15 ファイル入出力とOSとのインターフェース

15–1 復習 ファイル入出力 ---fopen() と fcl

cプログラムで操作するファイルは通常はテキストファイルで、
前から順に処理される。

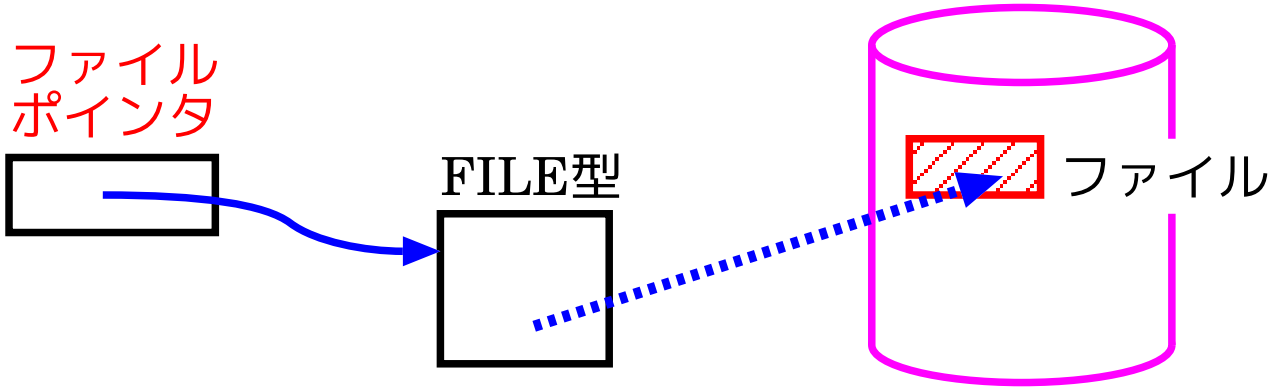
例えば、
ファイルからデータを入力する場合は、...
また、ファイルへデータ出力する場合は、...

⇨ ファイル操作の間ファイルのどの場所を現在処理しているかの情報を常にどこかに保持しておく必要があり、この情報も含めてファイル操作を快調に行うための情報を維持しておく必要がある。

しかし、ディスク上のファイルに直接アクセスするという操作は、システム管理の問題と関わって来るので、一般ユーザには許されていない。

そこで、C言語においては 、

- プログラムからの要求に応じて、
  言語処理系/OSがこれらのファイル操作に必要/有用な情報をFILE というデータ型名の付いた構造体の中に詰め込み、
- プログラムの中でこのFILE型構造体へのポインタ(i.e.所在番地)を保持する、

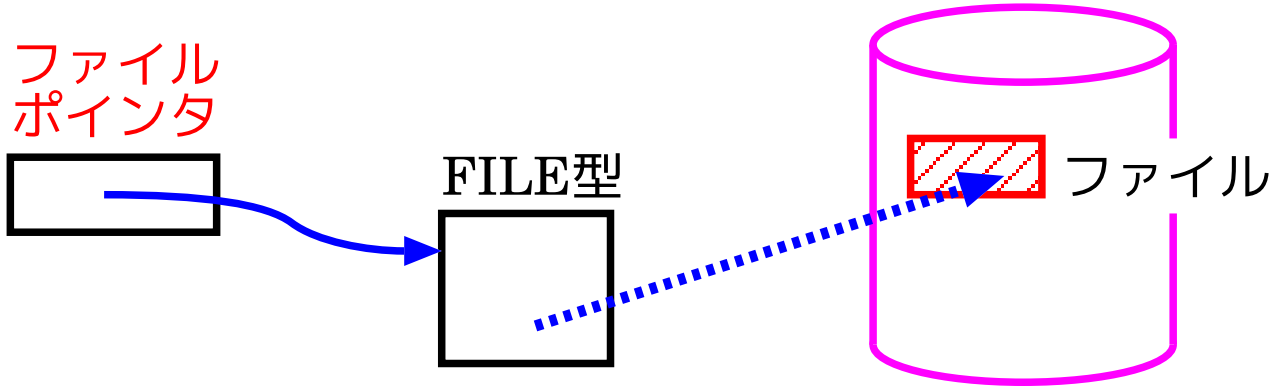


そして

- プログラムの中では、FILE型構造体へのポインタを明示することによって、間接的に操作目的のファイルを指定する、

という仕組みが取られている。

このFILE型構造体へのポインタのことをファイルポインタと言う。



また、
操作を始めたいファイルに関するFILE型構造体を言語処理系/OSに作ってもらってファイル操作の準備を行うことを、ファイルをオープンすると言い、逆に操作の終ったファイルのFILE型構造体の領域を解放してファイル操作の後始末を行うことをファイルをクローズすると言う。

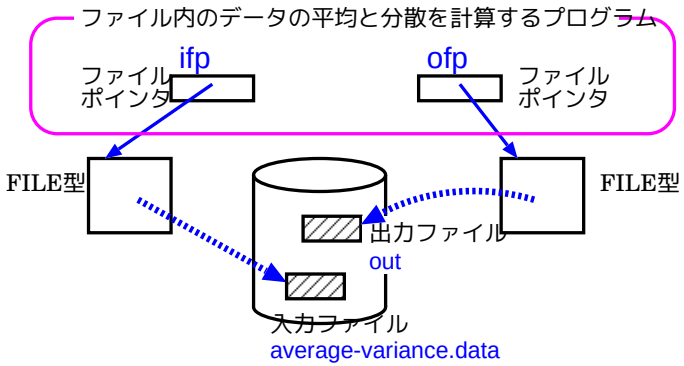
例題15．1（ファイル内のデータの平均と分散）　例題 1.12 と同じ様に、50個の実数データ $\mathbf{x_0}$，$\mathbf{x_1}$，$\mathbf{x_2}$，$\cdots$，$\mathbf{x_{49}}$ を読み込み、それらの平均 μ と分散 $\mathbf{V}$ を定義式

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

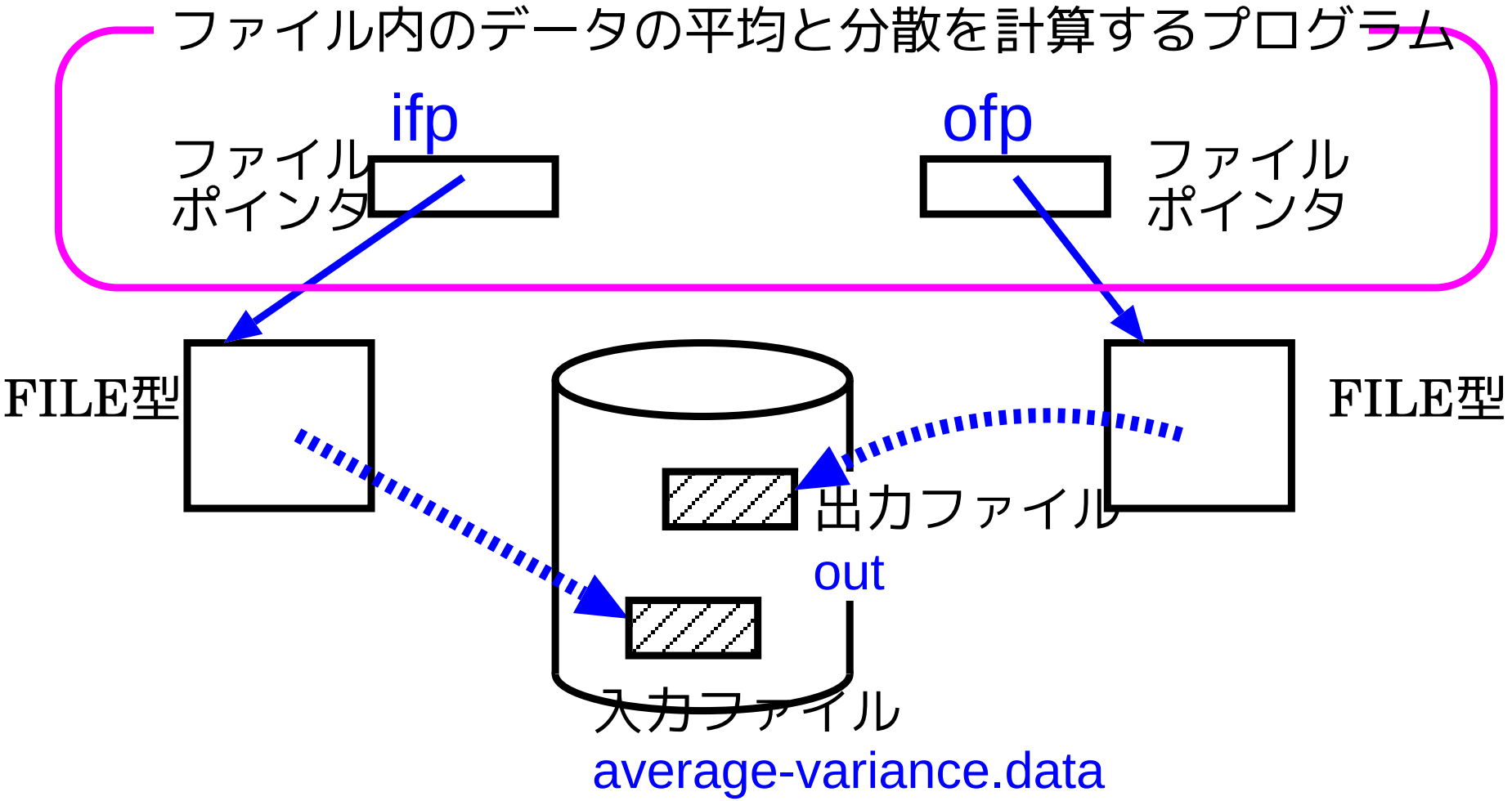
$$\mathbf{V} = \sum_{\mathbf{i=0}}^{\mathbf{49}} (\mathbf{x_i} - \mu)^{\mathbf{2}}/\mathbf{50}$$

に従って求めて出力するCプログラムを作成せよ。但し、ここでは入力データは **average-variance.data** という名前のファイルから読み込み、出力データは **out** というファイルに書き出すことにする。

(考え方)　入力ファイルに繋がるファイルポインタ，出力ファイルに繋がるファイルポインタの領域が必要である。



拡大→次ページ


ファイル内のデータの平均と分散を計算するプログラム
ifp
ファイル
ポインタ
ofp
ファイル
ポインタ
FILE型
FILE型
出力ファイル
out
入力ファイル
average-variance.data

例題15．1（ファイル内のデータの平均と分散）　例題1.12と同じ様に、50個の実数データ $\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{49}}$ を読み込み、それらの平均 μ と分散 $\mathbf{V}$ を定義式

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

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

に従って求めて出力するCプログラムを作成せよ。但し、ここでは入力データは **average-variance.data** という名前のファイルから読み込み、出力データは **out** というファイルに書き出すことにする。

（考え方）　入力ファイルに繋がるファイルポインタ，出力ファイルに繋がるファイルポインタの領域が必要である。

これらのものが用意されていれば、基本的なデータ処理の流れは例題1.12と同じで良いので、次の4点に注意して例題1.12のプログラムに少し手を加えるだけである。

（考え方）　入力ファイルに繋がるファイルポインタ，出力ファイルに繋がるファイルポインタの領域が必要である。

これらのものが用意されていれば、基本的なデータ処理の流れは例題1.12と同じで良いので、次の4点に注意して例題1.12のプログラムに少し手を加えるだけである。（⟹4.7節を参照）

- データ処理の前にライブラリ関数 `fopen()` を用いて2つのファイルをオープンする必要がある。
- データ処理の後にライブラリ関数 `fclose()` を用いて2つのファイルをクローズする必要がある。
- 指定されたファイルからデータを入力するので、`scanf()` ではなく `fscanf()` を用いる。
- 指定されたファイルへデータを出力するので、`printf()` ではなく `fprintf()` を用いる。

(プログラミング)

例題1.4で考えた配列や変数の他に、
ファイルポインタを保持のためにifp, ofp という名前の変数を
用意して、プログラムを構成した。

```
[motoki@x205a]$  nl average-variance-io-through-files.c
     1 /* 50個の実数データ x0, x1, x2, ... , x49 の平均 mu と
     2 /* 分散 $V$ を定義式
     3 /*    mu = (x0+x1+x2+ ...  + x49)/50
     4 /*     V = (x0-mu)^2 + (x1-mu)^2 + ... + (x49-mu)^2 / 50
     5 /* に従って求め、それらの値を出力するプログラム
     6 /* 但し、ここでは入力データは average-variance.data とい.
     7 /* 名前のファイルから読み込み、出力データは out というフ..*/
     8 /* イルに書き出すことにする。

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

```
11 int main(void)
12 {
13   int      i;
14   double   x[50], ave, var;
15   FILE     *ifp, *ofp;

16   if ((ifp=fopen("average-variance.data", "r"))
                                          == NULL) {
17     printf("ファイルをオープン出来ません: average-varianc
18     exit(1);
19   }
20   if ((ofp=fopen("out", "w")) == NULL) {
21     printf("ファイルをオープン出来ません: out\n");
22     exit(1);
23   }
24
```

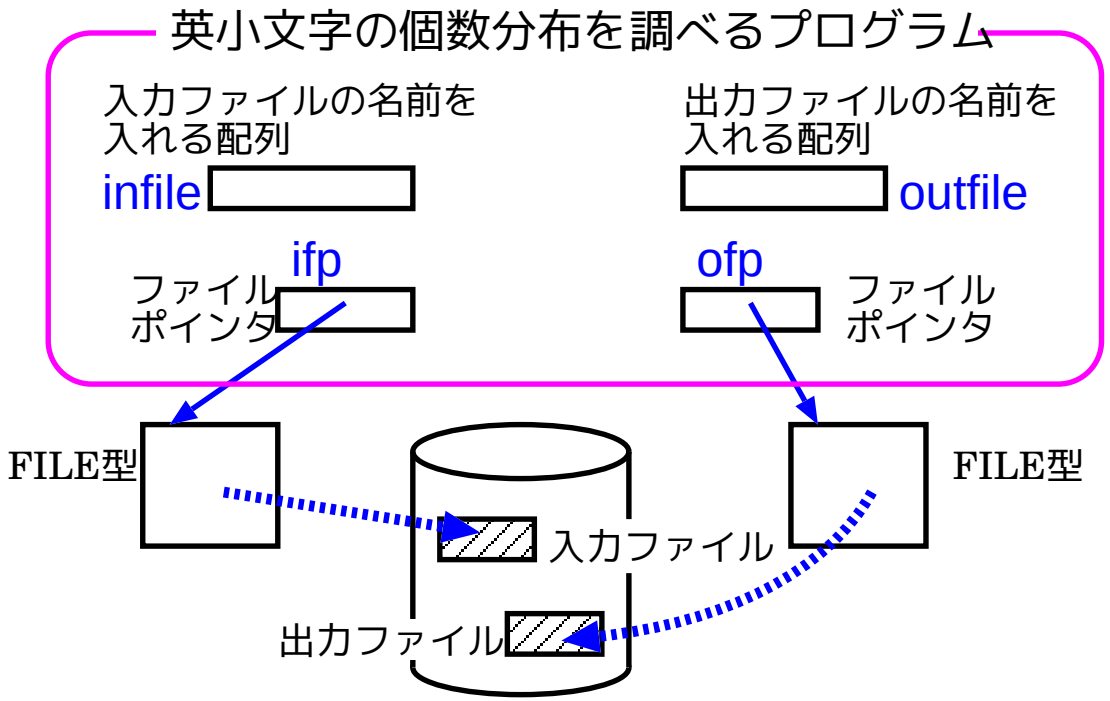
```
25    ave = 0.0;
26    for (i=0; i<50; ++i) {
27      fscanf(ifp, "%lf", &x[i]);
28      ave += x[i];
29    }
30    ave /= 50.0;

31    var = 0.0;
32    for (i=0; i<50; ++i)
33      var += (x[i]-ave)*(x[i]-ave);
34    var /= 50.0;

35    fprintf(ofp, "\nInput data:\n");
36    for (i=0; i<50; i+=5)
37      fprintf(ofp, "%14.5e%14.5e%14.5e%14.5e%14.5e\n",
38              x[i], x[i+1], x[i+2], x[i+3], x[i+4]);
39    fprintf(ofp, "\nAverage  = %14.6g\n"
```

```
    40              "Variance = %14.6g\n", ave, var);

    41    fclose(ifp);
    42    fclose(ofp);
    43    return 0;
    44 }
[motoki@x205a]$  gcc average-variance-io-through-files.c
[motoki@x205a]$  ./a.out
[motoki@x205a]$  cat out

Input data:
   1.00000e+00   1.00010e+00   1.00020e+00   1.00030e+00   1.0
   1.00050e+00   1.00060e+00   1.00070e+00   1.00080e+00   1.0
   1.00100e+00   1.00110e+00   1.00120e+00   1.00130e+00   1.0
   1.00150e+00   1.00160e+00   1.00170e+00   1.00180e+00   1.0
   1.00200e+00   1.00210e+00   1.00220e+00   1.00230e+00   1.0
   1.00250e+00   1.00260e+00   1.00270e+00   1.00280e+00   1.0
```

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

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

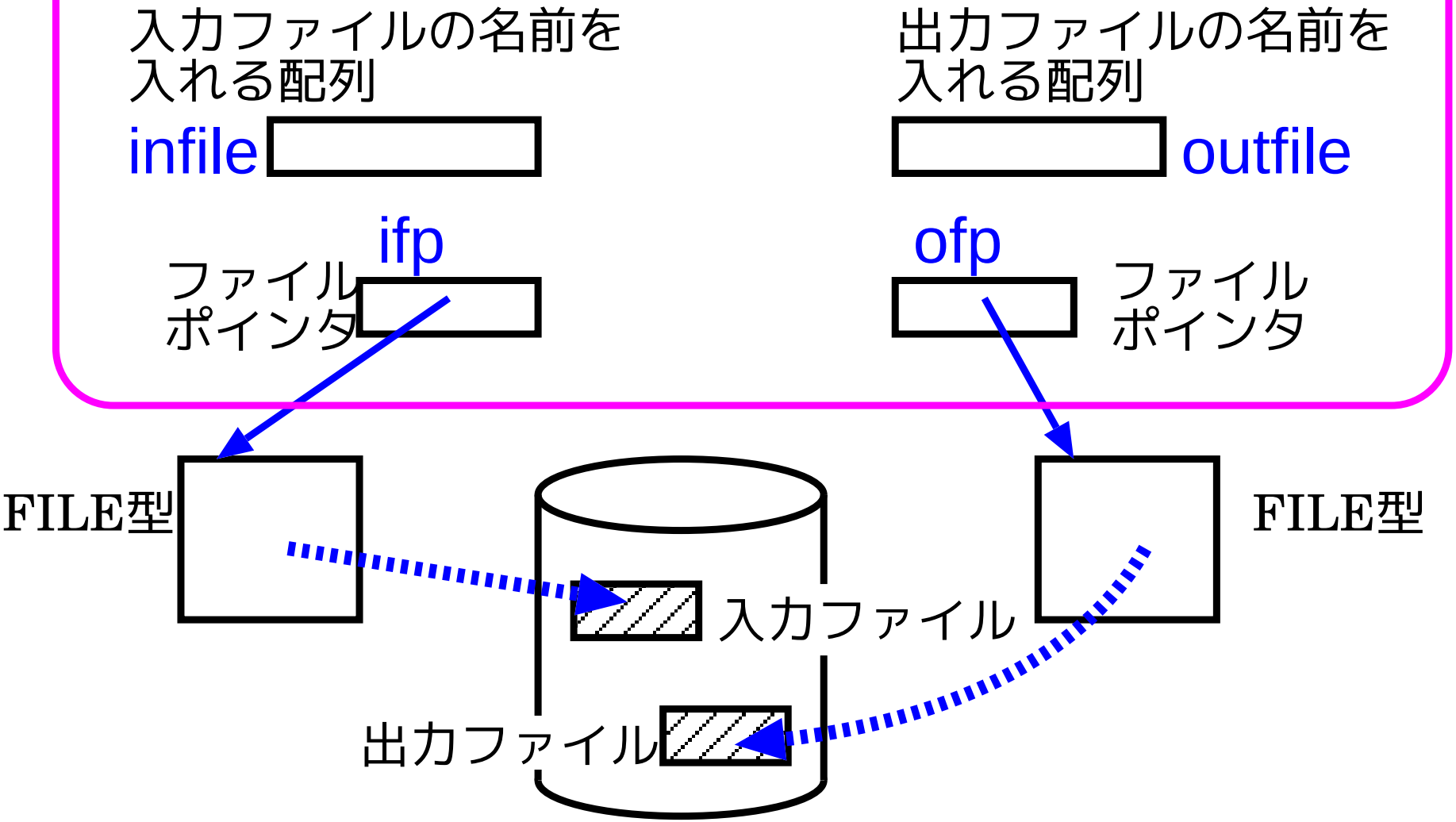
例題15．2（指定したファイル中の英小文字の個数分布）
①入力ファイル、出力ファイルの名前を標準入力から受け取り、
②入力ファイル中の英小文字の個数分布を出力ファイルに書き出す
Cプログラムを作成せよ。

（考え方）　入力ファイルと出力ファイルの名前を文字列として保持する十分長いchar型配列、それから入力ファイルに繋がるファイルポインタ，出力ファイルに繋がるファイルポインタの領域が必要である。



拡大→次ページ

英小文字の個数分布を調べるプログラム



例題15．2（指定したファイル中の英小文字の個数分布）
①入力ファイル、出力ファイルの名前を標準入力から受け取り、
②入力ファイル中の英小文字の個数分布を出力ファイルに書き出す
Cプログラムを作成せよ。

（考え方）　入力ファイルと出力ファイルの名前を文字列として保持する十分長いchar型配列、それから入力ファイルに繋がるファイルポインタ，出力ファイルに繋がるファイルポインタの領域が必要である。

これらのものが用意されていれば、

- ライブラリ関数 fopen() を利用して、char型配列で指定したファイルをオープンできる。 （⟹ 4.7節を参照）
- ライブラリ関数 fclose() を利用して、ファイルポインタで指定したファイルをクローズできる。 （⟹ 4.7節を参照）
- ライブラリ関数 fgetc() を用いて、ファイルポインタで指定した先から次のデータを1文字分だけ読み込むことができる。
- ライブラリ関数 fprintf() や fputc() を用いて、ファイルポインタで指定した先に出力データを流し込むことができる。

度数分布を調べるためには 、単に、英小文字26文字それぞれの出現回数を数えるカウンタ(初期値0)を用意して、入力ファイルから英小文字を検出する度にその文字のカウンタを1だけ増やす操作を続ければ良い。その際、

文字 'a' の出現回数を count_of_letter[0] で,
文字 'b' の出現回数を count_of_letter[1] で,
.....,
文字 'z' の出現回数を count_of_letter[25] で

数える場合は、

'a'-'a'=97−97=0,
'b'-'a'=98−97=1,
.....
'z'-'a'=112−97=25

⟹ 5.1節を参照

と計算できるので、一般には

変数 c に記憶されている文字(番号)の出現回数は
count_of_letter[c-'a'] で数える

ことになる。

（プログラミング）

ファイル名保持のためにinfile[]，outfile[] というchar型配列を、
ファイルポインタ保持のためにifp，ofp という変数を、そして
英小文字26文字の各々の出現回数を数えるために count_of_letter[]
というint型配列を
用意して、プログラムを構成した。

```
[motoki@x205a]$  nl counting-lowercase-letters.c
     1 /* (1)入力ファイル，出力ファイルの名前を標準入力から受け...
     2 /* (2)入力ファイル中の英小文字の個数分布を出力ファイルに...
     3 /* Cプログラム

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

```
 6 int main(void)
 7 {
 8   int   c, i, count_of_letter[26];
 9   char  infile[100], outfile[100];
10   FILE  *ifp, *ofp;

11   printf("Type in a name of an input file:  ");
12   scanf("%s", infile);
13   printf("Type in a name of an output file: ");
14   scanf("%s", outfile);
15   if ((ifp=fopen(infile, "r")) == NULL) {
16     printf("ファイルをオープン出来ません: %s\n", infile);
17     exit(1);
18   }
19   if ((ofp=fopen(outfile, "w")) == NULL) {
20     printf("ファイルをオープン出来ません: %s\n", outfile)
21     exit(1);
22   }
```

```
23    for (i=0; i<26 ; ++i)        /* カウンタを全て0に初期化
24      count_of_letter[i] = 0;

25    while ((c=fgetc(ifp)) != EOF)
26      if (c>='a' && c<='z')
27        ++count_of_letter[c-'a'];

28    for (i=0; i<26; ++i) {
29      fprintf(ofp, "%c:%4d      ",
                'a'+i, count_of_letter[i]);
30      if (i%5==4)
31        fputc('\n', ofp);
32    }
33    fputc('\n', ofp);
34    fclose(ifp);
35    fclose(ofp);
36    return 0;
```

```
    37 }
[motoki@x205a]$  gcc counting-lowercase-letters.c
[motoki@x205a]$  ./a.out
Type in a name of an input file:
                                  counting-lowercase-letters.c
Type in a name of an output file:  out
[motoki@x205a]$  cat out
a:  13      b:   1      c:  20      d:   5      e:  32
f:  47      g:   1      h:   4      i:  50      j:   0
k:   0      l:  20      m:   3      n:  32      o:  28
p:  23      q:   0      r:  13      s:  10      t:  32
u:  15      v:   0      w:   2      x:   2      y:   2
z:   1
[motoki@x205a]$
```

ファイルについてのまとめ：

- Cプログラムの中では(標準入出力も含めた)ファイルへのアクセスは、通常、ファイルポインタを介して行う。

- 内部的には、ファイルポインタは

  入力用か出力用か、
  次の読み込み文字の位置（または次の書き込み場所)、
  ファイル終端が起きたかどうか、

  などの情報から成る(FILE型)構造体へのポインタであり、特に標準入力，標準出力，標準エラー出力にアクセスするためのファイルポインタとしては、<stdio.h>の中でそれぞれ stdin, stdout, stderr という名前のものが用意されている。

補足：

FILE型の定義はシステムによって異なっている様である。

例えば、平林雅英「ANSI C/C++辞典」(共立出版,1996)にはFILE型の定義として次の様なものが例示されている。

```
typedef struct {
  unsigned char *fpi;    /* ファイル位置指示子 */
  unsigned char *bptr;   /* バッファへのポインタ */
  unsigned int  flags;   /* ファイル状態フラグ */
    ..........................
```

- ファイル処理は一般に次の様に行う。

① `#include <stdio.h>`

② `FILE *` ファイルポインタ型変数 ;

③ ファイルポインタ型変数
`=` `fopen(` ファイル名`,` 使用モード `);`
但し、使用モード としては次の3つが可能。
- `"w"` ... 新規書き込み
- `"a"` ... 追加書き込み
- `"r"` ... 読み込み

④ (ファイル操作)

⑤ `fclose(` ファイルポインタ型変数 `);`

15–2 自習 ファイル記述子による入出力

標準ライブラリ関数以外に、直接OSに処理を依頼するための関数（システムコールという）も用意されている。

例えば、ファイル入出力のシステムコール，
ファイル情報獲得のシステムコール，
プロセス生成のシステムコール，
プロセス間で通信するためのシステムコール，
.....

特にファイル入出力関連では 、オープン，クローズ，読み込み，書き込み という基本操作のためのシステムコールが用意されている。但し、

- システムコールの場合は、処理するファイルの指定はファイルポインタではなく、プロセスが開いたファイルに付けられるファイル記述子と呼ばれる識別番号によって行う。

- システムコールの場合は、入出力用のバッファもプログラマが確保する必要がある。

ファイル記述子：

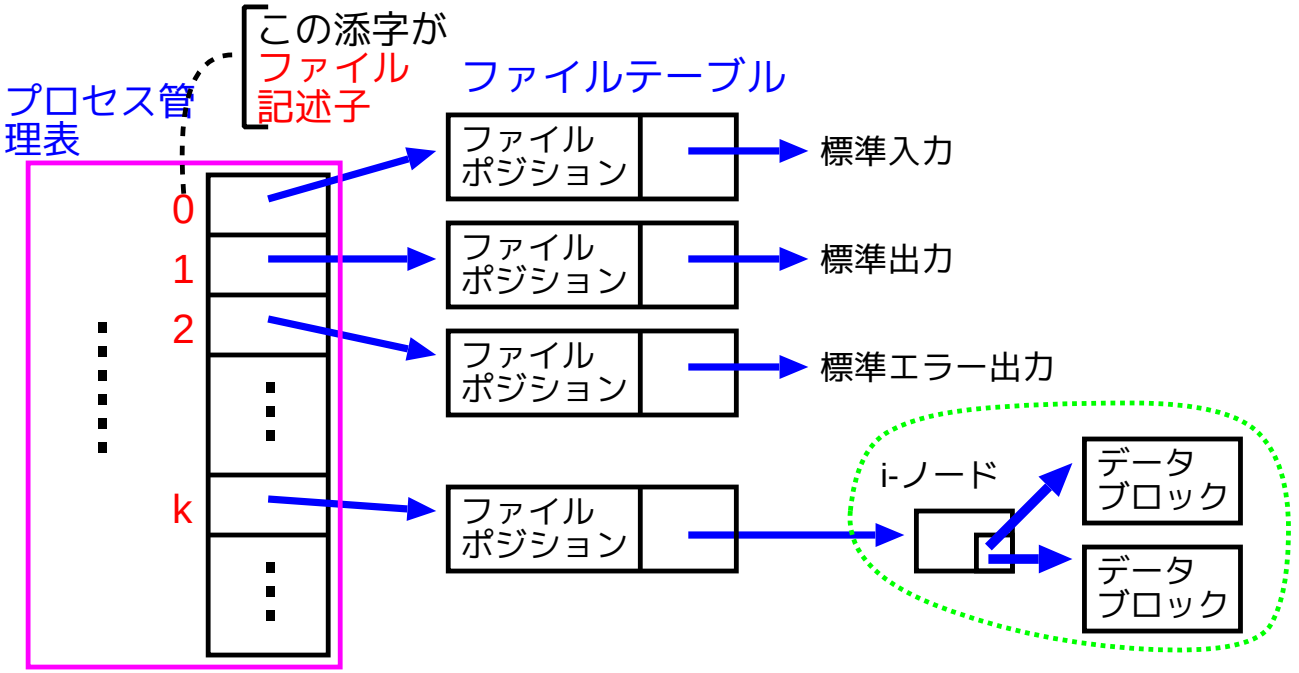
カーネルは、プロセスが開いたファイルの各々に対して

①次に読み書きする位置を示すファイルポジションと

②ファイルの実体を束ねるi-ノードへのポインタ

から成る、ファイルテーブルと呼ばれる構造体を構成する。

そして、プロセスが関与するファイルテーブルへのポインタの配列をプロセス毎に用意して管理する。これらの配列の添字はファイル記述子と呼ばれ、ファイル入出力に利用することが出来る。



ファイル記述子を用いた入出力の概要：

- ファイル入出力のシステムコールを使うためには、

```
#include <fcntl.h>
#include <unistd.h>
```

というヘッダファイル宣言が必要である。

また、入出力用のバッファをプログラマが確保する必要もある。

- ファイルをオープンしてそのファイル記述子を受け取るには、次の様に書く。

  int型変数 = open(ファイル指定, 使用モードの指定);

  または

  int型変数 = open(ファイル指定, 使用モードの指定, パーミッション指定);

補足：

◇ 関数値はオープンしたファイルのファイル記述子である。但し、オープンに失敗すると -1 である。

◇ 引数 ファイル指定 は、...

◇ 引数 使用モードの指定 はファイルの使い方を表したもので、次のマクロを OR 演算子 (|) で繋げて指定する。

O_RDONLY ··· 読み出し専用にオープンする。

O_WRONLY ··· 書き込み専用にオープンする。

O_EXCL ··· 作成しようとしているファイルが既に存在...

.....

◇ 引数 パーミッション指定 は、...

- ファイル記述子を指定してファイルをクローズするには、...

```
close( ファイル記述子 );
```

- ファイル記述子を指定してファイルからデータをバッファに読み込むためには、次の様に書く。

```
read( ファイル記述子, バッファ名, バッファの大きさ );
```

補足：

◇ 現在操作中の位置（この位置を指すポインタもファイルポインタと呼ぶ）からデータを読み込む。読み込みに成功すれば読み込んだデータのバイト数が関数値として返され、ファイルポインタは読み込んだバイト数だけ移動する。 また、失敗すれば -1 が返される。

◇ 引数 ファイル記述子 は...

◇ 引数 バッファ名 は読み込んだデータを保存するバッファのアドレスを表す。

◇ 引数 バッファの大きさ は読み込むデータのバイト数を表す。

- ファイル記述子を指定してバッファ内のデータをファイルに書き込むためには、次の様に書く。

```
write( ファイル記述子 , バッファ名 , データの大きさ );
```

補足：

◇ 成功すると書き出されたバイト数が返され、ファイルポインタは書き出されたバイト数だけ移動する。失敗すると -1 が返される。

◇ 引数 ファイル記述子 は...

◇ 引数 バッファ名 は書き出すデータを保存するバッファのアドレスを表す。

◇ 引数 データの大きさ は書き出すデータのバイト数を表す。

注意：

◇ システムバッファが用意される場合は、`read()`, `write()` はシステムバッファ上の仮想ファイルに対してだけ行われ磁気ディスクへの書き込みはしばらく後になる。

$\Longrightarrow$ 即座に磁気ディスクに書き込みたい場合は `sync()` システムコールを使う。

例題15．3（大文字↔小文字の反転）　コマンドラインの
１番目の引数で指定したファイル の大文字と小文字を反転して、
その結果を ２番目の引数で指定したファイル に書き出す
プログラムをファイル記述子を使って構成してみよ。

（考え方）

引数で指定された2つのファイルをopen()システムコールを使ってオープンし、

① 入力用ファイルの先頭からある大きさのブロック単位で順に文字列をread()システムコールで読み出し、

② 読み込んだ各々の文字の大文字・小文字をislower(), toupper(), tolower()ライブラリ関数を使って反転し、

③ その結果をwrite()システムコールを使って出力用ファイルに書き出す、

ということを繰り返せば良い。

入力用のファイルが無かったり、出力用のファイルが既に存在している場合はエラーとして強制終了させれば良い。

その際、ライブラリ関数 perror() を用いれば、open()の際にどういうエラーがコンピュータ内部で起こったのかを表示できる。

（プログラミング）　入力ファイルから一度に読み込むブロックの最大サイズを1024（BUFSIZE）として、プログラムを構成した。

[motoki@x205a]$ nl file-descriptor-flip-lower-upper.c

```
 1 /* 大文字-小文字の反転 */
 2 #include <stdio.h>
 3 #include <ctype.h>
 4 #include <fcntl.h>
 5 #include <unistd.h>
 6 #include <stdlib.h>

 7 #define BUFSIZE  1024

 8 int main(int argc, char **argv)
 9 {
10   char buffer[BUFSIZE], message[100];
11   int  in_fd, out_fd, in_size, k;
```

```
12    if ((in_fd=open(argv[1], O_RDONLY))==-1) {
13      sprintf(message, "open(%s, O_RDONLY)", argv[1]);
14      perror(message);
15      printf("  ===> Execution was aborted.\n");
16      exit(EXIT_FAILURE);
17    }
18    if ((out_fd=open(argv[2], O_WRONLY|O_EXCL|O_CREAT,
                       0644))==-1) {
19      sprintf(message,
                "open(%s, O_WRONLY|O_EXCL|O_CREAT, 0644)",
                argv[2]);
20      perror(message);
21      printf("  ===> Execution was aborted.\n");
22      exit(EXIT_FAILURE);
23    }
24    while ((in_size=read(in_fd, buffer, BUFSIZE)) > 0) {
```

```
25      for (k=0; k<in_size; k++) {
26        if (islower(buffer[k]))
27          buffer[k]=toupper(buffer[k]);
28        else
29          buffer[k]=tolower(buffer[k]);
30      }
31      write(out_fd, buffer, in_size);
32    }
33    close(in_fd);
34    close(out_fd);
35    return 0;
36  }
```

```
[motoki@x205a]$  gcc file-descriptor-flip-lower-upper.c
[motoki@x205a]$  ./a.out abc out
open(abc, O_RDONLY): No such file or directory
  ===> Execution was aborted.
```

```
[motoki@x205a]$  ./a.out file-descriptor-flip-lower-upper.c ou

open(out, O_WRONLY|O_EXCL|O_CREAT, 0644): File exists
  ===> Execution was aborted.
[motoki@x205a]$  rm out
rm: ‘out’ を削除しますか(yes/no)?  y
[motoki@x205a]$  ./a.out file-descriptor-flip-lower-upper.c ou

[motoki@x205a]$  ls -l {file-descriptor-flip-lower-upper.c,out

-rw-rw-r--  1 motoki  motoki  966 Mar 21 16:01 file-descriptor
-rw-r--r--  1 motoki  motoki  966 Mar 21 16:05 out
[motoki@x205a]$  cat out
/* 大文字-小文字の反転 */
#INCLUDE <STDIO.H>
#INCLUDE <CTYPE.H>
#INCLUDE <FCNTL.H>
```

```
#INCLUDE <UNISTD.H>
#INCLUDE <STDLIB.H>

#DEFINE bufsize  1024

MAIN(INT ARGC, CHAR **ARGV)
{
  CHAR BUFFER[bufsize], MESSAGE[100];
  INT  IN_FD, OUT_FD, IN_SIZE, K;

      ......(以下省略)......

[motoki@x205a]$
```

15–3 Cプログラムの中からのコマンド実行

- Cプログラムの中からOSのコマンドを実行するには、次の様に書く。(<s

```
system(" コマンド ");
```

例15．4（Cプログラムの中からemacsを起動する）

```
char    command[MAXLENGTH] ,
        file_name[MAXLENGTH];
     ......
sprintf( command , "emacs %s &", file_name);
printf("メッセージ");
system( command );
     ......
```

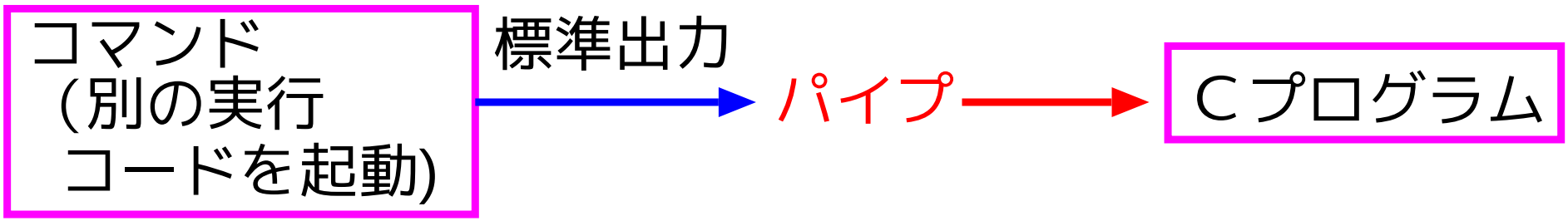
15–4 自習 Cプログラムの中からのパイプの利用

- UNIXの場合は、Cプログラムの中からパイプを使うための標準ライブラリ関数が用意されています。 (<stdio.h>)

- 実行コードを起動してその出力文字列をCプログラムの入力ストリームとして取り込むには、次の様に書く。

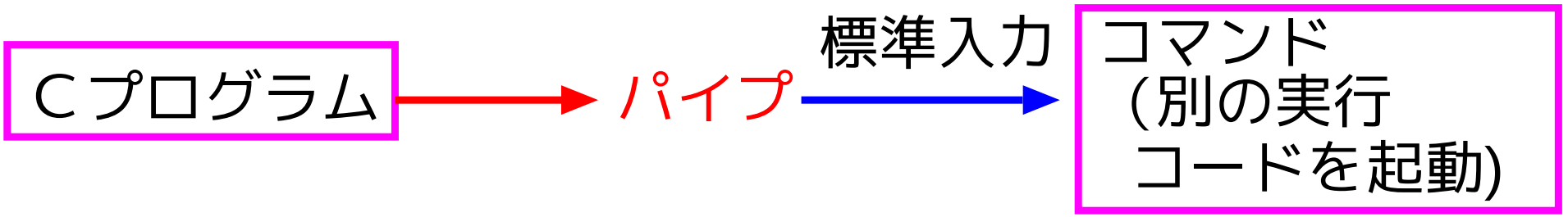
  ファイルポインタ型変数 = popen("コマンド", "r");

  あとはこのファイルポインタを指定して通常の入力を行うだけである。



- 別の実行コードを起動して、cプログラムの出力文字列をそこへの入力ストリームとして送り込むには、次の様に書く。

```
(FILE*)型変数 = popen("コマンド", "w");
```



- popen() 関数によって出来たパイプを解除するには、次の様に書く。

```
pclose( ファイルポインタ );
```

例15．5（lsコマンドの出力を全て大文字に変換して表示）

```
[motoki@x205a]$  nl file-popen-ls-to-toupper-Kelley.c
     1 /* popen("ls","r")でオープンした
                ストリームを大文字に変換して出力 */
     2 #include <stdio.h>
     3 #include <ctype.h>

     4 int main(void)
     5 {
     6   int  c;
     7   FILE *ifp;

     8   ifp = popen("ls", "r");
     9   while ((c=getc(ifp)) != EOF)
    10     putchar(toupper(c));
    11   pclose(ifp);
    12   return 0;
```

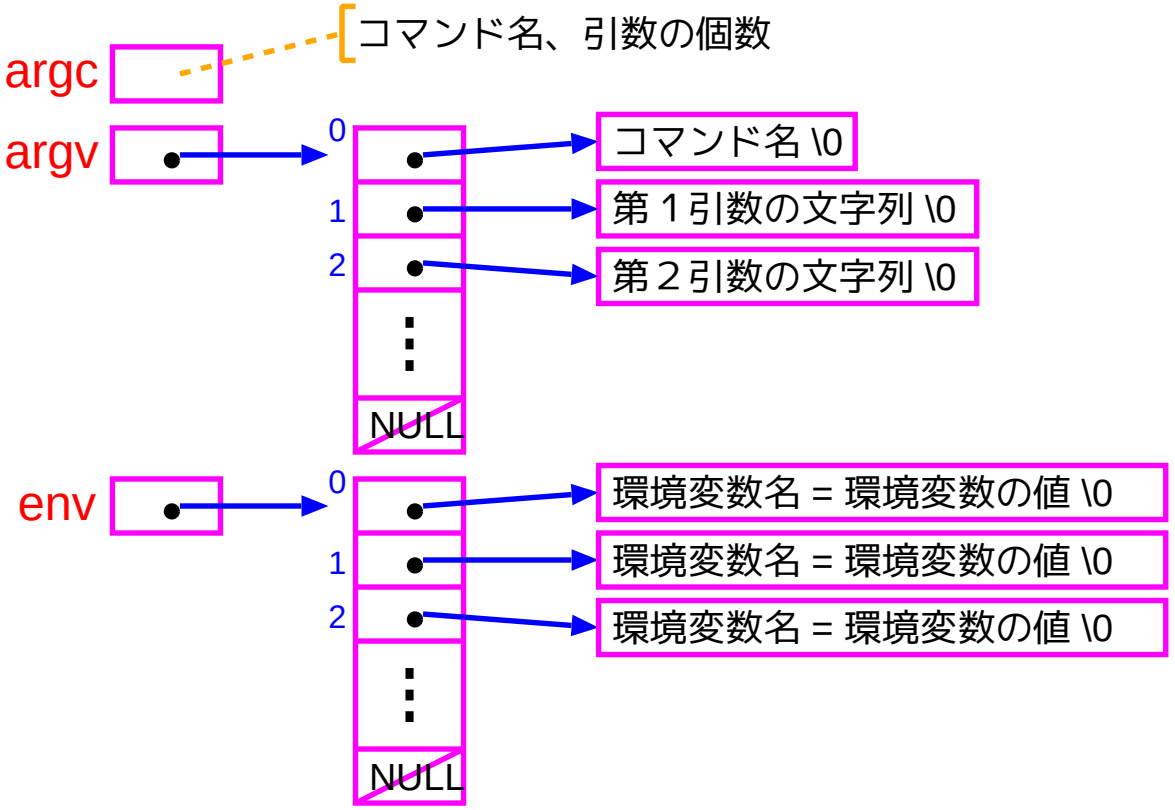
```
    13 }
[motoki@x205a]$  gcc file-popen-ls-to-toupper-Kelley.c
[motoki@x205a]$  ./a.out
COMPARE-SORT
REMARK-ON-PRINTF.C
REPORT-3.C
REPORT-3.LOG
REPORT-6
REPORT-6.TAR
A.C
A.OUT
BTREE-HEAPSORT.C
  (途中省略)
STRUCT-UNION-INT-OR-FLOAT.C
TYPEDEF-VECTOR-SPACE.C
[motoki@x205a]$
```

15–5 環境変数へのアクセス

- main関数の第3引数を指定すると、環境変数がどう設定されているかの情報を得ることが出来る。すなわち、

```
main(int argc, char *argv[], char *env[])
```

とすると、関数mainの起動直後には argc, argv, env は次の様に設定される。



- Cプログラムの中から特定の環境変数の値を得たい時のために、次の標準ライブラリ関数が用意されている。 (<stdlib.h>)

```
getenv("環境変数名");
```

例15．6（設定されている環境変数を全て表示）

```
[motoki@x205a]$  nl file-print-env-variables1-Kelley.c
     1 #include <stdio.h>

     2 int main(int argc, char *argv[], char *env[])
     3 {
     4   int i;

     5   for (i=0; env[i] != NULL; ++i)
     6     printf("%s\n", env[i]);
     7   return 0;
     8 }
```

```
[motoki@x205a]$  gcc file-print-env-variables1-Kelley.c
[motoki@x205a]$  ./a.out
LESSOPEN=|lesspipe.sh %s
USERNAME=
CANNA_SERVER=localhost
COLORTERM=gnome-terminal
HTTP_HOME=file:/usr/doc/HTML/index.html
  (途中省略)
LS_COLORS=no=00:fi=00:...(省略)...
[motoki@x205a]$
```

例15．7（指定した環境変数の値を表示）

```
[motoki@x205a]$  nl file-print-env-variables2-Kelley.c
     1 #include <stdio.h>
     2 #include <stdlib.h>

     3 int main(void)
     4 {
     5   printf("   Host: %s\n"
     6          "   User: %s\n"
     7          "  Shell: %s\n",
     8          getenv("HOST"), getenv("USER"),
                getenv("SHELL"));
     9   return 0;
    10 }
[motoki@x205a]$  gcc file-print-env-variables2-Kelley.c
[motoki@x205a]$  ./a.out
   Host: ...(省略)...
```

```
   User: motoki
  Shell: /bin/bash
[motoki@x205a]$
```
