システムプログラミング序論
第3回
ファイル入出力

大山恵弘

ライブラリ関数の使用

ライブラリ関数の使用

- 各言語処理系は，よく利用される一連の処理やOSとのやりとりを要する処理を，ライブラリ関数としてプログラマに提供している
  - `printf`，`scanf`，`strcpy`，`strcmp`，…
- プログラム内で普通に呼び出せばよい
  - 自分で定義した関数と同じ呼び出し方
  - ただし，ヘッダファイルを include する
    - どのヘッダファイルを include する必要があるかは，マニュアルに書いてある

```
PRINTF(3)                 BSD Library Functions Manual                 PRINTF(3)

NAME
     printf, fprintf, sprintf, snprintf, asprintf, dprintf, vprintf, vfprintf, v
sprintf, vsnprintf, vasprintf, vdprintf -- formatted output conversion

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdio.h>

     int
     printf(const char * restrict format, ...);

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

```
STRCMP(3)                 BSD Library Functions Manual                 STRCMP(3)

NAME
     strcmp, strncmp -- compare strings

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <string.h>

     int
     strcmp(const char *s1, const char *s2);
:
```

include すべきファイルを知る方法

- コンパイラによる警告をよく読む

```
foo.c:3:3: warning: implicitly declaring library function
'printf' with type 'int (const char *, ...)'
  printf("Hello.¥n");
  ^
foo.c:3:3: note: please include the header <stdio.h> or
explicitly provide a declaration for 'printf'
```

- ライブラリ関数のマニュアルから情報を得る
  - マニュアルを見る方法
    - ターミナルで `man fscanf` や `man 3 printf` を実行
    - Web で "manual fscanf" や "マニュアル printf" などを検索
    - JM Project の Web ページ https://linuxjm.osdn.jp/ で検索
  - マニュアルの初めの方に書かれた `#include <...>` という行を，自分のプログラムの初めの方にそのまま書く

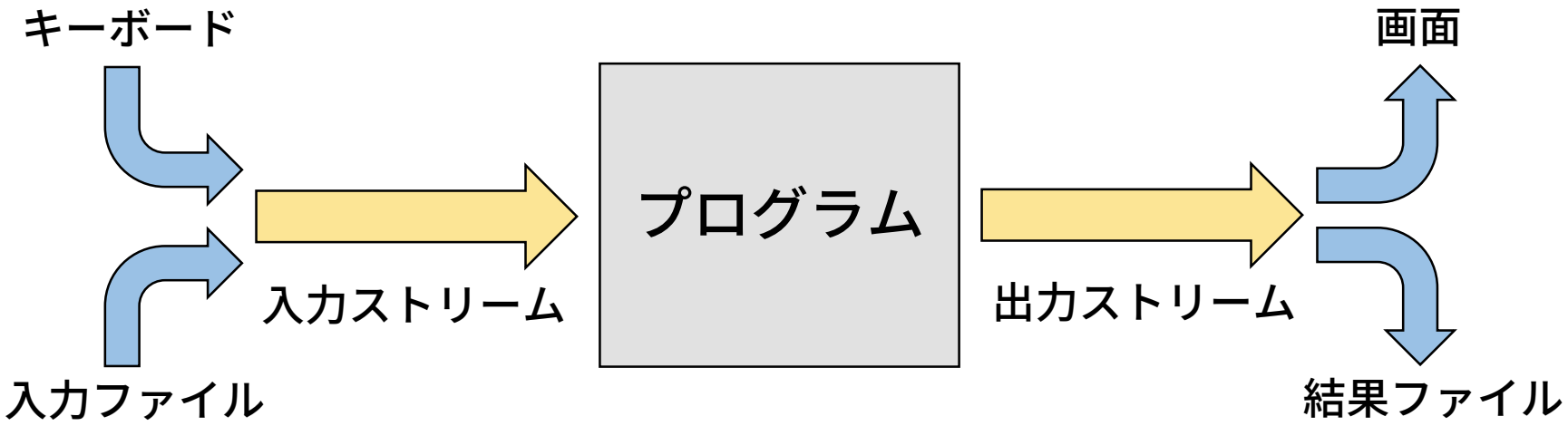
ファイル入出力

これまでのおさらい（入出力）

- これまでの入出力
  - 入力：`scanf`
  - 出力：`printf`
  - キーボードと画面（端末）
- `scanf`/`printf` は書式つき入出力
  - フォーマットを指定する
- 標準入出力が対象
  - 何も指定しなければ標準入出力はキーボードと画面

ストリームという考え方

- ストリーム（stream） = データの列
  - キーボードから打つ文字列
  - 画面に出力される文字列
  - ファイル



ファイルのストリーム

- ファイルのオープン
  - 操作を始める前に，ストリームとファイルを結び付ける
- ファイルの操作
  - 読み
  - 書き
- ファイルのクローズ
  - 操作が終わった後に，ストリームをファイルから切り離す

ファイルのオープン

- ストリームはファイルポインタで表される
  - 型は **FILE ***
- **fopen** 関数はファイルをオープンし，そのファイルに結び付けられたストリームを返す

```
FILE *fp1, *fp2;
…
fp1 = fopen("foo.txt", "r");
fp2 = fopen("bar.txt", "w");
```

fopen(ファイル名，オープンモード)
- ファイル名： オープンしたいファイル名
- オープンモード： 読み込みは"r"，書き込みは"w"，追記は"a"

- ファイルを書き込みでオープンすると，そのファイルの（オープン前までの）中身が消えることに注意

オープンの結果

- オープンに失敗すると NULL が返る
  - NULL は0に等しい
  - 読み込みでオープンしようとしたファイルがない場合
  - 書き込みでオープンしようとしたファイルが書き込み禁止の場合
  - など
- ファイルが正常にオープンされたかどうかを必ずチェックすること

```
fp = fopen("foo.txt", "r");
if (fp == NULL) {
  printf("Error!¥n");
  exit(1);
}
```

ファイルの操作（読み書き）

- ストリームに対する書式（フォーマット）付きの**scanf/printf**
  - **fscanf(ストリーム，書式文字列，引数)**
    - 例：`n = fscanf(fp, "%d %d", &num1, &num2);`
  - **fprintf(ストリーム，書式文字列，引数)**
    - 例：`fprintf(fp, "Answer: %d¥n", ans);`
- 「ストリーム」の引数には，**fopen** で得たファイルポインタなどを与える
  - オープンしなくても使える，組み込みのファイルポインタもある（後述）

ファイルのクローズ

- ストリームをファイルから切り離す

```
fclose(ストリーム)
```

- 「ストリーム」の引数には，fopen で得たファイルポインタなどを与える
- クローズされたストリームはもう使えない
  - 使った場合の動作は未定義

ファイルへの書き込みの例

```
#include <stdio.h>
int main(void)
{
  FILE *fp;
  fp = fopen("test.txt", "w");
  if (fp == NULL) {
    fprintf(stderr, "Error!¥n");
    exit(1);
  }
  fprintf(fp, "This is a sample file¥n");
  fclose(fp);
  return 0;
}
```

ファイルからの読み込みの例

```
#include <stdio.h>
int main(void)
{
  FILE *fp;
  int x;
  fp = fopen("test.txt", "r");
  if (fp == NULL) {
    fprintf(stderr, "Error!¥n");
    exit(1);
  }
  fscanf(fp, "%d", &x);
  printf("The file contains %d¥n”, x);
  fclose(fp);
  return 0;
}
```

いろいろなストリーム入出力関数（1）

- 一文字の入力
  ```
  char fgetc(FILE *fp);
  ```
- 一文字の出力
  ```
  int fputc(int c, FILE *stream);
  ```
- 行ごとの入力
  ```
  char *fgets(char *s, int size, FILE *stream);
  ```
- 行ごとの出力
  ```
  int fputs(const char *s, FILE *stream);
  ```

- **getchar，getc，gets，putchar，putc，puts** との違いを調べておくこと

いろいろなストリーム入出力関数（2）

- データ全般（バイナリデータ含む）の入力

```
int fread(void *ptr, int size,
          int nmemb, FILE *stream);
```

- データ全般（バイナリデータ含む）の出力

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

- **read，write** との違いを調べておくこと

ファイル終端への到達やエラー

- 入出力関数が，それを示す返り値を返す
  - `fgets` は `NULL` という特殊な値を返す
  - `fgetc` は `EOF` という特殊な値を返す
  - どんな場合にどんな返り値を返すかはマニュアルにある

標準入出力

- 実は，デフォルトの入出力はプログラムの起動時に暗黙にオープンされている
- 標準入力 **`stdin`**
  - 型は **`FILE *`**
  - 変更しなければ，標準入力はキーボード
- 標準出力 **`stdout`**
  - 型は **`FILE *`**
  - 変更しなければ，標準出力は画面（端末）
- 標準エラー出力 **`stderr`**
  - 型は **`FILE *`**
  - 変更しなければ，標準出力は画面（端末）

stdio.h

- 以上の入出力用関数は `stdio.h` で宣言されている
- これらの関数を使う際には `stdio.h` をincludeする

```
#include <stdio.h>
```

  - `stdio.h` はプログラムの先頭に書くおまじないではない
    - 入出力用の関数を使わないなら，大抵，必要ない
- 通常，`NULL`，`stdin`，`stdout`，`stderr` なども定義されている

scanf, printf, fscanf, fprintf

- **scanf** は **stdin** に **fscanf** をする関数

  scanf(書式文字列**,**引数**)**
  == **fscanf(stdin,** 書式文字列**,** 引数**)**

- **printf** は **stdout** に **fprintf** をする関数

  printf(書式文字列**,**引数**)**
  == **fprintf(stdout,** 書式文字列**,** 引数**)**

標準入出力の変え方

- コマンドラインで，標準入力や標準出力を変える（リダイレクトする）ことができる
- 標準入力の変え方
  - シェルで < を使う
  - 例： `a.out` の標準入力を `input.txt` にする
    ```
    $ ./a.out < input.txt
    ```
- 標準出力の変え方
  - シェルで > を使う
  - 例： `a.out` の標準出力を `output.txt` にする
    ```
    $ ./a.out > output.txt
    ```
- パイプについても調べてみよう

補足： printf での桁合わせ

- 数を出力する際に，桁数や精度を指定できる

```
$ cat digit.c
#include <stdio.h>

int main(void)
{
  printf(">>>%d<<<¥n", 12345);
  printf(">>>%9d<<<¥n", 12345);
  printf(">>>%f<<<¥n", 3.1415926535);
  printf(">>>%10.2f<<<¥n", 3.1415926535);
  return 0;
}
$ gcc digit.c
$ ./a.out
>>>12345<<<
>>>    12345<<<
>>>3.141593<<<
>>>      3.14<<<
$
```

出力のバッファリング

- printf や fprintf で出力した（はずの）データが，端末やファイルに出てこないことがある
  - かつ，プログラムの終了時などに，一気に出てくることがある
  - バッファリングという仕組みによる
    - 出すべきデータをため込んでおき，何かのタイミングで一気に出す
    - 処理を大きな単位にまとめて，入出力を高速化している
- すぐに端末やファイルにデータを出す方法はある
  - **`fflush`** 関数や **`setvbuf`** 関数を使う
  - （ファイルをクローズする，プログラムを終了する）
  - 出力先が端末の場合には，改行を出力すると，ため込んでいたデータが出てくることが多い

端末からの入力

- 同様に，普通は改行を入力するまでは，データはプログラムに渡されない
  - fgetc や getchar は1文字を読む関数だが，キーボードから（改行以外を）1文字入力しても，これらの関数にはまだそのデータは渡されない
  - 改行を入力すると，改行までの部分のデータが一気にプログラムに渡される

データの表現

符号付き整数と符号なし整数

- unsigned が付いた型の整数は符号なし整数
  - unsigned int 型など
  - 0以上，ある数以下の整数を表現
- unsigned が付かない型の整数は符号付き整数
  - int 型（signed int 型とも言う）など
  - ある負の数以上，ある正の数以下の整数を表現

整数の範囲

- 整数を32ビットで表現するときは，通常，
  - 符号付き整数： −2147483648 ～ 2147483647
    - $-2^{31} \sim 2^{31}-1$
  - 符号なし整数： 0 ～ 4294967296
    - $0 \sim 2^{32}-1$
- 整数を64ビットで表現するときは，通常，
  - 符号付き整数： −9223372036854775808 ～ 9223372036854775807
    - $-2^{63} \sim 2^{63}-1$
  - 符号なし整数： 0 ～ 18446744073709551615
    - $0 \sim 2^{64}-1$

マクロ

マクロ

- プログラム中の所定のデータを別のデータに置き換える指示を与えるための仕組み
- 使わなくてもプログラムは書けるが，他人の書いたプログラムにはよく出てくるので，知っておくとよい
- 引数をとるものととらないものがある
- 文法：
  - `#define` 置き換え前データ 置き換え後データ
  - `#define` 置き換え前データ`(...)` 置き換え後データ

使用例（1）：
引数をとらないマクロ

```
#include <stdio.h>

#define PI 3.14159

int main(void)
{
  double r;
  for (r = 1.0; r <= 10.0; r += 1.0) {
    printf("The area of a circle with radius %f = %f¥n",
           r, r * r * PI);
  }
  return 0;
}
```

使用例（2）：
引数をとるマクロ

```
#include <stdio.h>

#define PLUS(x, y) ((x) + (y))

int main(void)
{
  int a = 1;
  int b = 2;
  printf("%d + %d = %d¥n", a, b, PLUS(a, b));
  return 0;
}
```

マクロの使用 ＝ 文字列置換
マクロの使用 ≠ 関数呼び出し

```
#include <stdio.h>

#define SQUARE(x) ((x) * (x))

int main(void)
{
  int a = 0;
  int s;
  while (a <= 9) {
    s = SQUARE(++a);
    printf("Square of %d = %d¥n", a, s);
  }
  return 0;
}
```

コンパイル時の警告，
実行時のエラーへの対処

君は Segmentation fault をもう体験したか？

```
$ cat buggy.c
#include <stdio.h>

int main(void)
{
  int a = 8;
  printf("%s", a);
  return 0;
}
$ gcc buggy.c
buggy.c:6:16: warning: format specifies type 'char *'
but the argument has type 'int' [-Wformat]
  printf("%s", a);
          ~~   ^
          %d
1 warning generated.
$ ./a.out
Segmentation fault: 11
$
```

Segmentation fault とは何か

- 実行時エラーの一種
- 不正なメモリ領域へのアクセスなどで発生
- これが出たことは何を意味するか？
  → プログラムがどこか誤っている
  → 実行の途中で止まって（クラッシュして）しまった
- これを理解することが C 言語を学ぶ意義の1つ

まずは，どこで出たかを突き止めよう！

- （初級者は）printf を入れまくって調査する
- （少し上達してきたら）検査コードをあちこちに入れる
- （少し上達してきたら）デバッガを使う

プログラムが誤っていても，コンパイルと実行はできる

- ただし結果がおかしい
- 警告は出ることも出ないこともある
- 実行時エラー（segmentation fault など）が出ることもあれば，出ないこともある

```
$ cat bug1.c
#include <stdio.h>

int main(void)
{
  int a;
  printf("%d¥n", a);
  return 0;
}
$ gcc bug1.c
$ ./a.out
303399378
$
```

未初期化変数の使用

別の例

```
$ cat bug2.c
#include <stdio.h>

int foo(void)
{
  int a = 100;
}

int main(void)
{
  printf("%d¥n", foo());
  return 0;
}
$ gcc bug2.c
bug1.c:6:1: warning: control reaches end
of non-void function [-Wreturn-type]
}
^
1 warning generated.
$ ./a.out
0
$
```

return 文なしのリターン

別の例

```
$ cat bug3.c
#include <stdio.h>

int main(void)
{
  printf("%d %s %f %d¥n", 5, "Hello", 3.14);
  return 0;
}

$ gcc bug3.c
bug3.c:5:21: warning: more '%' conversions than data
arguments [-Wformat]
  printf("%d %s %f %d¥n", 5, "Hello", 3.14);
                   ~^
1 warning generated.
$ ./a.out
5 Hello 3.140000 -295589504
$
```

printf の書式文字列と引数の不整合

言いたいこと

- 警告を見る習慣をつけよう！
- 警告を消す習慣をつけよう！
- 多くの警告が出るコンパイルオプション（-Wall など）を与える習慣をつけよう！

焦らず急がず落ち着いて警告に対処するほうが，長期的には時間の節約になる