

UNIXシステムプログラミング

第4回 文字と文字列の操作

2019年10月18日

情報工学科

寺岡文男

文字・文字列の操作のねらい

- インタラクティブなソフトウェアでは文字や文字列の処理が必要
 - 課題2 (バッファキャッシュ) のコマンド処理の部分
 - 課題3 で出題予定のshell作成におけるユーザ入力の処理
- 文字処理の便利な関数もあるが、中身の理解も大事



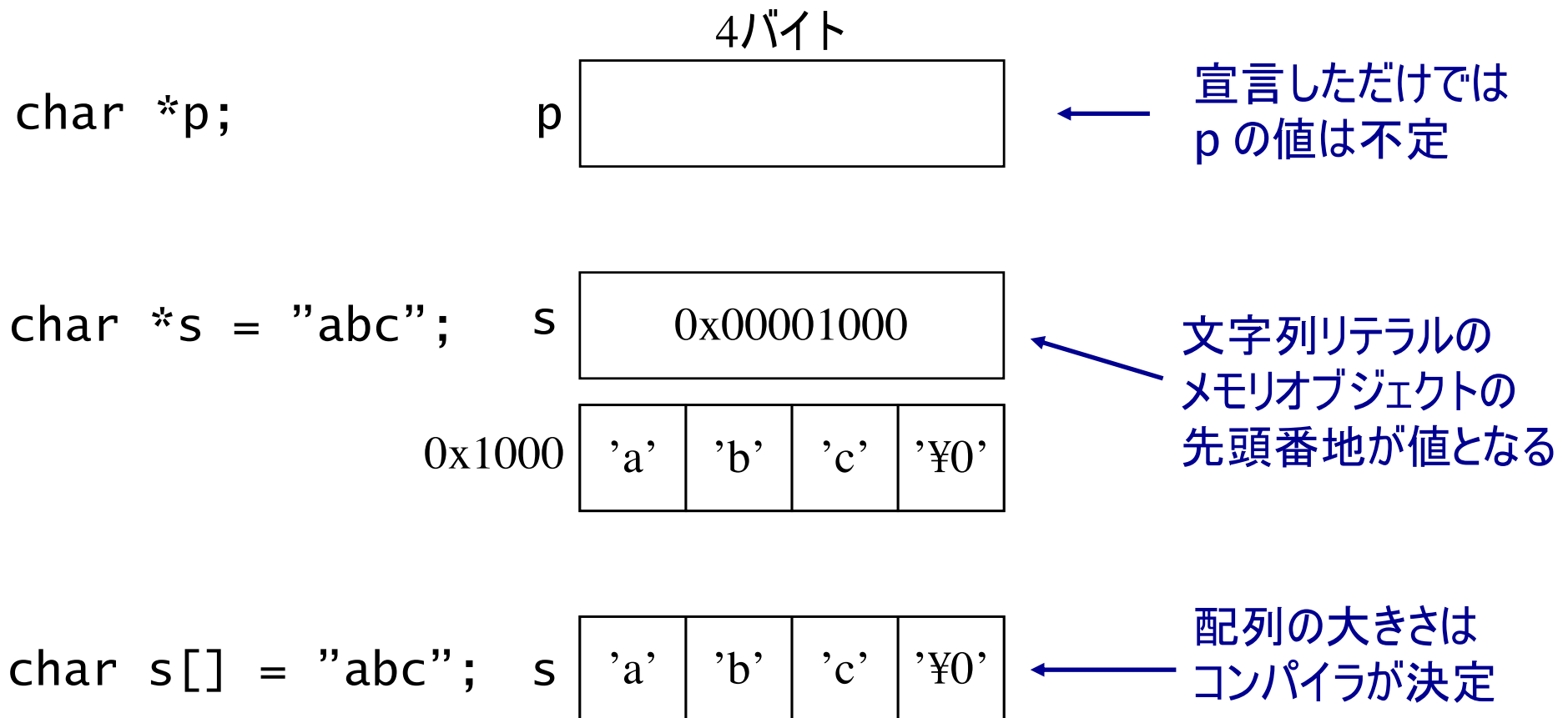
- 文字と文字列についての復習
- argc, argvについての復習
- よく使われる文字・文字列処理の関数の概要
- 演習

メモリマップ + メモリの内容(16進数で)

を常にイメージしよう

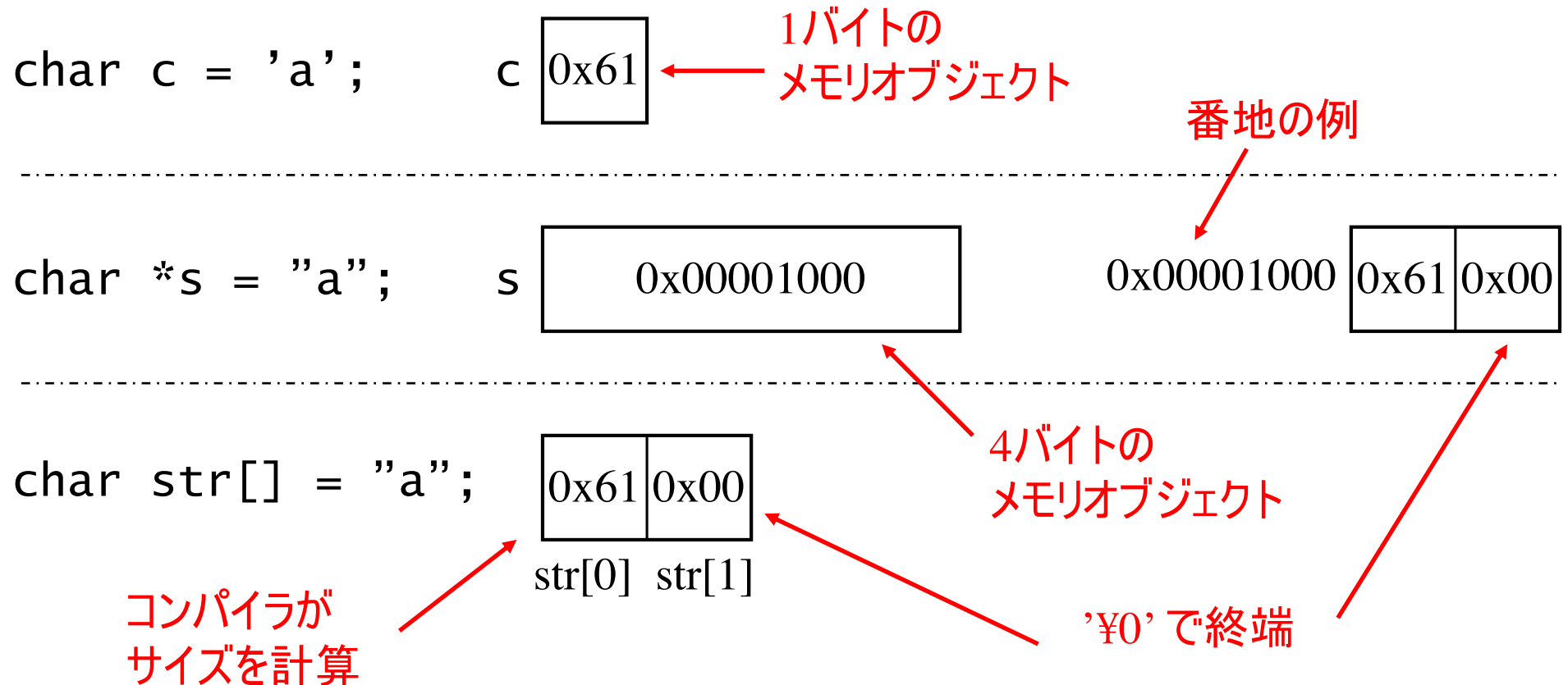
ポインタに関する注意

- ポインタ変数自体のメモリオブジェクトと、ポインタ変数が指すメモリオブジェクトを区別するように



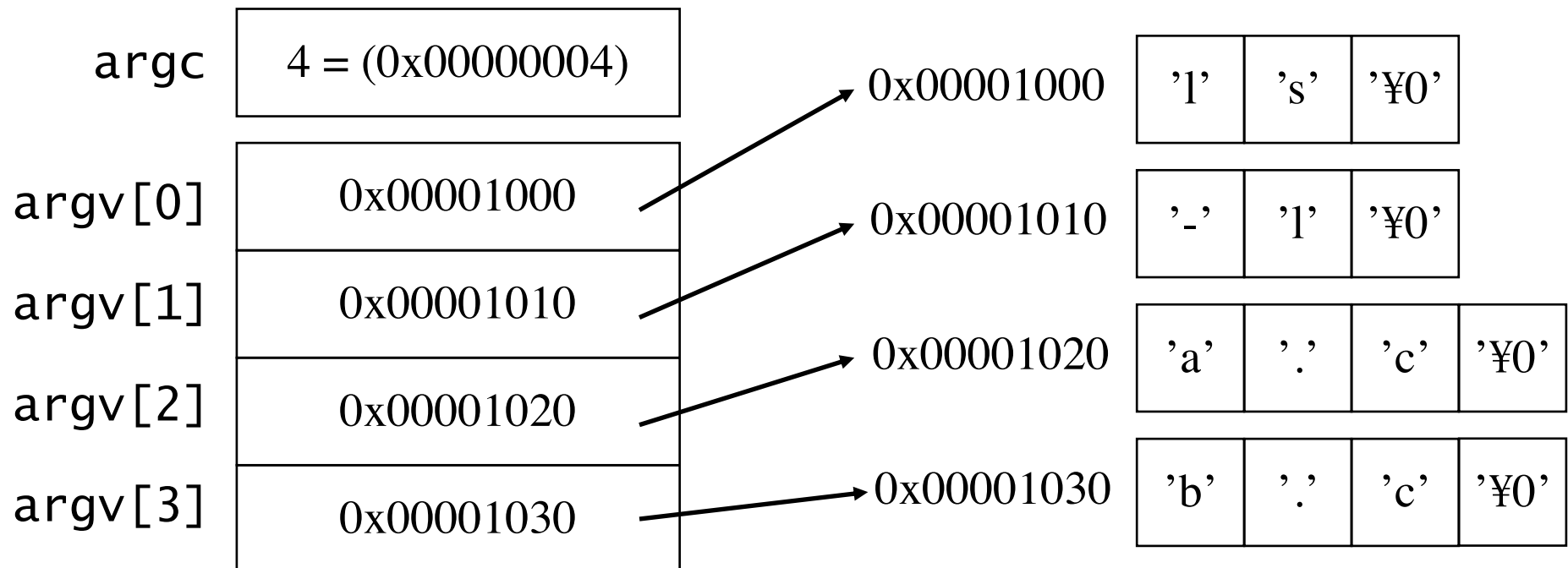
文字と文字列

- ・ 'a' は文字定数, "abc" や "a" は文字列リテラル
- ・ 'a' と "a" は同じように見えるがまったく別のもの



argc と argv

- `main(int argc, char *argv[])`
- `argc` はコマンドラインの文字列の数
 - `"ls -l a.c b.c"` → `argc` は 4
- `argv[]` は各文字列の先頭番地を示す



manコマンドの活用

- ライブラリ関数やシステムコールを使う場合は, manコマンドで仕様を調べるように.
 - 解説書やWebに頼らず, まずmanコマンドを使おう.
 - 英語(原語)で読もう.
- 構文や機能の確認
- 返り値の確認
 - エラーが発生した場合はどのような値が返るのかを確認
 - エラーをきちんと処理するプログラムを書くように心がける
- 安全な関数かどうかを確認
 - バッファオーバーフローなどを起こさないか, など.

ストリーム

- ストリーム: 入出力先のファイル
 - UNIXでは通常のファイルやデバイスなどが“ファイル”として抽象化される.
 - ストリームが通常のファイルを指す → ファイルへの入出力
 - ストリームがデバイスを指す → デバイスへの入出力
- プログラム実行開始時には以下の3つのストリームがあらかじめ設定されている.
 - `stdin` : 標準入力. 初期状態はキーボード.
 - `stdout` : 標準出力. 初期状態は画面.
 - `stderr` : 標準エラー出力. 初期状態は画面.
 - `stdin`, `stdout`, `stderr`は予約語.
 - “`#include <stdio.h>`”が必要

文字の入出力

- 以下の例では, `const`, `restrict`などの型修飾子は省略

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

```
int getchar();
```

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

- 標準入力、または`stream`からの1文字入力

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

- `stream`への1文字(`c`)プッシュバック

```
int putchar(int c);
```

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

- 標準出力、または`stream`への`c`の出力

型修飾子 (type-qualifier)

- **restrict**

- restrict修飾されたポインタ型が直接的または間接的に指す先について、同一関数/ブロック内の別ポインタが指さないことをコンパイラに伝える.

- **const**

- const修飾された変数が定数であることを示す.

- **volatile**

- volatile修飾された変数は処理系に不明な形で値が変更される可能性があり、それへのアクセスは処理系依存となる.
- volatile修飾された変数を最適化処理の対象にしないようにコンパイラに伝える.

なぜungetc()があるのか？

- キーボードからの入力 “ls|wc” を, getc() を用いて構文解析する場合を考える.
 - getc() で1文字ずつ読み込み, 解析する
 - “ls” と “|” と “wc” に分解したい
 - コマンド名1, パイプ文字, コマンド名2
- “|” を読み込んではいじめてコマンド名1が終了したことがわかる
 - “|” は次の要素
- “ungetc()” で入力バッファに戻す
 - 次の構文解析ループで “|” が認識される

書式指定の文字列入力

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

```
int scanf(char *format, ...);
```

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

```
int sscanf(char *str, char *format, ...);
```

- 標準入力, またはstream, またはメモリ領域からの書式指定の入力

scanf()についての個人的感想

- キーボードからのscanf()やfscanf()は思ったとおり動かないことがある
 - 入力バッファやCR (Enter)の扱い方のため?
- キーボードからの書式指定入力の場合, まず行全体を読み込み, sscanf()を使うことが多い.
 - 行単位の入力: fgets() (後述)

書式指定の文字列出力

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

```
int printf(char *format, ...);
```

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

```
int snprintf(char *str, size_t size,  
            char *format, ...);
```

- 標準出力, またはstream, またはメモリ領域への書式指定の出力
- `snprintf()`はメモリ領域のサイズを指定
→ バッファオーバーフローなし

```
fprintf(stderr, ...);: エラーメッセージ表示
```

行単位の入出力

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

```
char *fgets(char *str, int size,  
            FILE *stream);
```

- streamからstrへの1行の入力(改行文字まで)
- 末尾に '¥0' が付加される
- strはメモリ領域を指していなければならない
- **メモリ領域のサイズを指定→バッファオーバーフローなし**
- 最大 size - 1 文字まで

```
int puts(char *str);
```

```
int fputs(char *str, FILE *stream);
```

- 標準出力, またはstreamへの出力 ('¥0' まで)

文字列の長さ

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

```
size_t strlen(char *s);
```

- 文字列の長さ: NULL文字までの文字数
(空白文字も含む)

文字列の比較

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

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

```
int strncmp(char *s1, char *s2,  
            size_t len);
```

```
int strcasecmp(char *s1, char *s2);
```

```
int strncasecmp(char *s1, char *s2,  
                size_t len);
```

- 文字列s1が, 文字列s2よりも大きい, 等しい, 小さいかを返す
- strncmp(), strncasecmp()は最大len個文字を比較
- strcasecmp(), strncasecmp()は大文字・小文字の区別無し

文字列の複製

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

```
char *strcpy(char *dst, char *src);
```

```
char *strncpy(char *dst, char *src,  
               size_t len);
```

- srcが指すメモリ領域の文字列をdstが指すメモリ領域へコピー
- strncpy()は最大lenバイトをコピー

間違いの例

```
01: char *src = "string";
02: char *dst;
03: ...
04: strcpy(dst, src);
```

- dstの値は不定
- メモリオブジェクトを指していない

strcpy()はdstのためのメモリオブジェクトを自動的に割り当ててくれない

正しい例

```
01: char *src = "string";
02: char dst[80];
03: ...
04: strcpy(dst, src);
```

メモリオブジェクトを用意している

オーバフローを防ぐ

前ページの例

```
01:      char *src = "string";
02:      char dst[80];
03:      ...
04:      strcpy(dst, src);
```

もし *src が指す文字列長が
80バイトを超えたらどうなるか?

オーバフローを防ぐには

```
01:      char *src = "string";
02:      char dst[80];
03:      ...
04:      strncpy(dst, src, sizeof dst);
```

strcpy()ではなく
strncpy()を使用

dst が指すメモリオブジェクトの
サイズを超えないように指示

さらなる注意

前ページの例

```
01:      char *src = "string";
02:      char dst[80];
03:      ...
04:      strncpy(dst, src, sizeof dst);
```

- *srcが指す文字列が80文字を超えると
dst[]の文字列は '¥0' で終端しない

注意深いコーディングの例

```
01:      char *src = "string";
02:      char dst[80];
03:      ...
04:      memset(dst, 0, sizeof dst);
05:      strncpy(dst, src, sizeof dst - 1);
```

dst[]のすべての要素を
0 ('¥0')で初期化しておく

'¥0' のための領域をとっておく

文字列の連結

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

```
char *strcat(char *s, char *append);
```

```
char *strncat(char *s, char *append,  
              size_t count);
```

- 文字列sに、文字列appendを連結
- strncat()は最大countバイトの文字のみを連結
- sが指すメモリ領域には連結後の文字列を格納するのに十分なサイズが必要


間違いの例

```
01:    char *str1 = "It is";
02:    char *str2 = " a fine day";
03:    ...
04:    strcat(str1, str2);    Q: なぜ間違いか?
```

正しい例

```
01:    char *str1 = "It is";
02:    char *str2 = " a fine day";
03:    char str3[80];
04:    ...
05:    strncpy(str3, str1, sizeof str1);
06:    strncat(str3, str2, sizeof str3 -
               strlen(str1) - 1);
```

Q: なぜ“-1 ”が必要か?



動的に領域を確保

前ページの例

```
01:      char *str1 = "It is";
02:      char *str2 = " a fine day";
03:      char str3[80];
04:      ...
05:      strncpy(str3, str1, sizeof str3);
06:      strncat(str3, str2, sizeof str3 - sizeof str1);
```

```
01:      char *str1 = "It is";
02:      char *str2 = " a fine day";
03:      char *str3;
04:      ...
05:      str3 = (char *)malloc(strlen(str1) +
                                strlen(str2) + 1);
06:      strcpy(str3, str1);
07:      strcat(str3, str2);
```

Q: なぜ“+1”が必要か?



本来はエラーチェックが必要

文字列中の文字の検索

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

```
char *strchr(char *s, int c);
```

```
char *strrchr(char *s, int c);
```

- 文字列s中に文字cが含まれていれば、そこへのポインタを返す
- なければNULLが返る
- strchr()は先頭から, strrchr()は末尾から検索する

文字列中の文字列の検索

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

```
char *strstr(char *big, char *little);
```

```
char *strcasestr(char *big,  
                 char *little);
```

```
char *strnstr(char *big,  
              char *little, size_t len);
```

- 文字列bigに、文字列littleが含まれている場合、そこへのポインタを返す
- strcasestr()は大文字・小文字の区別無し
- strnstr()は最大len文字の検索

文字列数値の整数への変換

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

```
#include <limits.h>
```

```
long strtol(char *nptr,  
            char **endptr, int base);
```

- nptrが指す文字表現の数値を、base進法の整数に変換し、その値を返す
- endptrがNULLでない場合、数値表現でない文字が現れたところへのポインタが*endptrに設定される

文字のクラス分け・大文字と小文字の変換

```
#include <ctype.h>
int isalnum(int c);
    - 英数字か?
int isalpha(int c);
    - アルファベットか?
int isblank(int c);
    - 空白 or タブか?
int isdigit(int c);
    - 数字か?
int islower(int c);
    - 小文字か?
int isprint(int c);
    - 表示可能か?
```

```
int isspace(int c);
    - 空白文字か?
int isupper(int c);
    - 大文字か?
int tolower(int c);
    - 小文字へ変換
int toupper(int c);
    - 大文字へ変換
```

エラー処理

- プログラム実行中にエラーはよく起こる
 - ユーザの入力間違い, 存在しないファイルを指定, etc.
- エラーが発生しても正しく動くプログラムを作るように心がけることが大切

エラー処理の例

```
01: func()
02: {
03:     FILE *st;
04:     char lbuf[80];
05:     ...
06:     if (fgets(lbuf, sizeof(lbuf), st) == NULL) {
07:         /* エラー または EOF */
08:         if (ferror(st)) { ← この書き方に注意
09:             /* エラー処理 */
10:             fprintf(stderr, "input error¥n");
11:             return -1; ← この関数の呼び出し元に
12:         } else          エラーの発生を通知
13:             return 0; /* EOF */
14:     }
15:     /* 正常な場合の処理の続き */
16:     ...
17: }
```

課題2におけるコマンド処理：間違いの例

```
01:      char cmd[16];
02:      ...
03:      ... /* cmd[]に入力されたコマンド名を設定する */
04:      ...
05:      switch (cmd) {
06:      case "help":
07:          /* helpコマンドの処理 */
08:          break;
09:      case "init":
10:          /* initコマンドの処理 */
11:          break;
12:      ...
13:      }
```

Q: なぜ間違いか？

課題2におけるコマンド処理: 洗練されていない例

```
01: char cmd[16];
02: ...
03: ... /* cmd[]に入力されたコマンド名を設定する */
04: ...
05: if (strcmp(cmd, "help") == 0) {
06:     /* helpコマンドの処理 */
07:     ...
08: } else if (strcmp(cmd, "init") == 0) {
09:     /* initコマンドの処理 */
10:     ...
11: } else if (strcmp(cmd, ...) == 0) {
```

定義されたコマンド名と1つずつ比較する



課題2におけるコマンド処理: テーブルを使った例 (1/2)

```
01: void help_proc(int, char *[]), ...;
02: ...    /* 関数のプロトタイプ宣言 */
03: ...
04: struct command_table {
05:     char *cmd;                      /* 定義されたコマンド名 */
06:     void (*func)(int, char *[]);    /* それぞれの処理関数 */
07: } cmd_tbl[] = {
08:     {"help",  help_proc},           /* helpコマンド */
09:     {"init",  init_proc},           /* initコマンド */
10:     {"buf",   buf_proc},            /* bufコマンド */
11:     ...
12:     {NULL,    NULL}                 /* cmd_tblの終端 */
13: };
```

課題2におけるコマンド処理: テーブルを使った例 (2/2)

```
01: struct command_table *p;
02: int ac;
03: char *av[16];
04: ...
05: ... /* 入力を解析し、ac, avを設定 */
06: ...
07: for (p = cmd_tbl; p->cmd; p++)
08:     if (strcmp(av[0], p->cmd) == 0) {
09:         (*p->func)(ac, av);
10:         break;
11:     }
12: if (p->cmd == NULL)
13:     fprintf(stderr, "unknown command¥n");
14: ...
```

コマンドテーブルを検索

文字列を比較

対応する関数を呼び出す

演習

- 3章の練習問題3

- 標準入力から文字列を読み込み, `int argc, char *argv[]` を設定する関数 `void getargs(int *argc, char *argv[])` を作成しなさい.
- 文字列前後の余分な空白やタブは省きなさい.
- その他の引数を加えてもよい.
 - 例: `char lbuf[256]` に1行分の入力を読み込んでおき, この配列の先頭番地を引数とする.
`void getargs(char *lbuf, int *argc, char *argv[])`
- 課題3(shellの作成)でも利用可能