第13章　「ファイル入出力」のまとめ

13.1　用語のまとめ

教 p.290

ストリーム　ファイル（キーボード・ディスプレイも含む）に体する入出力は、プログラムからはストリーム（stream）を使って行なう。ストリームとはディスクあるいは他の周辺機器（ディスプレイ・キーボードなど）と結び付いているデータの送出元および送出先である。

教 p.291

標準ストリーム　C 言語では、つぎの 3 つの標準ストリームがあらかじめ準備されている。

- 標準入力ストリーム（＿＿＿＿（空欄 13.1.1））
- 標準出力ストリーム（＿＿＿＿（空欄 13.1.2））
- 標準エラーストリーム（＿＿＿＿（空欄 13.1.3））

教 p.205

入出力のリダイレクト（復習）

コマンド名 < 入力ファイル名

とすると、キーボードの代わりに入力ファイルからデータが読み込まれ、

コマンド名 > 出力ファイル名

とすると、ディスプレイの代わりに出力ファイルにデータが書き込まれる。

コマンド名 >> 出力ファイル名

とすると、出力ファイルにデータが追加される形で書き込まれる。
これは C 言語ではなく OS（Windows や Unix など）の機能で、リダイレクトと呼ばれる。

教 p.291

FILE 型　FILE 型はストリームを操作するために必要な情報を表す型（通常は構造体）。stdin, stdout, stderr は、この FILE へのポインタ型（FILE *）である。ファイル・ポインタとは、この FILE へのポインタのことである。C 言語ではファイル・ポインタはストリームと同じような意味で使われる。

教 p.292

ファイルのオープン　標準入出力以外のファイルを読み書きするためには、まず準備（オープン）する必要がある。ファイルをオープンするために fopen 関数

```
FILE *fopen(const char *filename, const char *mode);
```

を使う。mode は、ファイルに対して行なう操作に応じて選ぶ（教科書 p.293 参照）。

教 p.294 **ファイルのクローズ**　ファイルを使い終わったらクローズする必要がある。fclose 関数

```
int fclose(FILE *stream);
```

を使う。

教 p.296 **ファイルからの読み出し**　ファイルからデータを読みとるには fscanf 関数

```
int fscanf(FILE *stream, const char *format, ...);
```

を使う。第 1 引数にファイル・ポインタが増えている以外は使い方は scanf 関数と同じ。またこの場合、fopen 関数の mode は "r"になる。

教 p.298 **ファイルへの書き込み**　ファイルに出力するためには、fprintf 関数

```
int fprintf(FILE *stream, const char *format, ...);
```

を使う。第 1 引数にファイル・ポインタが増えている以外は使い方は printf 関数と同じ。またこの場合、fopen 関数の mode は "w"になる。（特に）List 13-4 は fclose の使い方に注意する。"r"モードで fopen したあと、一度 fclose してから、"w"モードで fopen している。（一度クローズしないと、再オープンできない。）

教 p.302 **fgetc 関数**　ストリームから一文字読み込むためには、fgetc 関数

```
int fgetc(FILE *stream);
```

を使う。ファイル・ポインタの引数を取る以外は使い方は getchar 関数と同じ。

教 p.304 **fputc 関数**　ストリームに一文字書き込むためには、fputc 関数

```
int fputc(int c, FILE *stream);
```

を使う。第 2 引数のファイル・ポインタが増えている以外は使い方は putchar 関数と同じ。

教 p.306 **テキストとバイナリ**　数値データを正確に保存したい、あるいは、少ない容量で保存したい場合にはバイナリファイルに保存する。バイナリモードで書き込み・読み出しをするには次の関数を使う。

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

（これらの関数を使用する時は、ファイルをオープンする時もバイナリモードを指定しなければならない。）

乱数　疑似乱数（でたらめに見える数）を発生させるには rand という関数を用いる。

```
int rand(void);  /* stdlib.hが必要 */
```

rand は 0 から、ある非常に大きな数（＿＿＿＿＿＿（空欄 13.1.4）というマクロの値、bcc32 の rand の実装の場合、 32767）の間の整数を返す。

例えば、0〜1 の範囲の乱数が必要な場合は＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿（空欄 13.1.5）、0 から 5 の整数の乱数が必要なときは ＿＿＿＿＿＿（空欄 13.1.6）のような式を用いる。（後者は RAND_MAX が充分大きくないと、無視できない偏りが生じるおそれがある。）

srand 関数は疑似乱数の “種” をセットする。

```
void srand(unsigned int seed);  /* stdlib.hが必要 */
```

rand は乱数といっても、人間にとってでたらめに見えるだけで、実際には、ある計算規則に基づいて数列を計算している。（だから、疑似乱数と呼ばれる。）
例えば、rand, srand の実装例として、次のような定義が考えられる（K & R より抜粋— 実際の rand, srand の実装は処理系により異なる）。

```
1  #define RAND_MAX 32767
2
3  unsigned long int next = 1;
4
5  int rand(void) {
6      next = next * 1103515245 + 12345;
7      return (unsigned int)(next/65536) % (RAND_MAX+1);
8  }
9
10 void srand(unsinged int seed) {
11     next = seed;
12 }
```

この実装例でもわかるように、srand を実行しなければ、rand はプログラムの実行のたびに同じ乱数系列を発生しまう。
そこで例えば、現在時刻を種にして、

```
srand((unsigned)time(NULL));    /* time.hが必要 */
```

という文をプログラム中で rand を使うまえに一度だけ実行しておく。すると、プログラムの実行のたびに異なる乱数列を発生させることができる。

例: saikoro.c — 5 個のさいころを振った時の和がいくつになりやすいか、をシミュレーションする。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
```

```c
#define NUM 26

void chargraph(int n) {
    int j;
    for (j = 0; j < n; j++) {
        putchar('*');
    }
    putchar('\n');
}

int main(void) {
    int i, k;
    int result[NUM] = {0} ;  /* 大きさ 26（添字 0 〜 25）の配列 */

    srand((unsigned int)time(NULL));  /* これがないとどうなるか? */

    for (k = 0; k < 500; k++) {
        int sum = rand() % 6 + rand() % 6 + rand() % 6
                + rand() % 6 + rand() % 6;
        result[sum]++;
    }

    for (i = 0; i < NUM; i++) {
        printf("%2d␣", i);
        chargraph(result[i]);
    }

    return 0;
}
```