ファイル入出力（２）

これまでのおさらい（入出力）

- これまでの入出力は
  - 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
- 標準入出力を対象とする
  - 何もしなければ、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つける
- 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（FILE *)であらわされている。
- fopen関数でファイルをオープンし、ストリームを返す

```
FILE *fp;
      …
fp = fopen(“test.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：読みこみ”r”, 書き込み”w”

ファイルのオープン

- オープンに失敗した時には、0 (= NULL)が返る
  - 読み込み“r”でオープンしようとして、ファイルがなかった場合
  - 書き込み”w”で、ファイルが書き込み禁止になっていた場合
- ファイルが正常にオープンされたかチェックすること

```
fp = fopen(“test.txt”,”r”);
if(fp == NULL) printf(“error!”);
```

- 書き込みの場合は既存の内容はクリアされるので注意

fopenのモード

- fopen のモード（下の表）
- バイナリモード（"wb"、"rb" のように指定）

モード	動作	ファイルがあるとき	ファイルがないとき
"r"	読み出し専用	正常	エラー（NULL返却）
"w"	書き込み専用	サイズを 0 にする（上書き）	新規作成
"a"	追加書き込み専用	最後に追加する	新規作成
"r+"	読み込みと書き込み	正常	エラー（NULL返却）
"w+"	書き込みと読み込み	サイズを 0 にする（上書き）	新規作成
"a+"	読み込みと追加書き込み	最後に追加する	新規作成

ファイルの読み書き

- ストリームに対する書式つきのscanf/printf

  `fscanf(ストリーム、フォーマット、引数...)`

  `fprintf(ストリーム、フォーマット、引数...)`

- ストリームは、`fopen`で得たファイルポインタ

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)`

- ストリームは、fopenで得たファイルポインタ
- 成功した場合は１、失敗した場合は0を返す

- 書き込んだ内容は、fcloseしないと全部かきこまれないので注意。

stdio.h

- 以上の入出力ストリームを扱う関数は、stdio.hに定義されているので、stdio.hをincludeすることをわすれないこと

```
#include <stdio.h>
  …
```

- NULL（＝０）も定義されている

例

```
#include <stdio.h>

main()
{
    FILE *fp;
    fp = fopen(“test.txt”,”w”);
    if(fp == NULL){
        printf(“error!!”);
        exit(1);
    }
    fprintf(fp,”this is sample¥n”);
    fclose(fp);
    return 0;
}
```

例

```
#include <stdio.h>

main()
{
    FILE *fp; int x;
    fp = fopen("test.txt","r");
    if(fp == NULL){
        printf("error!!");
        exit(1);
    }
    fscanf(fp,"%d",&x);
    printf("file contain %d¥n",x);
    fclose(fp);
    return 0;
}
```

いろいろなストリーム入出力関数

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

標準入出力

- 実は、defaultの入出力はプログラムの起動時にオープンされている
- 標準入力　`FILE *stdin`
  - なにもしなければ、キーボード
- 標準出力　`FILE *stdout`
  - なにもしなければ、画面
- これらの変数`stdin`,`stdout`は、`stdio.h`に定義されている。使う時には`stdio.h`を`include`.

scanf/printfとfscanf/fprintf

- scanfは、stdinにfscanfをする関数

```
scanf(フォーマット、引数) ＝＝
        fscanf(stdin,フォーマット、引数)
```

- printfは、stdoutにfprintfをする関数

```
printf(フォーマット、引数) ＝＝
        fprintf(stdout,フォーマット、引数)
```

いろいろなストリーム入出力関数

- データ（バイナリ、構造体）の入力

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

- データ（バイナリ、構造体）の出力

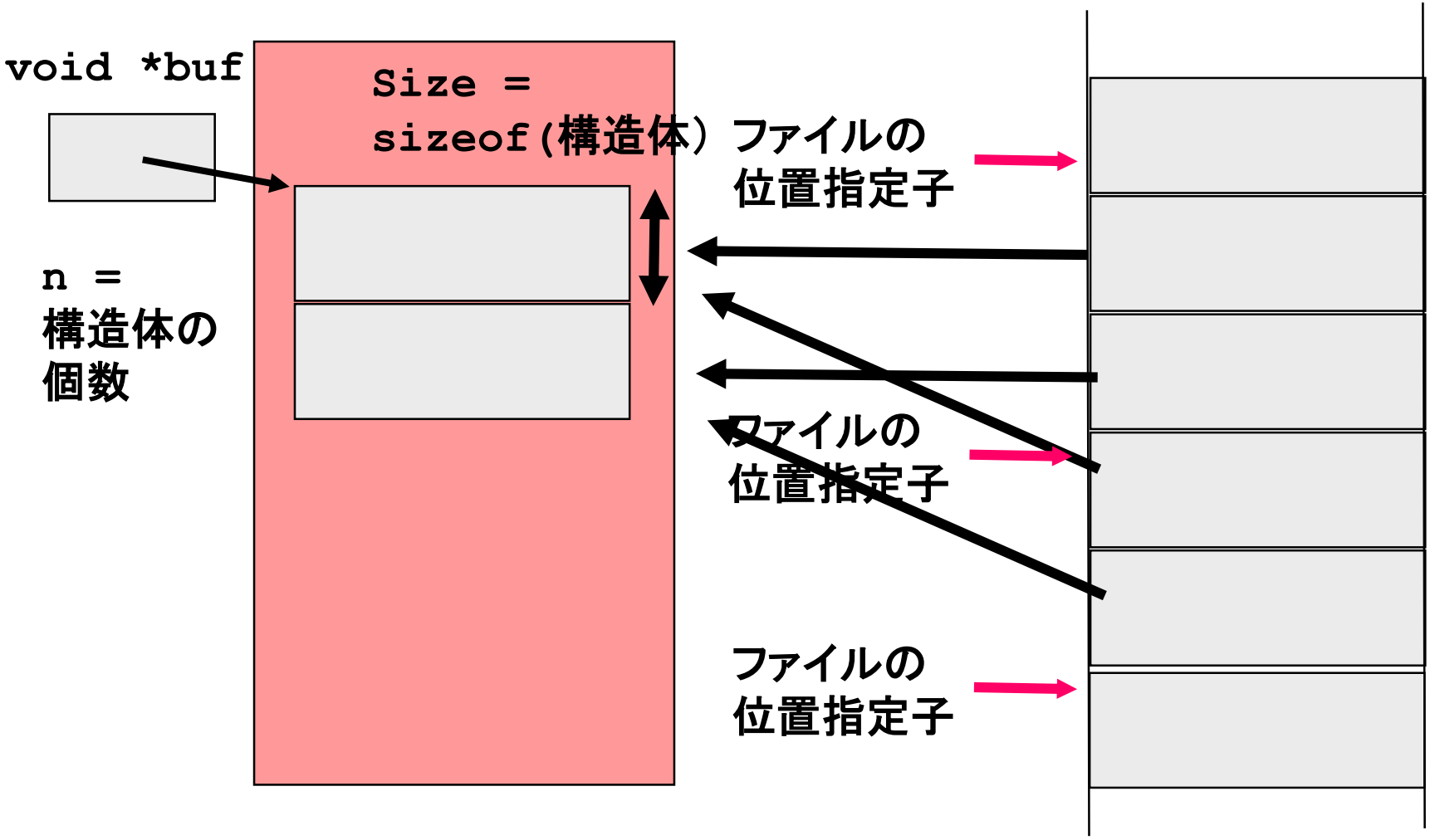
```
int fwrite(const  void  *ptr, int  size,
     int  nmemb,  FILE *stream);
```

- read, write との違いを調べておくこと。

fread関数

- ファイルfpからsizeバイトのデータをn個読み込み、bufに格納します。
- 返り値は、読み込んだデータの個数、EOFの時は0
  - したがって、EOFかエラーかは、feofとferrorで判定する。
- ファイル位置指示子を読み込んだデータバイト分進めます。
  - エラーが発生した場合にはファイル位置指示子の値は不定です。

```
#include <stdio.h>
 size_t fread(void *buf, size_t size,
  size_t n, FILE *fp);
```


void *buf
Size =
sizeof(構造体）
n =
構造体の
個数
ファイルの
位置指定子
ファイルの
位置指定子
ファイルの
位置指定子

関数fwrite

- bufからファイルfpへsizeバイトのデータをn個書き込みます。
- エラーはfreadと同じ
- ファイル位置指示子を書き込んだデータバイト分進めます。
  - エラーが発生した場合にはファイル位置指示子の値は不定です。

```
#include <stdio.h>
  size_t fwrite(const void *buf, size_t
  size, size_t n, FILE *fp);
```

fseek、ftell

- *stream によって指定されたストリームにおいて、ファイル位置表示子 (file position indicator) をセットする*
- *whence* 引数が SEEK_SET, SEEK_END, SEEK_CUR

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
```

- ftell
  - *stream* によって指定されたストリームにおける、ファイル位置表示子の現時点での値を与える

fflush

- ユーザー空間でバッファリングされているすべてのデータを与えられた出力に書き出す (フラッシュする)
- あるいはストリーム *stream* の下位にある書き込み関数を用いてこのストリームを更新する。ストリームは開いた状態のままであり、この関数によって何の影響も受けない。

```
#include <stdio.h>
int fflush(FILE *stream);
```

プログラム例

- プログラム１：データファイル（テキスト、以前の例と同じ）を読んで、バイナリデータファイルを作る
- プログラム２：バイナリデータファイルから、データを読んで検索する

プログラム1

テキストファイルからバイナリファイルを作る

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

struct record {
  char name[10];
  int point;
};

int main(int argc, char *argv[])
{
  FILE *fp;
  FILE *fq;
  int x,r;
  char name[10], buf[256];
  struct record d;

  if (argc != 2) {
    printf("missing file argument¥n");
    return 1;
```

プログラム1

テキストファイルからバイナリファイルを作る

```
fp = fopen(argv[1], "r");
if (fp == NULL) {
  printf("can't open %s¥n", argv[1]);
  return 1;
}
fq = fopen("data-file","w");
if (fq == NULL) {
  printf("can't open data-file¥n");
  return 1;
}
while (fgets(buf, sizeof(buf), fp) != NULL){
  sscanf(buf, "%s %d", name, &x);
  d.point = x;
  strcpy(d.name,name);
  r = fwrite(&d,sizeof(d),1,fq);
  if(r != 1){
    printf("write error¥n");
    exit(1);
  }
  printf("name=%s, point=%d stored¥n",d.name,d.point);
}
```

プログラム1

テキストファイルからバイナリファイルを作る

```
while (fgets(buf, sizeof(buf), fp) != N
    sscanf(buf, "%s %d", name, &x);
    d.point = x;
    strcpy(d.name,name);
    r = fwrite(&d,sizeof(d),1,fq);
    if(r != 1){
      printf("write error¥n");
      exit(1);
    }
    printf("name=%s, point=%d stored¥n",d.name,d.point);
  }
  fclose(fp);
  fclose(fq);

  return 0;
}
```

プログラム2

バイナリファイルから検索

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

struct record {
  char name[10];
  int point;
};

int main(int argc, char *argv[])
{
  FILE *fp;
  int r,x;
  char *name;
  struct record d;

  if (argc != 2) {
    printf("missing name argument¥n");
    return 1;
  }
  name = argv[1];
```

プログラム2

バイナリファイルから検索

```
 fp = fopen("data-file","r");
 if (fp == NULL) {
    printf("can't open data-file¥n");
    return 1;
  }

  x = -1;
  while(fread(&d,sizeof(d),1,fp) == 1){
    if(strcmp(d.name,name) == 0){
      x = d.point;
      break;
    }
  }
  if(x >= 0)
    printf("name '%s', point = %d¥n",name,d.point);
  else
    printf("name '%s', data is not found¥n",name);

  fclose(fp);

  return 0;
}
```

考えてみましょう

- このプログラムでは、１個1個データを読み込んでいますが、数十個ずずデータを読み込んだほうが効率がいいです。どうすればいいでしょうか？

プリプロセッサと分割コンパイル

プリプロセッサ

- なぜ、#include <stdio.h> と書くのか
  - そもそも、どういう意味？
- 定数はどうやってかいておくのがいいのか
  - 数字でかいておくと、意味を忘れてしまう
- 簡単な関数でも、いちいち定義しておくのか
  - ちょっとした関数であれば、関数にすると遅い？
  - よく出る数式は、関数にしておきたい

プリプロセッサ　#include

```
#include <file名>
#include “file名”
```

- 現れた位置にファイルを埋め込む
- 通常、ヘッダーファイル xxx.hを埋め込む
  - 標準関数のプロトタイプ宣言
  - マクロ名定義
  - 関数型マクロ定義
  - typedef での型名の宣言

- #include <ファイル名>とした場合，標準ライブラリとして開発環境が指定しているディレクトリ（演習室のシステムでは/usr/include）内のファイルを読み込むことになります
.

プリプロセッサ　#define

- 識別子を置き換えるマクロ
  - 通常、定数（または式）に意味のある名前をつける

```
#define 識別子　置き換えるパターン
#define N 100
```

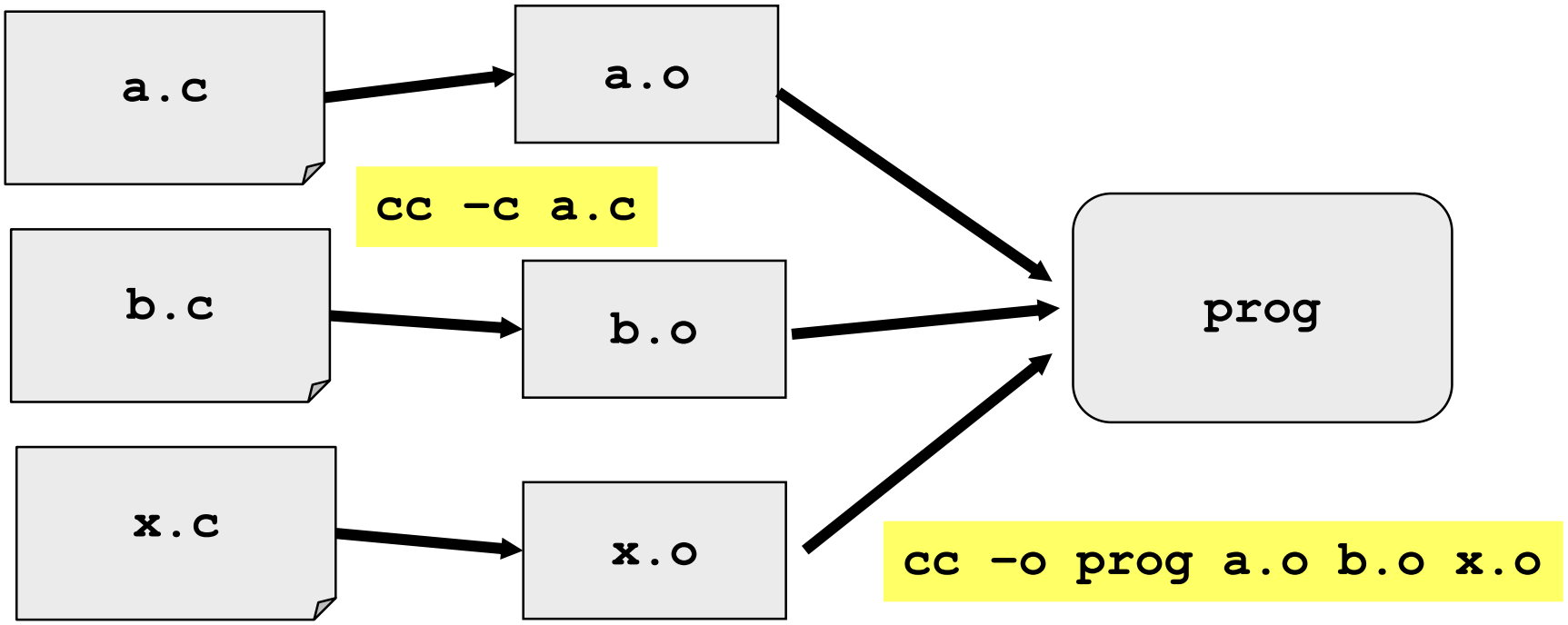
- 関数型マクロ
  - 関数の形でかいて、それを展開する

```
#define マクロ名（パラメータ）置き換えるパターン
#define add(x,y)  x+y
```

いまでは、inline宣言のほうがいいかも。

分割コンパイル

- 大きいプログラムを書く場合には機能的に分かれる部分を別々のファイルにしておくことが普通
  - 一部分を直した場合、それだけをコンパイル



分割コンパイル

```
#include <stdio.h>
int add(int x);
int a;
int main(void)
{
 int b;
 scanf("%d", &a);
 scanf("%d", &b);
 printf("a + b = %d¥=n",
    add(b));
 return 0;
}

int add(int x)
{
return x + a;
}
```

```
#include <stdio.h>
extern int add(int x);
int a;

int main(void)
{
 int b;
 scanf("%d", &a);
 scanf("%d", &b);
 printf("a + b = %d¥=n",
    add(b));
 return 0;
}
```

```
extern int a;

int add(int x)
{
return x + a;
}
```

外部定義の参照

- extern 宣言をつける
  - 他のファイルに定義されていることを明示

```
extern int a; /* 変数の場合  */

extern int add(x,y); /* 関数の場合 */
```

- 関数の場合は、プロトタイプ宣言で可

- このような外部定義や、マクロを一つのファイルにまとめ（ヘッダファイル*.h）として、参照するファイルに#includeしておくのが、普通

コンパイルの仕方

- -c をつけると、中間オブジェクト *.oができる
- これを、ccでコンパイル（リンク）する。
- -o は結果ファイルの名前を指定

```
% cc -o prog main.c add.c
```

```
% cc -c main.c
% cc -c add.c
% cc -o prog main.o add.o
```

makeコマンドとMakefile

- どのようにコンパイルするかを指定するツール
  - 多くのソースファイルやヘッダファイルからなるプログラムの一部を変更した際に，どのファイルが変更されたか，また変更されたファイルどうしの依存性をユーザが判断して再コンパイルするのは面倒です．
  - そこで，このような処理を自動的に行ってくれるのがUNIX のmake コマンド

ターゲット：ターゲットが依存するもの
**　　ターゲットを作るためのコマンド**

Makeのルール

- 「ターゲット」や「ターゲットが依存するもの」の部分には複数のファイル名を書くことができる.
- ターゲット」が存在しなかったり，「ターゲットが依存するもの」より古い時には，「ターゲットを作るためのコマンド」を実行して「ターゲット」を作り直す.
- 「ターゲットを作るためのコマンド」の行はTabで始める．複数の行を書いてもよい．Tab で始まらない行が来るとルールの終わりになる.

```
prog: main.o add.o
	cc -o prog main.o add.o
main.o: main.c
	cc -c main.c
add.o: add.c
	cc -c add.c
```

- prog はmain.o とadd.o に依存している.
  - prog が存在しなかったり，main.o やadd.o より古い場合には cc -o prog main.o add.o としてprog を作り直す.
- main.o はmain.c に依存している.
  - main.o が存在しなかったり，main.c より古い場合にはcc -c main.c としてmain.o を作り直す.
- add.o はadd.c に依存している.
  - add.o が存在しなかったり，add.c より古い場合にはcc -c add.c としてadd.o を作り直すとなります.

makeコマンド

- MakeのルールはMakefileまたはmakefileというファイル名にしておく

```
% make
```

- -fオプションで、makefileの名前も指定もできる。