

ソフトウェア工学 (第2回)

土村 展之

(関西学院大学 工学部 教育技術職員)

<http://ist.ksc.kwansei.ac.jp/~tutimura/>

東京農工大学 工学部 情報工学科

2012年7月7日

レポートの講評

- ❖ 前回のレポート課題
- ❖ 期待していた解答
- ❖ もしプログラムを作るなら
- ❖ ここから学ぶべきこと
- ❖ スタイルの注意点 (1)
- ❖ スタイルの注意点 (2)
- ❖ 省略の美德 (?)
- ❖ 命名習慣
- ❖ 命名のコツ (1)
- ❖ 命名のコツ (2)

暦計算

デバッグのコツ

標準入出力

配列の扱い

2次元配列

レポートの講評

前回のレポート課題

1. 30年後の祝日を判定するプログラムは作れない。
その理由を述べよ。
2. 2012年6月30日には第25回めのうるう秒の挿入が予定されている。うるう秒への対応状況をOSや環境、あるいは時計などの機器別に調査し、次の点を考察せよ。
 - 対応する意義、しない場合の問題点は何か。
 - 対応していない環境や機器が多いのはなぜか。

期待していた解答

- 日本の祝日は法律・政令などで決まる。
 - 過去 10 年 20 年の状況を調べる。
(未来を予想するには、過去の状況をふまえる)
 - ❖ 春分の日決定には 1993 年に混乱があった。
 - ❖ みどりの日・昭和の日にはかなりの変遷がある。
 - ❖ 海の日は 1996 年に新設された。
 - ❖ ハッピーマンデー制度により、
2000 年と 2003 年にいくつかの祝日が移動した。
 - カレンダーソフトの中には、祝日をユーザが指定できるものがある。
(競合ソフトを調査する)
- ⇒ 30 年後の法律の制定状況を予想するのは困難。

もしプログラムを作るなら

戦略その1

- 祝日は年ごとにデータとして持つ

戦略その2

- 元旦など、固定した日はルールとして組み込む
(第2月曜などもルール化は可能)
- 春分の日、国民の休日などを年ごとにデータとして持つ

ここから学ぶべきこと

- プログラムが一度ちゃんと動けば、未来永劫ちゃんと動きつづける、などというのは**妄想**。
 - ❖ あてにしていたルールが変わるかもしれない。
 - ❖ いつでも修正できるように、テストプログラムの準備を。
- 未来を予想するには、過去の状況をふまえる。
- 競合ソフトを調査する。

スタイルの注意点 (1)

- 空行、スペース、インデントを適切に。
 - ❖ 変数宣言の後は空行
 - ❖ コンマやセミコロンの後ろはスペース
(for 文の () の中では特に)
 - ❖ #include の直後にはスペース
 - ❖ スペースと改行は等価
 - ❖ 文字列は連結される "A" "B" ⇔ "AB"
- main() をファイルの先頭に書くか、末尾に書くか。
 - ❖ 関数の型宣言を省略したければ main() を末尾に。
 - ❖ 関数同士の依存関係を気にしたくなければ、
型宣言を書いて main() を先頭に。

スタイルの注意点 (2)

- `int hoge();` では型チェックされない。
`int hoge(void);` と書く。
- `{}` のスタイルのおすすめ。

❖ 行末開始

○ <code>if (is_hoge) {</code>	△ <code>if (is_hoge)</code>
<code>printf("A");</code>	<code>{</code>
<code>}</code>	<code>printf("A");</code>
	<code>}</code>

❖ if の後ろで改行するならばブロックを書く

○ <code>if (is_hoge) {</code>	△ <code>if (is_hoge)</code>
<code>printf("A");</code>	<code>printf("A");</code>
<code>}</code>	

省略の美德 (?)

- コンピュータ資源の貧弱な時代の習慣：
 - ❖ 変数名・関数名の長さを短く。(上限6～8文字)
 - ❖ 省略(節約)可能なものは省略(節約)する。
 - △ `long int a;` → ○ `long a;`
 - △ `int main(void) {...}` → ○ `main() {...}`
 - △ `if (a % 2 != 0) ...` → ○ `if (a % 2) ...`
 - △ `i=10; while (--i>=0) puts("10回ループ\n");`
- 現在の豊かな計算機資源・コンパイラ環境では悪影響も。
 - ❖ Javaでは以下はコンパイルエラーになる。

```
int a=10;  if (a % 2) { ... }
```
 - ❖ 初学者にも誤解のないように書くのがトレンド。

命名習慣

- 変数名・関数名

- ❖ (C 言語) 小文字のみ・単語の切れ目にアンダースコア (_)
(例) `is_leap_year`, `day_of_week`, `snake_case`
- ❖ (C++/Java) 単語の切れ目のみ大文字
(例) `isLeapYear`, `dayOfWeek`, `CamelCase`

- 定数

- ❖ (C/C++/Java) 大文字のみ・単語の切れ目にアンダースコア (_)
(例) `TRUE`, `MAX_VALUE`

命名のコツ (1)

- ループ変数は短い名前
- 通用範囲の広い変数ほど長い名前を
- ローカル変数の名前は、時と場合に応じて短くも長くも
- 関数には「動詞＋目的語」(オブジェクト指向の影響)
(例) `add_item()`, `get_size()`
- よくある省略形
(例) `num` (`number`), `len` (`length`), `str` (`string`),
`diff` (`difference`), `buff` (`buffer`)
- おすすめできない省略形: `flg` (`flag`), `tbl` (`table`)
 - ❖ `flag` という変数名にせず、フラグの役割で命名すべき

命名のコツ (2)

- 前置詞を数字に置き換える
(例) hex2str (hex to string), num2date (number to date)
- 対になる単語の例
set↔get, next↔last (previous), pre↔post
- 適切な名前をつけることは、永遠の課題
 - ❖ 「名は体を表す」ように命名する
 - ❖ コメントを沢山書くよりも、適切な名前を
 - ❖ 適切な命名能力は、プログラマにとって重要な資質

レポートの講評

暦計算

- ❖ 身につけたいこと
- ❖ 暦計算
- ❖ グレゴリオ暦の閏年
- ❖ 課題 (20)
- ❖ 関数が複数の値を返すには
- ❖ 課題 (21), (22)
- ❖ 課題 (23)
- ❖ 課題 (24), (24')
- ❖ 課題 (25),(25'), 曜日計算
- ❖ この章の目標を復習
- ❖ ツェラーの公式、ユリウス日

デバッグのコツ

標準入出力

配列の扱い

2次元配列

暦計算

身につけたいこと

- プログラムを作る前に考える
 - ❖ ルールを調査する
 - ❖ 自作すべきか、ライブラリを探すべきか

- プログラムを自作するとなれば
 - ❖ 小さな機能に分割して関数を独立させる
 - ❖ 単体テスト
 - ❖ 値渡し (call by value) と参照渡し (call by reference)

暦計算

- 西暦 y 年 m 月 d 日の前日を求めなさい
 - ❖ 2012年6月23日 → 6月22日
 - ❖ 2012年5月1日 → 4月30日
 - ❖ 2012年4月1日 → 3月31日
 - ❖ 2012年3月1日 → 2月29日 (閏年)
 - ❖ 2011年3月1日 → 2月28日 (平年)
- ルールを調査すべき → 暦は非常に繁雑
- 本来はシステム関数を用いるべき

グレゴリオ暦の閏年

- 閏年には2月が29日までである
- 西暦年が4で割り切れる年は閏年
 - ◆ ただし、西暦年が100で割り切れる年は平年
 - ただし、西暦年が400で割り切れる年は閏年
- 従って400年間に閏年は97回
- 西暦2000年は運が良かった
- 日本では1872年（明治5年）にグレゴリオ暦を採用

課題 (20)

- 閏年かどうかを判定する関数
`int is_leap_year(int year)`
を書きなさい。
- テストプログラムも含めなさい。
 - ❖ 平年 2001, 2002, 2003, 2005, 2100, 2200, ...
 - ❖ 閏年 2000, 2004, 2008, ...
- この関数が完成しないと、後の課題に影響あり。

関数が複数の値を返すには

- 大域変数に書き込む。 (非推奨)
- 複数の関数に分離して、1つずつ値を返す。
- 呼出側で準備した変数をポインタで受け取り、書き換える。
(いわゆる参照渡し)

```
void let_five(int *a) { /* a は int のポインタ */
    *a = 5;             /* *a は int */
}

void foo(void) {
    int i = 1;         /* 1 で初期化 */
    let_five(&i);      /* ポインタを渡す */
    printf("%d\n", i); /* 5 が出力される */
}
```

- 構造体1つを返し、中に複数の値を納める。
(推奨ながらも手間がかかる)

課題 (21), (22)

- (21) ある日付が存在するかどうかを判定する関数
`int is_valid_date(int year, int month, int day)`
を書きなさい。
- (22) 1900年1月1日を基準に、何日めかを計算する関数
`int date_to_number(int year, int month, int day)`
を書きなさい。
(`date_to_number(1900, 1, 1)` を 1 とする。)

課題 (23)

(23) (22) の逆関数

```
void number_to_date(int number,  
                    int *year, int *month, int *day)
```

を書きなさい。

<呼び出し方の例>

```
int main(void) {  
    int year, month, day;  
  
    number_to_date(12345, &year, &month, &day);  
    printf("%d年%d月%d日\n", year, month, day);  
    return 0;  
}
```

課題 (24), (24')

(24) `date_to_number()` と `number_to_date()` を用いて、前日の日付を返す関数 `yesterday()` と、翌日の日付を返す関数 `tomorrow()` を書きなさい。

- ❖ 関数の引数は、ここでは省略したが、必要である。
- ❖ どのような引数をとると都合がよいか、よく考えること。

(24') 2つの日付の間の日数を返す関数

```
int diff_date(int y1, int m1, int d1,  
              int y2, int m2, int d2)
```

を書きなさい。

課題 (25),(25'), 曜日計算

(25) 1900年1月1日が月曜日であることを利用して、
曜日を計算する関数

`int day_of_week(int year, int month, int day)`
を書きなさい。

ただし関数値は0=日曜, 1=月曜, ..., 6=土曜を表すとする。

(25') 以下のような出力をするプログラムを作れ。

```
$ cal 6 2012
      June 2012
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

この章の目標を復習

- 関数分割のありがたみを味わう。
- 不規則なもの（＝日付）は、規則的なもの（＝経過日）に置き換えてから処理する。
- 相互変換の関数が重要。
 - ❖ 対で作ってデバッグにも利用する。
- 関数は「名は体を表す」ように命名する。
 - ❖ 処理内容が少なくても気にしない。

ツェラーの公式、ユリウス日

- ツェラーの公式を用いると、曜日計算は以下のように簡単。

```
int day_of_week(int year, int month, int day) {  
    if (month < 3) { year--; month += 12; }  
    return (year + year / 4 - year / 100 + year / 400  
            + (13 * month + 8) / 5 + day) % 7;  
}
```

- 経過日の計算には、ユリウス日が便利。
- 通常はシステムの `mktime()`, `localtime()` を用いる。
- 複雑なルールには、自分で対処せず、調べることも重要。

レポートの講評

暦計算

デバッグのコツ

- ❖ コンパイラ
- ❖ メモリリーク
チェック
- ❖ 実行回数測定 (プロ
ファイラ)

標準入出力

配列の扱い

2 次元配列

デバッグのコツ

コンパイラ

- 最初のエラーメッセージが重要。
- 警告もなくなるように努力する。
- 警告レベルを上げてみる。

```
gcc -Wall hoge.c
```

```
gcc -Wall -O2 hoge.c
```

```
cl.exe /Wall hoge.c
```

検出される代表的な間違い

- 未初期化変数の利用 → `int a; int b=a;`
- 条件式で代入 → `if (a=b) ...;`
- `printf` のフォーマット間違い → `printf("%d", 5.5);`

メモリリークチェック

- 配列あふれを検出することはC言語では困難。
- 環境を限定すれば、ある程度支援してくれるツールがある。
 - ❖ gdb (UNIX, エミュレーション)
 - ❖ valgrind (UNIX, エミュレーション)
 - ❖ Electric Fence (UNIX, ライブラリの差し替え)
 - ❖ VisualStudio (Windows)
 - ❖ Purify (Windows/UNIX・高価な商品)

実行回数測定 (プロファイラ)

- 関数ごとの実行回数や CPU 時間などの統計情報があると、デバッグや高速化の役に立つ。
- 環境ごとにそれなりのツールがある。
 - ❖ gprof (UNIX, Cygwin)

```
gcc -pg -g -Wall hoge.c  
./a.exe  
gprof a.exe
```
 - ❖ Visual Studio プロファイラ (Windows)
 - ❖ VTune (Windows)

レポートの講評

暦計算

デバッグのコツ

標準入出力

- ❖ 標準入力・標準出力
- ❖ リダイレクト
- ❖ 標準入出力の利点
- ❖ 例題 (30)
- ❖ 標準入出力の限界
- ❖ scanf(), gets() は危険

配列の扱い

2 次元配列

標準入出力

標準入力・標準出力

- コンソールから起動されたプログラムは、起動の時点で3つのファイルをオープンしている。

	通常の接続先	C	C++	Java
標準入力	キーボード	stdin	cin	System.in
標準出力	画面	stdout	cout	System.out
標準エラー出力	画面	stderr	cerr/clog	System.err

- キーボードや画面は、ファイルの一種 (!)
- `printf(...)` は `fprintf(stdout, ...)` と等価

stdin を利用	<code>printf()</code>	<code>getchar()</code>	<code>gets()</code>
stdout を利用	<code>scanf()</code>	<code>putchar()</code>	<code>puts()</code>

リダイレクト

- 標準入出力の接続先は、コマンド起動時に切替えられる。
- 切替えることを「リダイレクトする」という。

標準入力	<code>command < file</code>
標準出力	<code>command > file</code> (上書き) <code>command >> file</code> (付け足し)
標準エラー出力	(sh/bash) <code>command 2> file</code> (csh/tcsh) <code>command >& file</code>

標準入出力の利点

- プログラムの視点から
 - ❖ `fopen()` のエラー処理を書かずに済む。
 - ❖ `stderr` は `fflush()` せずとも即座に表示される。
- ユーザの視点から
 - ❖ 入出力ファイル名をプログラムに束縛されない。
 - ❖ パイプ処理により、他のコマンドとの連携が容易。

行ごとのソート	<code>command</code>		<code>sort</code>
データ圧縮	<code>command</code>		<code>gzip > output.gz</code>
単語数を数える	<code>command</code>		<code>wc</code>
表示しながらファイルに保存	<code>command</code>		<code>tee output</code>

- フィルタ = 標準入出力でデータのやりとりをするプログラム

例題 (30)

- 標準入力から文字列を読み込み、次の出力を下さい。
 - ❖ 標準出力には、大文字に変換したもの
 - ❖ 標準エラー出力には、統計情報
(変換した文字の数、変換していない文字の数など)
- `dir` コマンドの出力を加工下さい。
- 標準エラー出力は “>” でリダイレクトしても、ファイルに落ちないことを確認下さい。
- “>>” でリダイレクトすると追記になることを確認下さい。

標準入出力の限界

以下のいずれでも、自前でファイルオープンする必要がある。

- 入力ファイルが2つ以上
- 出力ファイルが2つ以上
 - ❖ 2つめの出力先としては標準エラー出力は不適當
- 入力ファイルを先頭に戻って同じ内容をもう一度読み込む

scanf(), *gets()* は危険

- `scanf("%d", &i)` ではエラーが起こっても原因がわからない。
 - ❖ 数字以外の文字? 桁あふれ? どこまで読み込んだ?
- `scanf("%s", buff)`, `gets(buff)` では受け取る文字列の長さ制限ができない。
 - ❖ メモリ破壊につながる。
- 以下の処理なら安全

```
int i; char buff[1024];
```

```
fgets(buff, sizeof(buff), stdin);  
sscanf(buff, "%d", &i);
```

レポートの講評

暦計算

デバッグのコツ

標準入出力

配列の扱い

- ❖ 配列 (固定長)
- ❖ 配列 (エラー処理付き)
- ❖ キャストの必要性
- ❖ 配列サイズを超過すると
- ❖ 配列 (可変長)
- ❖ C99 の動的配列
- ❖ 例題 (5') (6)
- ❖ 例題 (31), (31')
- ❖ 例題 (32)

2 次元配列

配列の扱い

配列 (固定長)

```
int hoge1[100];
static int hoge2[100000]; /* 大きな配列ならば */

int *hoge3 = malloc( 100 * sizeof(int) );
int *hoge4 = malloc( 100 * sizeof(hoge4[0]) );
int *hoge5 = malloc( 100 * sizeof(*hoge5) );

int *hoge6 = calloc( 100, sizeof(int) );

if (hoge3 == NULL) {
    printf("Can't allocate memory.\n"); exit(1);
}
hoge3[12] = 4;
free(hoge3);
```

配列 (エラー処理付き)

```
int *hoge7 = xmalloc( 100 * sizeof(hoge[0]) );  
free(hoge7);  
hoge7 = NULL; /* 確実に使えなくする */
```

```
void *xmalloc(size_t size) {  
    void *p;  
  
    if ( size == 0 ) size++;  
    if ( (p=malloc(size)) == NULL ) {  
        fprintf( stderr, "Not enough memory.\n" );  
        exit( EXIT_FAILURE );  
    }  
    return p;  
}
```

キャストの必要性

`(int *)malloc(size)` のようなキャスト（型変換）は、

- 今の C (ANSI C89, C99, C11) には不要。
- C++ と太古の C (K&R) には必要。

- 一般論としてキャストは必要最小限にとどめるべき。
- `malloc()` 周辺には C++ との共通化を優先して無駄にキャストする流儀もある。

配列サイズを超過すると

下のプログラムで何が起こるか試してみなさい。

```
int c[10], d[10];

int main(void) {
    int i, a[10], b[10];

    for (i=0; i<1000; i++) {
        printf("i = %d\n", i);
        a[i] = -1; /* b[i] や c[i] にすると? */
    }
    return 0;
}
```

即座にエラーにならないためにバグが見つかりにくい。

配列 (可変長)

```
int  hoge_num = 0;    /* 使用中の配列サイズ */
int  hoge_max = 10;   /* 確保した配列サイズ */
int  *hoge = xmalloc( sizeof(hoge[0]) * hoge_max );

...
if (hoge_num == hoge_max) { /* 足りなくなったら */
    hoge_max *= 2;         /* 掛算で増やすのがコツ */
    hoge = xrealloc(hoge, sizeof(hoge[0])*hoge_max);
}
hoge[hoge_num++] = 123;
```

C99の動的配列

C99では配列サイズを定数にしなくてもよい。

```
void foo(int size) {  
    int a[size];  
  
    ...  
}
```

例題 (5') (6)

(5') 5つの数値を受け取り、
それぞれ2乗して和を表示しなさい。

(6) ユーザが指定する個数の数値を受け取り、
それぞれ2乗して和を表示しなさい。

- ❖ 1, 10, 100, 1000 のような個数を想定する。
- ❖ ユーザがどのようにして「個数」を指定するのか?
- ❖ 受け取った数値を保持する?しない?
- ❖ scanf() は危険。

例題 (31), (31')

100点満点の成績が40人分ある。この度数分布を調べたい。

(31) 40人の成績を受け取り、1点ごとの人数を表示するプログラムを作れ。

❖ 配列のサイズは40? 100?

(31') 同様に、10点ごとの人数を表示しなさい。

```
100 **
 99 *
 98
 97 *****
...
 10 *
  9 ***
  8 *
```

例題 (32)

- (32) 100 以下の素数は 25 個ある。
これらの素数の 10 の位の数字の度数分布を調べなさい。
- ❖ プログラム 1 本で実現する必要はない。

レポートの講評

暦計算

デバッグのコツ

標準入出力

配列の扱い

2次元配列

- ❖ 2次元配列
- ❖ 2次元配列の関数への引渡し方
- ❖ 2次元配列もどき
- ❖ 2次元配列の改良
- ❖ レポート課題
- ❖ レポート課題（デザインチョイス）
- ❖ レポート課題（目標）
- ❖ レポート課題（予備）

2次元配列

2次元配列

```
int hoge1[10][20]; /* 長方形配列 */
```

```
int **hoge2 = xmalloc(sizeof(hoge2[0]) * 10);  
for (i=0; i<10; i++) {  
    hoge2[i] = xmalloc(sizeof(hoge2[0][0]) * 20);  
}
```

```
hoge1[5][17] = 123;
```

```
hoge2[5][17] = 123;
```

- malloc() は比較的遅い関数なので、ループの中では使いたくない。
- free() もループで回す必要あり。

2次元配列の関数への引渡し方

```
/*長方形配列*/
```

```
void func1a(int a[10][20]) {...}
```

```
void func1b(int a[][20]) {...}
```

```
int hoge1[10][20];
```

```
func1a(hoge1);
```

```
func1b(hoge1);
```

```
/* ポインタへのポインタ */
```

```
void func2(int **a) {...}
```

```
int **hoge2;
```

```
func2(hoge2);
```


2次元配列もどき

```
int *hoge3 = xmalloc( sizeof(hoge3[0]) * 10 * 20 );
```

(1)

hoge3[i][j] の代わりに hoge3[i*20 + j] を用いる

(2)

```
#define HOG3(i,j) hoge3[(i) * 20 + (j)]
```

として HOG3(i,j) を用いる

(関数ではなくマクロとしているのは、代入するため)

2次元配列の改良

```
int **hoge4;
int i, size1, size2;
char *p;

size1 = sizeof(hoge4[0]) * 10;
size2 = sizeof(hoge4[0][0]) * 20;
p = xmalloc(size1 + size2 * 10);

hoge4 = (int**)p; p += size1;
for (i=0; i<10; i++) {
    hoge4[i] = (int*)p; p += size2;
}
```

レポート課題

ビンゴゲームのビンゴ判定をするプログラムを作りなさい。

- 5×5 のマスに 1 ~ 72 の数字が並んでいる。
- 中央のマスは初めから有効になっている。
- 数字がひとつずつ選ばれる。
選ばれた数字が盤面にあれば、そのマスが有効になる。
- 縦横斜めいずれか 1 列が揃って有効になったらビンゴ。
- ビンゴになるまでに選ばれた数字の個数を出力せよ。

<http://ist.ksc.kwansei.ac.jp/~tutimura/tuat/>

で入力データを配布する。ビンゴカードも 1 人 1 枚配布する。

レポート課題（デザインチョイス）

- 盤面のデータはソース中に書き込んでもよい。
外部ファイル（標準入力）から読み込んでもよい。
- 選ばれる数字は標準入力から読み込む。
- 選ばれる数字に重複は許されない。
チェックするか？
- 2次元配列を引数で渡す？グローバルに確保する？
1次元配列で代用する？
- 盤面データを保存する？破壊する？

レポート課題（目標）

- 関数分割を工夫する。
 - ❖ 小さな、まとまった機能に名前をつけて切り出す。
 - ❖ 最初に作るのは状態表示の関数。
- リーチ判定を加える、 5×5 以外への対応など、自由に拡張してよい。
- テストの方法も考えよ。

- 提出期限は7/17(火)。
- 早く提出すれば、評価は上がる。
- 遅れての提出は、提出しないよりは評価する。

レポート課題（予備）

ポーカーの役判定プログラムを作りなさい。

- 5枚のカードの数字のみを用いて、次の役を判定しなさい。
 - ❖ ノーペア（いずれの役もできていない）（例）1,3,5,7,9
 - ❖ ワンペア（同じ数字が1組） 1,1,3,5,7
 - ❖ ツーペア（同じ数字が2組） 1,1,2,2,5
 - ❖ スリーカード（同じ数字が3枚） 1,1,1,5,7
 - ❖ ストレート（5枚が連番） 1,2,3,4,5
 - ❖ フルハウス（ツーペア＋スリーカード） 1,1,1,2,2
 - ❖ フォーカード（同じ数字が4枚） 1,1,1,1,7
- 実際のポーカーではスート（♠♦♥♣）が関係するが、ここでは無視する。