
85    }
86  }
```

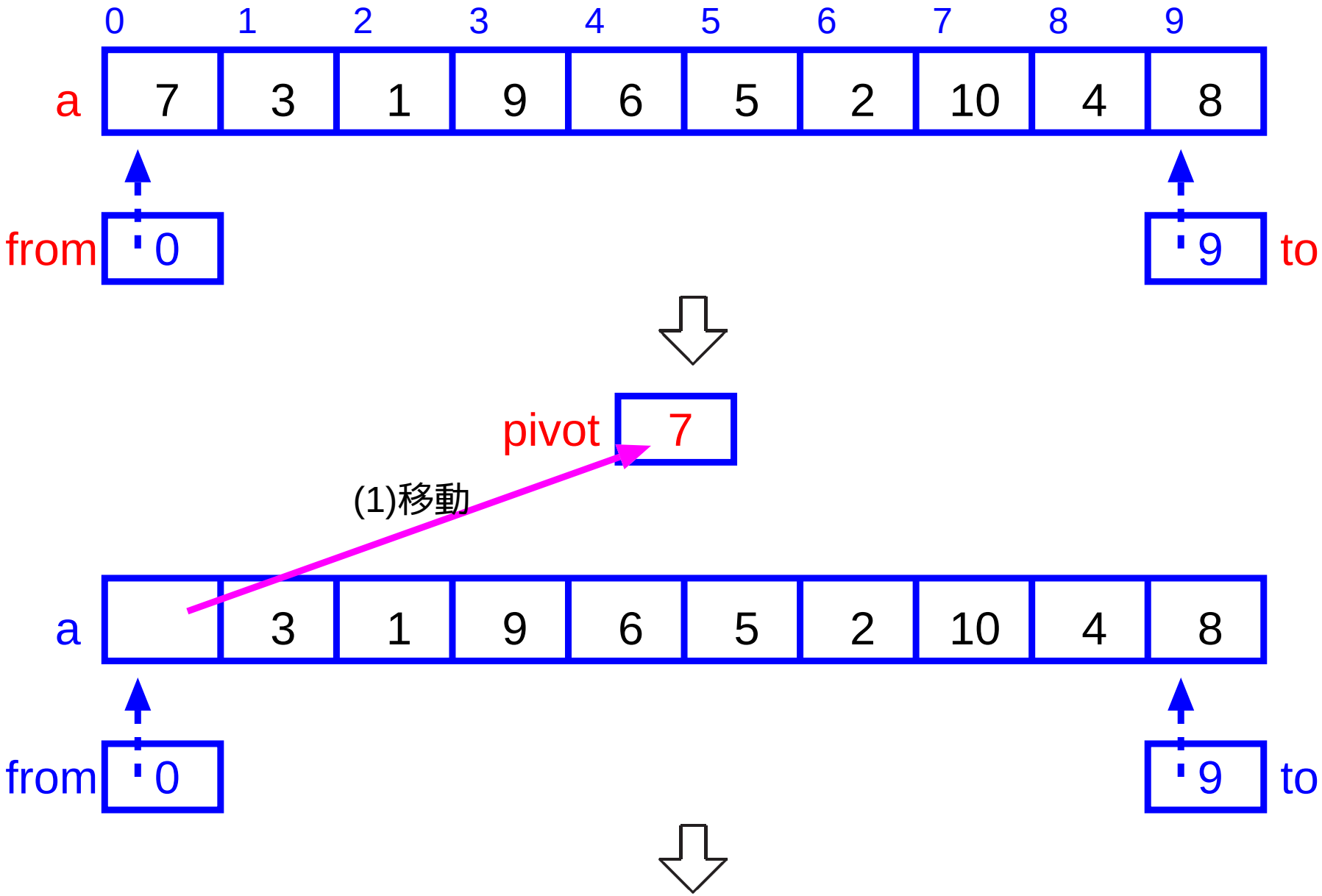
```
 87  /*-----------------------------------------
 88  /* 引数で与えられた配列の部分列にquicksortの分割操作... *
 89  /*                                  (quicksortの関数) */
 90  /*-----------------------------------------
 91  /* (仮引数) from : int型配列 a の添字
 92  /*           to   : int型配列 a の添字
 93  /* (関数値)  :  分割操作によって得られた枢軸要素の添字...
 94  /*               (以下の「(機能)」の項で出て来る pivot_sub
 95  /* (機能) :  a[from]～a[to]を並べ替えて
 96  /*           max{a[from],...,a[pivot_sub-1]} <= a[pivo
 97  /*           a[pivot_sub] < min{a[pivot_sub+1],...,a[t
 98  /*           となるようにする。
 99  /*-----------------------------------------
100  int  partition(int from, int to)
101  {
102    int pivot;
```

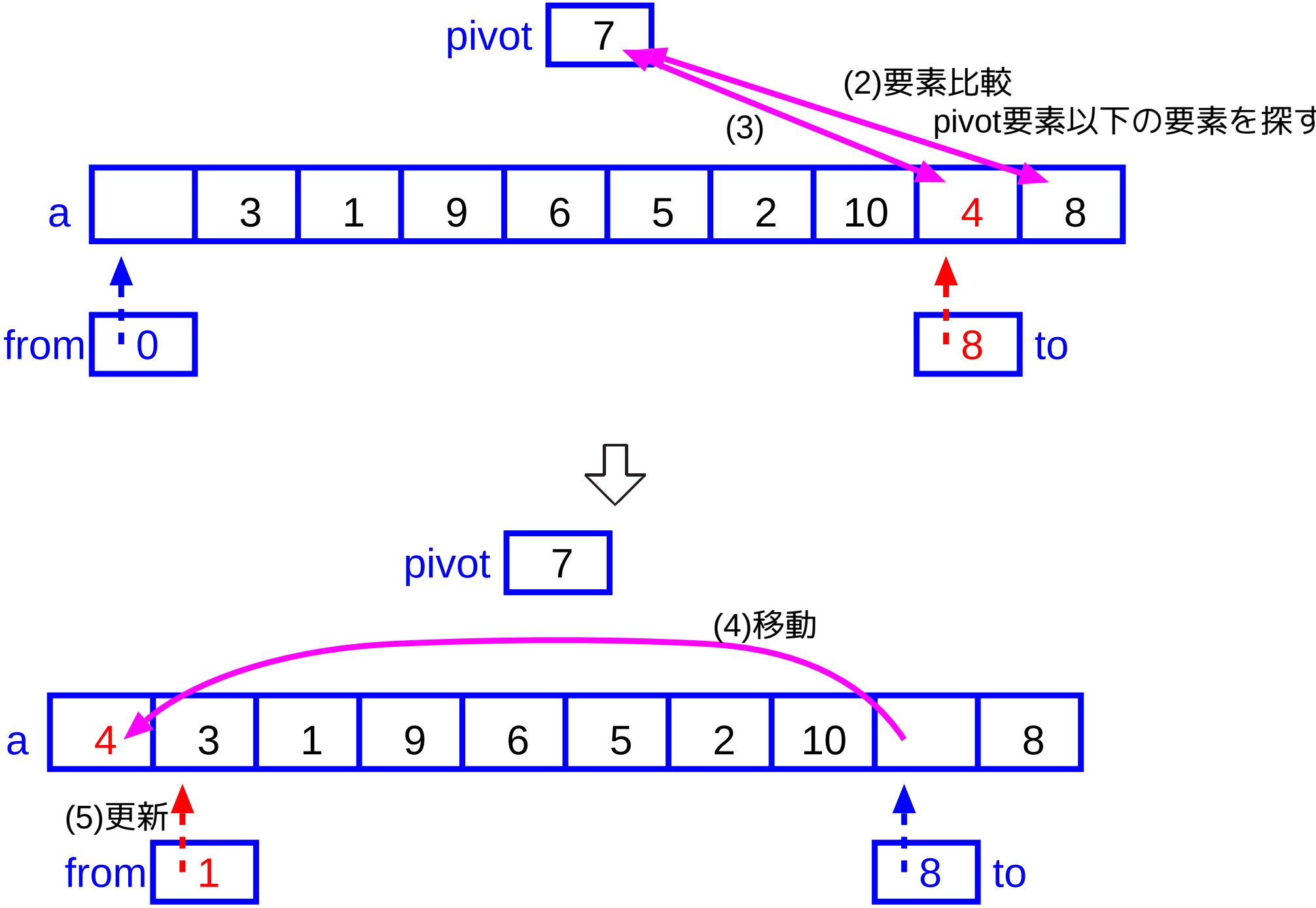
```
103     pivot = a[from];        /* 最初の要素を枢軸要素に選ぶ。 *
104     while (TRUE) {          /* 工夫の余地あり。              *
105       for ( ; from<to && a[to]>pivot; --to)
106         ;
107       if (from == to) {
108         a[from] = pivot;
109         return from;
110       }
111       a[from++] = a[to];

112       for ( ; from<to && a[from]<=pivot; ++from)
113         ;
114       if (from == to) {
115         a[to] = pivot;
116         return to;
117       }
118       a[to--] = a[from];
```

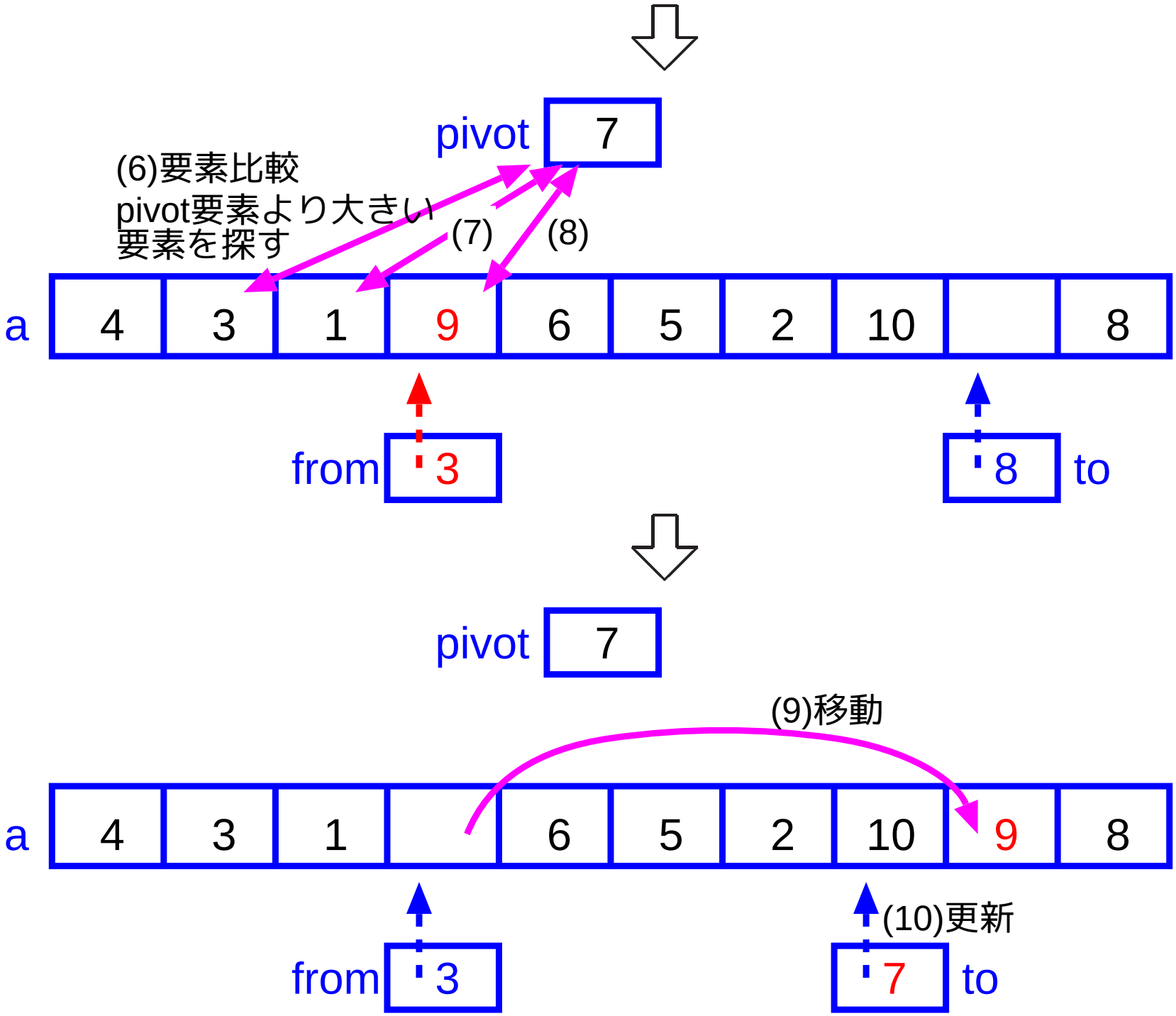
```
   119    }
   120  }
[motoki@x205a]$  gcc function-quicksort.c
[motoki@x205a]$  ./a.out
Input a random seed (0 - 32767):   333

before sorting:
    556    289    435    368    666    319    214    273     13
     64    943    869    956     50    298    112    218
    603    936    515    385    671    776    137    886
    718    913    204    153    281    870    473    495     14
    432    208    548    653    517    950    951    629     52
    630    476    893    498    861    917    626    998     80
    913    521    544    470     27    825    340    500     67
    105    104    397      6    110    914    308     61     89
     18    878    305    264    376    518    181    354     51
    985    782    857    881    252    236    706    945     73
```

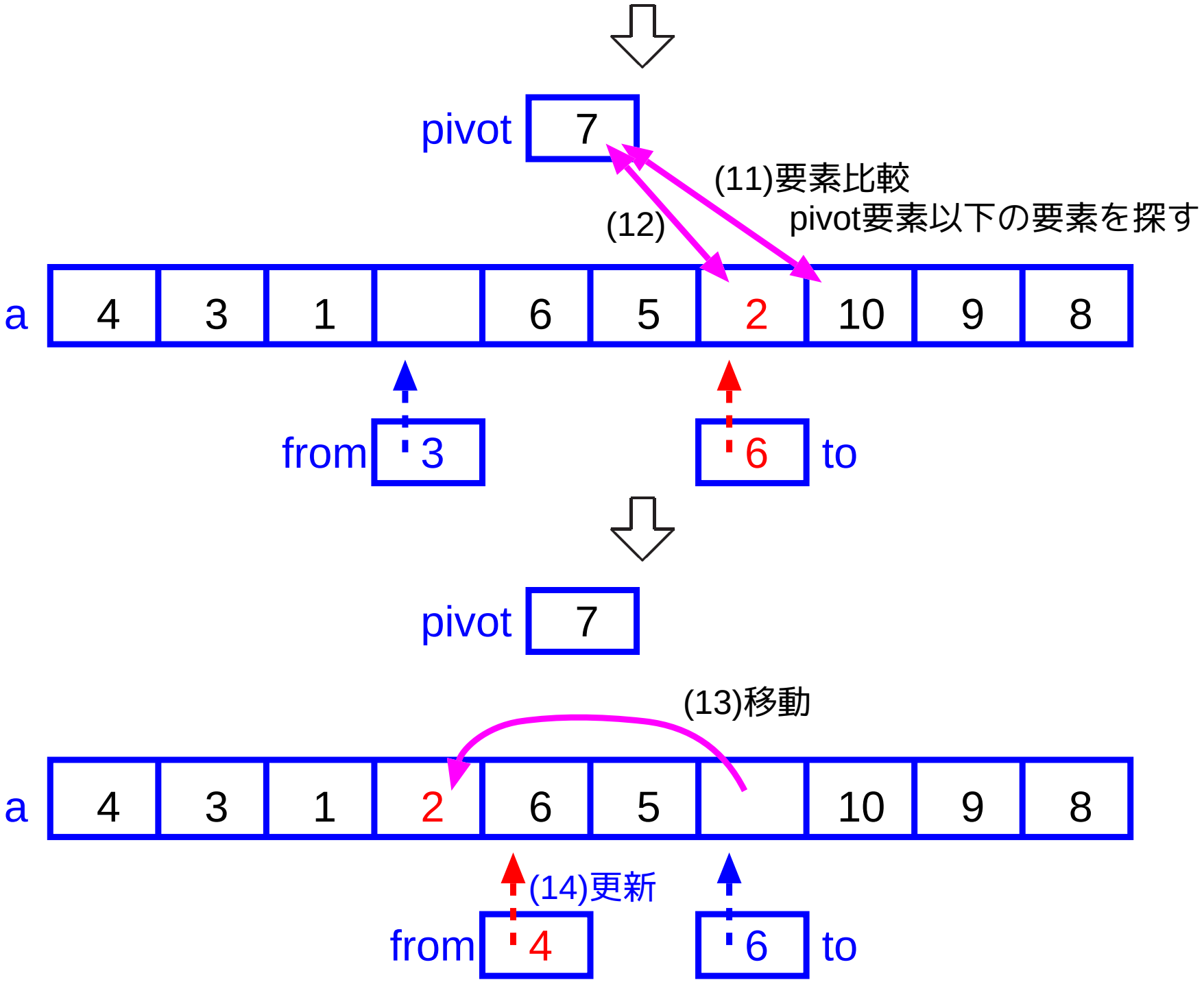
```
after sorting:
       4       5       6      18      27      50      61      64      10
     110     112     132     137     144     153     181     204     20
     218     236     252     264     273     281     289     298     30
     319     336     340     354     368     376     385     397     43
     470     473     476     495     498     500     515     517     51
     520     521     544     548     556     563     585     603     60
     629     630     631     649     653     666     671     672     70
     730     736     776     782     803     825     829     836     85
     869     870     878     881     886     893     895     913     91
     917     936     943     945     950     951     956     957     98
[motoki@x205a]$
```

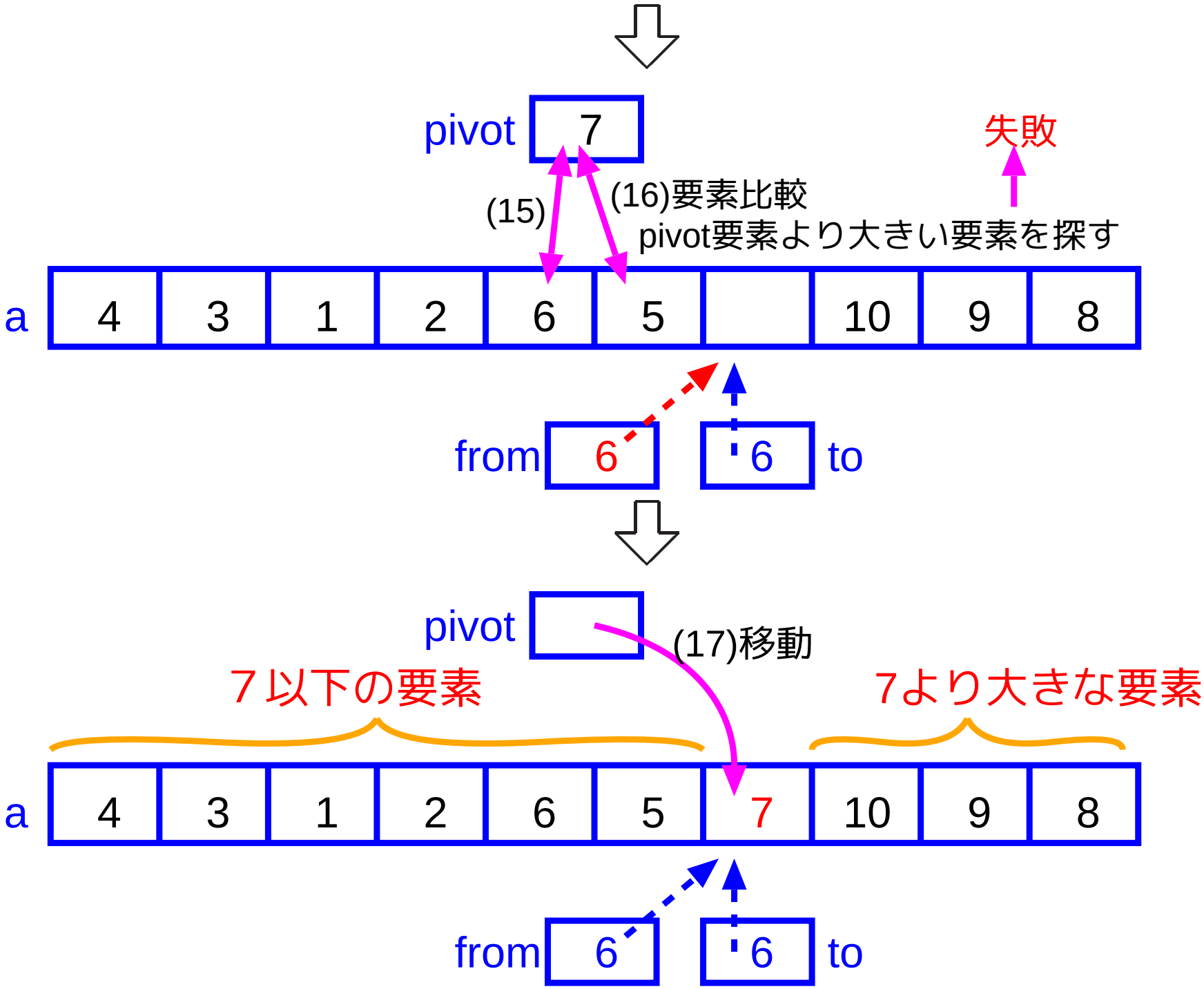
関数 partition(from, to) におけるデータの移動：




pivot
7
(2)要素比較
(3)
pivot要素以下の要素を探す
a
3
1
9
6
5
2
10
4
8
from
0
8
to
pivot
7
(4)移動
a
4
3
1
9
6
5
2
10
8
(5)更新
from
1
8
to


pivot
7
(6)要素比較
pivot要素より大きい
要素を探す
(7)
(8)
a
4
3
1
9
6
5
2
10
8
from
3
8
to
pivot
7
(9)移動
a
4
3
1
6
5
2
10
9
8
from
3
(10)更新
7
to


pivot
7
(11)要素比較
pivot要素以下の要素を探す
(12)
a
4
3
1
6
5
2
10
9
8
from
3
6
to
pivot
7
(13)移動
a
4
3
1
2
6
5
10
9
8
(14)更新
from
4
6
to


pivot
7
失敗
(15)
(16)要素比較
pivot要素より大きい要素を探す
a
4
3
1
2
6
5
10
9
8
from
6
6
to
pivot
(17)移動
7以下の要素
7より大きな要素
a
4
3
1
2
6
5
7
10
9
8
from
6
6
to

関数 quicksort(from, to) の通常のコード化：

```
[motoki@x205a]$ nl function-quicksort-std-code.c
          .........
    78 void quicksort(int from, int to)
    79 {
    80   int pivot, i, j, tmp;

    81   pivot = a[(from+to)/2];
    82   i = from;
    83   j = to;

    84   while (i <= j) {          /* 分割操作 */
    85     while (a[i] < pivot) i++;
    86     while (pivot < a[j]) j--;
    87     if (i <= j) {
    88       tmp  = a[i];  /* swap */
```

```
89          a[i] = a[j];
90          a[j] = tmp;
91          i++;
92          j--;
93        }
94    }

95    if (from < j)                 /* 再帰処理 */
96      quicksort(from, j);
97    if (i < to)
98      quicksort(i, to);
99  }
```

4–7 付録 標準ライブラリ関数についての案内

- 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。

関数プロトタイプ　…　説明
`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。
`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, 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。