プログラミング環境 第11回

久野 靖 *

1991.12.17

16 Cによるプログラミング:データ構造編

16.1 宣言の構文

前回は宣言がどういう形をしているかを適当にごまかしたので、今回はもう少しそこを詳しく見てみる。実は宣言は次のような形をしている。

宣言 ＝型指定 (宣言子 [= 初期設定]),... ;
型指定 ＝[unsigned](int |char |short |long)|float |double | 構造体型
宣言子 ＝名前 |*宣言子 | 宣言子 [整定数] | 宣言子 () |(宣言子)

ここで*はポインタ、[] は配列、() は関数を表す。実はこれらを組み合わせることで Pascal のような様々な型宣言ができてしまう。少し例を見ていただこう。

```
int x;          x は整数の変数
int *x;         x は整数へのポインタ
int x();        x は整数を返す関数
int x[10];      x は大きさ 10 の整数配列
int **x();      x は整数へのポインタへのポインタ
int *x();       x は整数へのポインタを返す関数
int (*x)();     x は整数を返す関数へのポインタ
int *x[10];     x は大きさ 10 の、整数へのポインタの配列
```

この中に「大きさ 10 の整数配列へのポインタ」がなかったので不思議に思われた方がいらっしゃるかも知れない。その訳は次節で。

16.2 ポインタ、配列、文字列

C ではポインタに対しては基本的に次の 2 つの演算が用意されている。

```
  &x    -- x のアドレスを取る。
 *p     -- ポインタ値 p が指す場所を意味する。
```

アドレスが取れるだけでも Pascal とはだいぶ毛色が違うのだが、おまけにポインタ値に対して足し算と引き算ができる (ただし、その場合ポインタ値は配列のどれかの要素を指している必要がある)。つまり:

```
p + N   -- p が指す要素の N 個先の要素のアドレス。
p - N   -- p が指す要素の N 個前の要素のアドレス。
```

*筑波大学経営システム科学専攻

だから、例えば

```
int a[10];
int *p = &a[6];
*(p+2) = 3;
*(p-1) = 4;
```

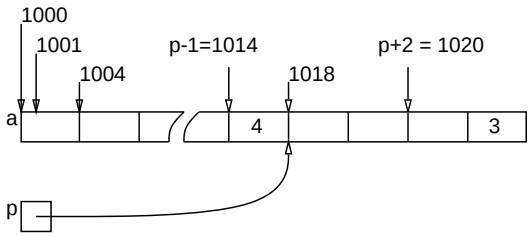

図 1: ポインタ変数とポインタ演算

だと、結果として a の 8 番目に 3、5 番目に 4 を入れていることになる。注意して置きたいのは整数 1 個は 4 バイトを占めるので番地として見た場合には p+2 は p の指す A ドレスの 8 番地先になることである。つまり、ポインタ演算はポインタ値に整数をそのまな足す/引くのでなく、指されている要素の大きさ倍してから足す/引くということになる。

そして次に面白いのは、

```
x[y]
```

という式は実は

```
*(x+y)
```

と全く同じ意味であること、および配列名というのは配列の先頭要素のアドレスを表すポインタ定数にすぎない、ということである。たとえば a が配列名だとすると、`a[5]` というのは`*(a+5)` と同じだから、結果として「配列 a の先頭から 5 要素ぶん先にある場所」を表すことになるだけである。これでようやく、「大きさ 10 の配列へのポインタ」という型がないことが説明できる。つまり、C では配列の場所を渡すにはその先頭要素へのポインタを渡すから、結局「要素へのポインタ型」だけあれば足りてしまうようになっているわけである。

16.3 文字列

文字列というのはなにか数値計算だけでない仕事をやろうと思うと必ず必要になるものである。文字列はその字の通り、「文字が並んだもの」であるから、「文字の配列」で表すのは自然である。例えば Pascal がそういう流儀であるが、Pascal の厳密な型の世界だと「長さ 10 の文字列」と「長さ 11 の文字列」は型が違うことになるので、それがいろいろな厄介のもとになる。

ところが、C では「配列名」は「先頭要素へのポインタ」にすぎないので、文字列も「先頭の文字へのポインタ」として扱う。その型は文字列の長さがどんなであっても「文字へのポインタ型」なわけで、上の問題が起きない。例えば次のような具合である。[1]

```
char *s = "ABCD";  -- s に先頭の文字のアドレスが入る。
... s[2] ...       -- 3 文字目 (0 から数えるのだ!) つまり'C'。
```

ただし、あくまでもこの=はポインタ値の代入だから、次のようにして文字列全体をコピーするというのは許されない。

[1] 申し遅れたが、C では文字定数列は"....."、のようにダブルクォート、文字定数は'X' のようにシングルクォートで囲む。長さ 1 の文字列は文字定数とは別物であるので注意。

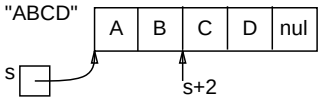

図 2: 文字列の内実

```
char a[10];
a = s;       -- a は定数!代入はできない。
```

また、次のような比較も意味をなさない (なぜか分かります?)。

```
if(s == "ABCD") ...
```

その代わりに、そういう仕事をしたければそのためのサブルーチンを呼び出す。[2] 具体的には文字列の長さを調べたり (`strlen(s)`)、コピーしたり (`strcpy(s1, s2)`)、連結したり `strcat(s1, s2)`)、比較したり (`strcmp(s1, s2)`) する関数などがある。詳しくは「man 3 string」参照のこと。

ところでここで一つ問題なのは、文字列の先頭は分かっても Pascal のように長さが決まっている訳ではないのでどこが最後か分からない、ということである。そこで C の世界では文字列定数は最後に必ず NUL コード (`'\0'` と表記できることになっている) をつけて「終わりの印」にする「習慣」がある。定数に限らず、自分で配列の中に文字列を蓄える場合でもこの「習慣」を守るとなにかと便利であることが多い。実は、前述のサブルーチン群もこれに主としてこれに従った文字列を扱うようになっている。

16.4 構造体

Pascal あたりでもそうだが、配列と並んで重要なデータ構造はレコードである。C ではこれを構造体 (struct) と呼ぶ。その定義と使用方法は:

構造体型 ＝構造体定義 | 構造体参照
構造体定義 ＝struct 名前 宣言 **...**
構造体参照 ＝struct 名前

「名前」は各構造体型を区別する「名札」であり、ある構造体は 1 回定義すれば、あとは同じ名前を指定することで何回でも参照できる。次のような具合である。

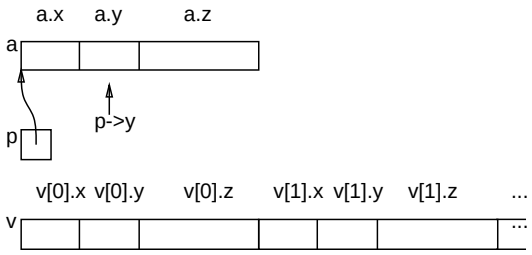

図 3: 構造体、構造体へのポインタ、構造体への配列

```
struct pair { int x, y; double z; } ;   -- これも「宣言」なので; が必要
...
struct pair a;
struct pair *p;
struct pair v[100];
```

[2] 単純な 1 命令で済まないものは言語の基本演算にない、というのが C の方針だという話は前回しましたよね?

構造体の各要素をアクセスするには Pascal と同様「a.x」「v[10].z」のようにピリオドで区切ってフィールド名を指定する。ところで、構造体へのポインタが指している特定のフィールドをアクセスする、という操作は多く出てくるので、そのための専用の記法として「`p->y`」のような書き方ができる。これは「`(*p).y`」と等価であるがより読みやすい。[3]

17　入出力のお話と実験

17.1　チャネルとディスクリプタ番号

さて、今回もそろそろ言語の話ばかりだと飽きるので、プログラムを動かす話に進む。そして今回はちゃんとシステムコールを使うことにしよう。何と言っても OS に頼む仕事のうちで、一番基本なのは入出力であるから、まずはその話である。

ところで、多くのプロセスは特にファイルを指定しない状態では標準入力からデータを読み込み、結果を標準出力に書き出し、そしてエラーメッセージなどは標準エラー出力に出す、という話は前に出てきた。実は Unix ではこれら外部との入出力のための「通路」あるいは「チャネル」を、ディスクリプタ番号小さい番号を使って識別している。具体的にはディスクリプタ番号は、標準入力は 0、標準出力は 1、標準エラー出力は 2 と始めから決まっている。それ以外のファイル等を読み書きするチャネルはまた別の番号を持つことになる。

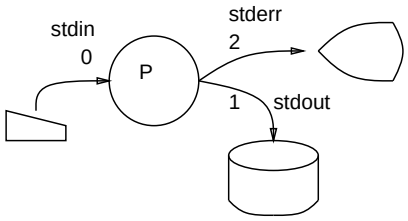

図 4: チャネルとディスクリプタ番号

17.2　write システムコール

そして、ファイル等に出力を行なう `write` システムコールは次のような形で呼び出す:

```
write(番号, バッファ, バイト数);
```

ではさっそく、これを使う例を見ていただこう。

```
/* t04.c -- simple usage of 'write'. */

main() {
  write(1, "This is a pen.\n", 15);
  write(2, "This is a dog.\n", 15);
}
```

これはもちろん、標準出力に`"This is a pen."`、標準エラー出力に`"This is a dog."`と書き出す。

[3]Pascal ではなんでそういう記法が要らないかというと、ポインタ参照演算子↑が後置演算子なので「p ↑.y」のようにかっこ無しで書けるからですね。

```
% cc t04.c
% a.out
This is a pen.
This is a dog.
% a.out >t
This is a dog.
% cat t
This is a pen.
%
```

17.3 read システムコール

さて、次は読む方であるが、そのシステムコールはご想像の通り read という名前で、使い方は次の通り:

バイト数 = read(番号, バッファ, バイト数);

ちなみに、read に渡す「バイト数」というのは「これだけ読みたい」という希望であり、read から返される「バイト数」というのは「実際はこれだけ読めたよ」という値である。[4] これを使って読む例を次に示す。

```
/* t05.c -- simple usage of 'read'. */
char buf[20];

main() {
  int n = read(0, buf, 20);
  write(1, buf, n);
}
```

ところで、読む方が後になったのは、もちろん読んだ結果を書かないと面白くないからである。では実行例:

```
% cc t05.c
% cat t
This is a pen.
% a.out <t
This is a pen.
% cat t1
This is a pen.  That is a dog.
% a.out <t1
This is a pen.  That%
```

最大 20 文字しか読まないので、ファイル t1 ではちょん切れている。

17.4 ごく普通の読み書きループ、そして...

これを解決するには?もちろん、繰り返し読んで書く。

```
% cat t06.c
/* t06.c -- file copy by 'read' and 'write'. */
#define BUFSIZE 20
char buf[BUFSIZE];

main() {
  int n;
  while((n = read(0, buf, BUFSIZE)) > 0)
    write(1, buf, n);
}
```

[4] この両者が一致しないのはどういう場合だと思うか?

ところで、ファイルが終りになったらどうなるのか?実は、`read` はファイルが終りになると「もう読めない」から「0 バイト読めた」と言ってくる。ので、これでいいわけである。では実行例:

```
% cc t06.c
% a.out <t1
This is a pen.  That is a dog.
%
```

確かにできる。もちろん、もっと長いファイルでも動く。ところで、このプログラムはどのくらい「効率がいい」だろうか?試しに cp と比べてみよう。

```
% time cp /usr/dict/words /tmp/t
0.0u 0.1s 0:00 33% 0+29k 4+29io 26pf+0w
% time a.out </usr/dict/words >/tmp/t
0.2u 3.4s 0:03 101% 0+20k 1+28io 0pf+0w
%
```

あれあれ... なぜだと思うか?`/usr/dict/words` は 200,000 バイトくらいの大きさだが、それを上のプログラムでコピーするということは 10,000 回ループを回るわけで、read/write システムコールもその回数だけ呼ばれる。システムコールを 1 回やると 1〜2ms くらい掛かるのが普通だから、上の経過時間約 3 秒というのはうなずけてしまう。もちろん、cp の方はもっとずっと大きなバッファサイズを使うからシステムコールの回数も少ないわけである。では、BUFSIZE は大きければ大きいほど有利か?もし違うとすれば、それはなぜだと思う?

17.5 バッファつき入出力

さて、そういうわけで read や write を 1 回使うにつき 1 回ちゃんとシステムコールが起きていることは分かったし、その問題を避けるにはある程度大きな単位で読み書きするべきだということも分かったわけだが、これでは仕事に使うには大変困る。(だってプログラムではほんの 1 行程度のメッセージを何回も書いたり、入力を 1 行ずつ処理したりするでしょう?)

では、どうするかというと、例えば書く方でいうとどこかにバッファを用意して、「1 文字書け」といわれたら「はいはい、書きますよ」と書くふりだけして、実はバッファにためておき、バッファが満杯になった時 (あるいはもうおしまいの時) ためてあったのをどさっと書けばいいわけである。読む方も同様で、まずどさっとバッファに読んで、あとは「1 文字読んで」と言われたらバッファの中から 1 字取り出して返すようにすればいい。それをみんながそれぞれ書くのはあんまりだから、標準のライブラリとしてこういうバッファつき入出力をやるものが既に用意されている。

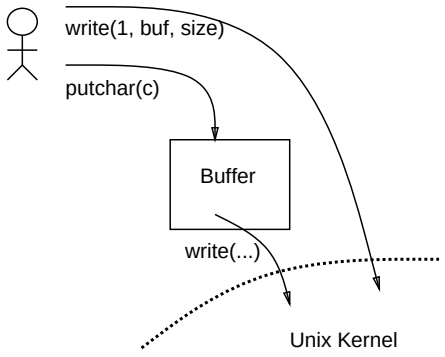

図 5: バッファつき入出力ライブラリ

具体的には、1 文字ずつ読み書きするためには次の関数がある。

```
putchar(c)                  -- stdout に 1 文字書き出す
文字または EOF = getchar()   -- stdin から 1 文字読み込む
```

これを使うと、先のファイルコピーは次のように書ける。

```
/* t07.c -- file copy using buffered i/o. */
#include <stdio.h>

main() {
  int c;
  while((c = getchar()) != EOF)
    putchar(c);
}
```

=というのは代入「演算子」だから式の一部に書ける、というのは前にやった。だから上の while の条件部では getchar を呼んで返ってきた値を c に入れると同時にその値が EOF かどうかを調べるわけである。ここで EOF というのは「ファイルの終り」を示す値であるが、これはどこで定義するのだろう。実は 2 行目は「/usr/include/stdio.h というファイルをここに挿入してから翻訳せよ」という指示であり、そのファイルの中に EOF の値も定義してある。[5] さて、このプログラムはさっきの BUFSIZE=20 のと比べて勝つと思うか、負けると思うか?見てみよう。

```
% cc t07.c
% a.out <t
This is a pen.
% time a.out </usr/dict/words >/tmp/t
0.9u 0.2s 0:01 103% 0+27k 2+27io 0pf+0w
%
```

システムコールの時間はちゃんと減っているが 200,000 回 while ループを回るのでユーザモード時間がそれなりの値になっている。でも全体としてはさっきよりずっと速い!

1 文字ずつ読み書き、というのはそうたびたびやることではないが、このライブラリには他に 1 行書く (ptus) とか、n バイト書く (fwrite) とか色々「便利な」関数が用意してある。中でも一番お世話になるのは次の書式つき入出力であろう:

```
printf("書式文字列", 式,...);
scanf("書式文字列", 場所,...);
```

printf は awk の時もやったし、これからもすぐ出てくるから例は省略。最後に改めて先の基本入出力とバッファつき入出力の違いをまとめておこう。

基本入出力 -- read, write
- ○直接 Unix システムコールに対応
- ○最適な読み方、書き方があるので注意
- ○基本的。

バッファつき入出力 -- getchar, putchar, printf, scanf, ...
- ○ライブラリルーチンで実現
- ○多様な機能、多数の関数が用意されている
- ○間接的に read/write が呼ばれる

17.6 open/close システムコール

さて、ここまではディスクリプタ番号は 0、1、2 だけだったが、実は

```
番号 = open(パス名, 種別 [, モード]);
```

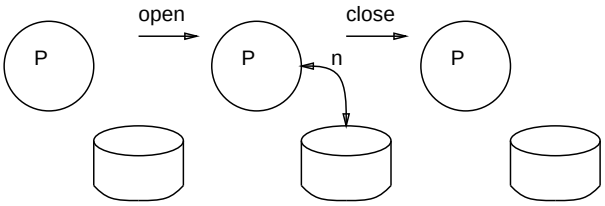

図 6: open と close

によって指定したパス名のファイルにつながる新しいチャネルが作れ、それを識別するディスクリプタ番号 (3, 4, ...) が使えるようになる。ここで「種別」は読むか、書くかなどの指定、モードは新しくファイルが作られる場合にその保護モードを指定する (詳しくは man ページ参照)。また、

```
close(番号);
```

により指定した番号のチャネルを切り離すこともできる。あるプロセスが一時に持てるチャネルの数は上限があるので、使い終ったら close する、というのが行儀良い作法である。
ではごたくはそれくらいにして例へ行こう:

```
/* t08.c -- sample of open/close. */
#include <fcntl.h>
#define BUFSIZE 1024
char buf[BUFSIZE];

main() {
  int n;
  int d0 = open("t", O_RDONLY);
  int d1 = open("t1", O_RDWR|O_CREAT, 0714);
  printf("input desc = %d, output desc = %d\n", d0, d1);
  while((n = read(d0, buf, BUFSIZE)) > 0)
    write(d1, buf, n);
  close(d0); close(d1);
}
```

ちなみに、O_RDONLY 等の「種別」は include に指定されているファイル/usr/include/fcntl.h で定義されている。[6]

```
% cc t08.c
% cat t
This is a pen.
% cat t1
t1: No such file or directory
% a.out
input desc = 3, output desc = 4
% cat t1
This is a pen.
% ls -l t1
-rwx--xr--  1 kuno           15 Dec 15 22:23 t1
%
```

open を使ったからリダイレクトが不要なので、printf が標準出力へ出した文字列は画面に見える。

[5]なぜこういう方法を取っていると思うか?

[6]モードの方は、例の rwxrwxrwx を on の所を 1、そうでない所を 0 とした 2 進数の値で指定する。ときに、実は!C では 0 で始まる定数は 8 進であるので 0714 は rwx--xr--という変てこりんなモードに相当する。普通は 0644 あたりが無難である。

18 システムプログラミングのためのおまけ

18.1 コマンド引数

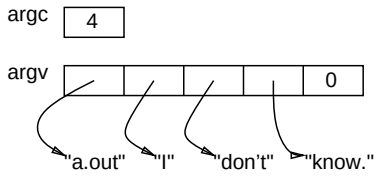

図 7: コマンド引数の渡され方

さて、上のプログラムはファイル名が t と t1 に固定だったが、やっぱり「a.out ファイル 1 ファイル 2」のようにコマンド行から指定できないとらしくないので最後にその説明をしよう。実は、関数 main にはコマンド行の情報が引数として渡されている (が、これまではそれを無視してきた)。その引数の使い方を示す例題を説明しよう:

```
/* t09.c -- sample of command argument usage. */

main(argc, argv)
  int argc;
  char *argv[]; {
  int i;
  printf("argc = %d\n", argc);
  for(i = 0; i < argc; ++i)
    printf("%4d : %s\n", i, argv[i]);
}
```

つまり、main はコマンド引数の数を表す整数、およびコマンド引数の各要素を順に指す配列 (各要素は文字列だから、それを指す型は「文字へのポインタ」であり、従ってその配列は「文字へのポインタの配列」である) を渡される。実行結果であるが:

```
% cc t10.c
% a.out I don\'t know.
argc = 4
   0 : a.out
   1 : I
   2 : don't
   3 : know.
%
```

このように「0 番目」として指令そのものが渡される (シェル変数の$0 を覚えてますか?)。また、などの脱出記号はシェルが指令に渡す前にはがすのでなくなっていることに注意。

で、例えばファイル名をコマンド行から指定したければ適当な argv[i] を open に渡せば良いわけである。

```
/* t10.c -- specifying file names. */
#include <fcntl.h>
#define BUFSIZE 1024
char buf[BUFSIZE];

main(argc, argv)
  int argc;
  char *argv[]; {
  int n, d0 = 0, d1 = 1;
  if(argc > 1) d0 = open(argv[1], O_RDONLY);
  if(argc > 2) d1 = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, 0644);
```

```
  while((n = read(d0, buf, BUFSIZE)) > 0)
    write(d1, buf, n);
}
```

このプログラムではファイル名がないときは stdin を stdout にコピーするが、ファイル名が 1 つ指定されると代わりにそれが入力になり、もう 1 つ指定されるとそれが出力になる。

```
% cc t10.c
% cat t
This is a pen.
% a.out t
This is a pen.
% a.out t t2
% cat t2
This is a pen.
%
```

18.2 エラー処理

さて、先の t10.c には重大な欠陥があるが、それが何だかお分かりだろうか?実は、システムコールが成功しようが何かの原因で失敗しようが構わずつっ走るプログラムなどまともに使うものではない。各システムコールごとに「失敗した時はどういう値になる」と言う情報がマニュアルに明記してあるので、まっとうなプログラマなら必ずこれに従ってエラーチェックを行なうべきである。[7]

で、検出したはよいが、その原因もある程度教えてあげないと、ただ「エラーだ」といわれたユーザは頭にくるに違いない。そのために便利なのが次の関数である:

```
perror("メッセージ");  -- メッセージに続けてエラー内容を表示
```

また、エラーがあったら最低限そこで止まるべきなので、止まり方も:

```
exit(完了コード);  -- プログラムの実行を終り OS に戻る
```

完了コードは 0 が「正常」だから、エラーで止める時は何か 0 以外の値にする。ではこれらを使って「まっとうにした」t10.c を示そう:

```
/* t11.c -- specifying file names. */
#include <fcntl.h>
#define BUFSIZE 1024
char buf[BUFSIZE];

main(argc, argv)
  int argc;
  char *argv[]; {
  int n, d0 = 0, d1 = 1;
  if(argc > 1) d0 = open(argv[1], O_RDONLY);
  if(d0 < 0) { perror("t11: input"); exit(1); }
  if(argc > 2) d1 = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, 0644);
  if(d1 < 0) { perror("t11: output"); exit(1); }
  while((n = read(d0, buf, BUFSIZE)) > 0)
    if(write(d1, buf, n) < 0) { perror("t11: write"); exit(1); }
  if(n < 0) { perror("t11: read"); exit(1); }
  exit(0);
}
```

[7]といいつつ、まっとうでないフレームバッファいじりで皆様に迷惑をお掛けしたことをお詫びいたします。

ちなみに、最後も exit(0) するのが「お行儀よい」。そうしないとごみのリターンコードが返されることがある。それにしても、あんなに簡単だったプログラムがエラー処理を「まっとうに」しただけで何と無惨になることか... やれやれ。でも実行結果はずっとまっとうである。

```
% cc t11.c
% rm t
% a.out t t1
t11: input: No such file or directory
% mv t1 t
% chmod u-w .
% a.out t t1
t11: output: Permission denied
% chmod u+w .
% a.out
%
```

A　演習および課題

例によって前半の 14 節は演習にしようがないので 15 節と 16 節に内容から出すが、わざわざ分ける程でもないので全部 15-x. ということにした。

A.1　15 節の演習

この節の演習もものによっては他人に迷惑が掛かるものがあるから、注意すること。あと、ファイルの読み書きの場合、そのファイルが実行するマシンにくっついたディスクにあるのかネットワーク経由でアクセスされるのかによって違いが出るかもしれないので、一応注意しておく。[8]

15-1. 私が実行例を作成したマシンとあなたが使っているマシンでのシステムコールオーバーヘッドはどのくらい違うか?できれば前回の t01.c などでやった比較をもとに推定してから実測せよ。さらに、この結果をもとに、t06.c の BUFSIZE を 1 にしたらどれくらい掛かるかを予測せよ。

15-2. あなたのマシンでは、最適な BUFSIZE はどれほどか、実際に調べてみよ。できれば cp の速度に勝てるとすばらしい。(きわどい勝負の場合は数回反復実験して平均を取ること!)

15-3. 自分でもバッファつき i/o を作ってみよ。つまり、myputc(c) というのを作ってこれはバッファに 1 文字ため、バッファが一杯なら write で書き出す、というようにするわけである。これと putchar(c) の性能やその他の挙動を色々比較してみよ。

15-4. open を使った例題を修正して「種別」のところを色々変えてみよ。例えば出力の側で O_CREAT や O_TRUNC がないと何が起きるか?また O_APPEND などもうまく動くか試してみよ。

15-5. まずは、echo のそっくりさんを作ってみよ (すごく簡単!)。-n オプションをちゃんとつけること。加えて、できれば次のような機能拡張も行なってみて欲しい。

- 本ものの echo には「-n と打ち出して、しかも改行する」ように指定できないという欠点があるが、これを可能にしてみよ。
- 任意の場所に改行をはさむよう指定できるようにしてみよ。
- 「"abc"が 20 回」のように「繰り返し」が指定できるようにしてみよ。
- 標準出力だけでなく指定した名前のファイルに出力することもできるようにしてみよ。

プログラミングもさることながら、指定のやり方をいかにセンス良くデザインするかが勝負であるので、がんばって欲しい。

A.2　本日の課題

本日の課題は、15-1.〜15-5. のうちから 2 問選択とします。

[8] だからといってどうしようもない、と言われればそれまでだが。