
卒論・修論に役立つプログラミングセミナー

土村 展之 (関西学院大学 理工学部)

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

2019年12月20日/2020年1月8日→10日に延期

はじめに

- ❖ 自己紹介
- ❖ 教室予約システム
- ❖ セミナーの構成
- ❖ プログラミング授業 vs. 卒業研究

講義編

実習編

はじめに

自己紹介

- 関西学院大学 理工学部 教育技術職員 (2008年9月~)
 - ◆ 学科独自 PC 演習室の管理
 - ◆ (ソフト) 理工学部 教室・会議室 使用予約システム
 - ◆ (ソフト) 分散計算によるタンパク質構造解析の高速化
- 「戦略ソフトウェア創造人材養成プログラム」
(2002年~2005年:東京)
 - ◆ 広く使われるソフトを作る人材を育てる
 - ◆ ソフトウェア製作の授業も実施
 - ◆ (ソフト)p T_E Xの文字コード自動判定 (UTF-8)
- 研究テーマ: 組み合わせ最適化・メタ解法・離散凸解析

理工学部 教室・会議室 使用予約システム

2018年度

2019年度

2020年度

前月

<<

前週

<

2019/12/15 (日)-12/21 (土)

>

翌週

>>

翌月

今週

日

週

学期

利用者

管理者

再読込

[初期選択に戻す](#)

予約状況(文字)

予約状況(グラフ)

申請内容の一覧

無効申請検索

CSV出力

予約状況(文字)

凡例：予約された時間帯の色の意味は、次の通りです。

- 緑色：自分の予約
- 茶色：他人の予約
- 黒色：通常授業・事務室利用
- 背景が灰色：貸出対象外

2019/12/17 (火) 19:33 現在

日付	12/15 (日)	12/16 (月)	12/17 (火)	12/18 (水)	12/19 (木)	12/20 (金)
事務室開室時刻	閉室	8:50-16:50	8:50-16:50	8:50-16:50 当日予約は 事務室の窓口で	8:50-16:50	8:50-16:50
4号館201 (定員98人)		11:10-12:40 16:50-18:20	11:10-12:40 13:30-15:00 15:10-16:40	11:10-12:40 15:10-16:40	13:30-15:00 予約する	11:10-12:40 予約する
4号館202 (定員98人)		9:00-13:00 13:30-15:00 16:50-18:20	11:10-12:40 [重複] 11:10-12:40 [重複] 15:10-16:40	11:10-12:40 13:30-15:00	11:10-12:40 13:30-15:00 16:50-18:20 予約する	11:10-12:40 13:30-15:00 予約する

セミナーの構成

- 想定する受講対象者
 - ◆ 関学理工学部 B3 以上
(テキスト差分 (diff) を知っている人)
 - ◆ 何かの言語でプログラムが書ける人
 - ◆ 卒業研究でプログラムを作る可能性のある人
- 講義 50 分 + 演習 30 分
- 講義では OS や言語に依存しないように配慮
- 演習は C, Java, Python など

プログラミング授業 vs. 卒業研究

- プログラミング授業
 - ◆ 作るべきプログラムを明確に指示される
 - ◆ クラス中で同じ作業
 - ◆ 答えがある
- 卒業研究
 - ◆ やりたい作業は（ぼんやりと）決まっている
 - ◆ 作業を実現するためにプログラムを作ってもよい
 - ◆ 答え合わせできない
 - ◆ 目標や前提が途中で変わるかも

はじめに

講義編

- ❖ ソフトウェアは残すべし
- ❖ 再利用を考慮したソフト開発-1
- ❖ 再利用を考慮したソフト開発-2
- ❖ コピー&ペーストがいけない理由
- ❖ 身につけるべき知識
- ❖ バージョンの習慣
- ❖ 大規模ソフトのバージョン
- ❖ ドキュメンテーション技術
- ❖ コメント最小限のソースコード
- ❖ プログラム例
- ❖ プログラムの構成 – 関数分割の方針
- ❖ プログラム例
- ❖ ソースコードの読み合いの勧め
- ❖ 開発ツールの利用
- ❖ プロファイラの簡単な使い方

❖ プロファイラの簡単な使い方
卒業論文に役立つプログラミングセミナー

❖ 何を指すか

講義編

ソフトウェアは残すべし

- 研究室の後の人のために
- 「明日の自分 = 他人」のために
- 再利用による効率化・信頼性確保
- 全体を再現 vs. パーツとして切り出す
- 実行形式の再利用 vs. ソースコードの再利用

再利用を考慮したソフト開発-1

- 他人のプログラムを読んでみる
(研究室内で読み合い)
- ソースコードの一元管理 (最新版を維持・公開)
- ソースコードのバックアップ (バージョン管理)
不具合が出たら、過去のソースを比較して追跡調査

再利用を考慮したソフト開発-2

- パーツの単体テスト
 - コピー&ペーストは保守性の低下を招く
Cソースなら月に3000行（コメント除く）書ければ十分
 - 適切なドキュメンテーション
 - Makefile, ant などを利用したコンパイルの自動化
 - バージョン管理ツール（git, Subversion 等）の利用
- 「効率の良いソフトウェア開発」とほぼ同義

コピー & ペーストがいけない理由

- 処理内容の見通しが悪い。(例外処理が埋もれてしまう。)
- データの差し替えが困難になる。
- アルゴリズムの差し替えが困難になる。
- コピーせずに、それをプログラムで表現できるはず。
- 実はこのプログラムは無駄の塊！

```
int code_num(char *code)
{
    if (strcmp(code, "ABS")==0) return 0;
    if (strcmp(code, "ADE")==0) return 1;
    if (strcmp(code, "ATY")==0) return 2;
    /* ... 205 行省略 */
    if (strcmp(code, "ZCT")==0) return 208;
    return -1;
}
```

身につけるべき知識

- バージョン番号
- ドキュメンテーション（説明文とコメントの書き方）
- プログラムの構成
- 開発環境
- 大きな問題を小さな簡単な問題に分割

バージョンの習慣

- もうバージョンアップしないと思ったプログラムは、必ずバージョンアップするはめになる
- 番号を付けて世代管理（小規模なら 日付で十分）
- バージョン違いのプログラムを手にした人に新旧の区別がつくよう
- 通常 Ver 1.20、小改定で小数を++, 大改定で整数を ++, 初期バージョンは 0.01 や 1.00
- ほかにリビジョン/build/beta/alpha など、多くの（勝手な）習慣

大規模ソフトのバージョン

- 西暦 (2018), 年+月 (1804), 連番 (20, 21, 22, ...)
- メジャーバージョンアップ直後は不具合多い
マイナーバージョンアップを待て
- 「安定 \neq 時代遅れ」 先進・高性能とは両立しない
- 数回メジャーバージョンアップの後の完成度が高い
(Ver 3.1 が使い物になる? !)
- 近頃は最新版が強要される
- セールストークのためのバージョンアップも

ドキュメンテーション技術

- ソースコードのコメント
- ソースコード以外の文章
オブジェクト指向言語ならクラス図
- ドキュメント生成ツール JavaDoc, Cxref, etc...
関数呼びだし回数一覧などは機械生成可能
- コメント必要論 vs 不要論
間違ったコメントならないほうがマシ
- コードから即座には読み取れない情報を書き留める

コメント最小限のソースコード

- 意味のわかる変数名・関数名
(命名規則 — 大文字/小文字の使い分けの習慣、is...)
- 1つの関数は短く
(30行を目標、長くても100行、局所変数は10個まで、インデントは5段まで)
- マジックナンバーの排除
- 「難解なコード+コメント」より「自然なコード」
- 関数・大域変数・エラー発生時の返り値・不自然な処理には説明を
- 嘘（修正忘れ）なら、書かないほうがマシ

プログラム例

```
#define SHRINK_FACTOR 1.3

/* a[] を昇順に整列 */
void combsort(int a[], int size) {
    int i, j, is_swapped, gap = size;

    do {
        gap = (int)(gap / SHRINK_FACTOR);
        if (gap == 0) gap = 1;           /* バブルソートと同じになる */
        if (gap == 9 || gap == 10) gap = 11; /* Combsort11 を実現 */
        is_swapped = FALSE;
        for (i = 0, j = gap; j < size; i++, j++) {
            if (a[i] > a[j]) {
                int swap = a[i]; a[i] = a[j]; a[j] = swap;
                is_swapped = TRUE;
            }
        }
    } while (gap > 1 || is_swapped);
}
```

プログラムの構成 – 関数分割の方針

- 独立した機能
 - 処理が似ている部分を切り出すのではない
 - 今は中身が同じ処理であっても、本質の違うものは別関数に
- 簡潔なインタフェース
- 機能が連想できる名前
 - 名前をつけて具体的な処理を忘れよう
- サイズは小さく

普通の間人は一度に多くのことを考えられません。長いコードはそれだけで読む気が萎えます。一般に 50 行が限度のようですが、個人的には 20 でも良いと思っています。短いほど機能を絞り込めます。極端な話 1 行でも独立した機能ならそれはそれで問題ありません。

プログラム例

```
short fget_int16(FILE *fp) {  
    short a; fread(&a, sizeof(a), 1, fp); return a;  
}
```

```
int fget_int32(FILE *fp) {  
    int a; fread(&a, sizeof(a), 1, fp); return a;  
}
```

```
long fget_int64(FILE *fp) {  
    long a; fread(&a, sizeof(a), 1, fp); return a;  
}
```

```
double fget_double(FILE *fp) {  
    double a; fread(&a, sizeof(a), 1, fp); return a;  
}
```

```
double fget_float(FILE *fp) {  
    float a; fread(&a, sizeof(a), 1, fp); return a;  
}
```

ソースコードの読み合いの勧め

- プログラムを書いた本人が解説する
- 説明を受ける人が疑問点や改良点を指摘
- 他人のよい習慣を身につける場に
- バグが発見されることも

開発ツールの利用

- プロファイラ（関数ごとの実行時間/回数測定）
通例、実行時間の半分以上はプログラムの4%未満の部分に費やされる。
D.E.Knuth(1971)
- 検査コードの挿入 `#include <assert.h>`
- 文法チェックツール（コンパイラの警告, lint)
- デバッガ（ステップ実行)
- メモリリーク検査ツール
valgrind, Electric Fence, Dmalloc, etc..

プロファイラの簡単な使い方

gcc, gprof の場合

- コンパイル・リンク時のオプションに `-pg` を加える
 - gcc は `-O?` オプションと共存可能
 - `% gcc hogehoge.c -O2 -Wall -g -pg`
- 統計情報採取のためテスト実行（速度低下している）
 - `% ./a.out`
- "gmon.out" というファイルができていることを確認
 - 正常終了しないとできない
- プロファイラを実行
 - `% gprof ./a.out | less`

プロファイラの簡単な使い方

java の場合

- 実行時に `-Xprof` オプションをつける
`java -Xprof Hoge`
- 終了時に統計情報が出力される

Flat profile of 4.48 secs (418 total ticks): main

Interpreted + native				Method
91.1%	381	+	0	TSP.main
0.5%	2	+	0	TSP.readDistance
0.5%	2	+	0	TSP.orOpt
0.2%	1	+	0	java.lang.AbstractStringBuilder.<init>
0.2%	1	+	0	TSP.initialTour
0.2%	1	+	0	TSP.twoOpt
92.8%	388	+	0	Total interpreted

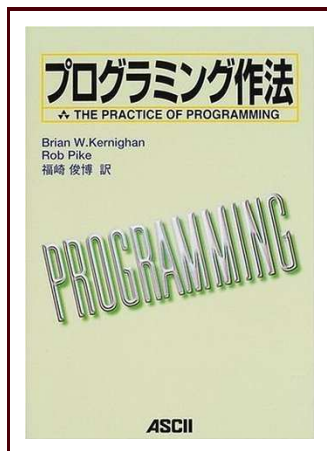
何を目指すか

- コードの読み合いを通してよい習慣を身につける
- コンピュータではなく、人間に読みやすく
- 独立した機能を抜き出して関数・モジュールに分離
- 意味のわかる変数名・関数名
- 冗長な記述（コピー＆ペーストや過剰なコメント）を排除
- 確かなアルゴリズムに裏打ちされた素直な処理

というのは理想で、
動かないよりかは、動く汚ないプログラム

参考文献

- [1] 「プログラミング作法」 ISBN4-7561-3649-4
Brian W.Kernighan, Rob Pike 著, 福崎敏博 訳, アスキー
- [2] 「ソフトウェア再利用の神話—ソフトウェア再利用の制度化に向けて」
ISBN4-894714256
Will Tracz 著, 畑崎 隆雄・林 雅弘・鈴木 博之 訳, ピアソン・エデュケー
ション
- [3] 「UNIX プログラミングツール」 ISBN4-88135-799-9
Eric Foster-Johnson 著 たかのゆきまさ~ 監訳 翔泳社



参考文献

[4] 「Cプログラミング診断室」 ISBN4-87408-571-7

藤原博文著, 技術評論社

<http://www.pro.or.jp/~fuji/mybooks/cdiag/>

[5] 「Cプログラミング専門課程」 ISBN4-7741-0090-0

藤原博文 著, 技術評論社

<http://www.pro.or.jp/~fuji/mybooks/cpro/>

[6] 「バージョン表記のフシギ」

<http://www.forest.impress.co.jp/article/2001/07/16/yomoyama19.html>



はじめに

講義編

実習編

- ❖ 例題 (1), (2)
- ❖ 解答例 (1)
- ❖ 解答例 (2) その 1 (C)
- ❖ 解答例 (2) その 1 (Java)
- ❖ 解答例 (2) その 1 (Python)
- ❖ 解答例 (2) その 2 (C)
- ❖ 解答例 (2) その 2 (Python)
- ❖ 例題 (3)
- ❖ 解答例 (3) (C)
- ❖ 解答例 (3) (Python)
- ❖ 例題 (4)
- ❖ 解答例 – 独立した関数に
- ❖ 解答例 – コマンドラインオプション
- ❖ 解答例 – 構成変更
- ❖ 解答例 – 一応完成
- ❖ 解答例 – Mersenne Twister を使う
- ❖ Mersenne Twister を

実習編

例題 (1), (2)

(1) “Hello, world!” を表示しなさい。

- ◆ ウォーミングアップ。

(2) ユーザが指定した回数だけ “Hello, world!” を表示しなさい。

- ◆ ユーザがどのようにして回数を指定するのか？
- ◆ 実現方法は 1 通りではない。
- ◆ 回数は int に格納する？ double に格納する？

解答例(1)

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

```
int main(void) {  
    printf("Hello, world!\n");  
    return 0;  
}
```

```
print("Hello, world!")
```

解答例(2) その1 (C)

- 回数を コマンドライン引数 から受け取る

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

int main(int argc, char *argv[]) {
    int num = atoi(argv[1]);
    for (int i=0; i<num; i++) {
        printf("Hello, world!\n");
    }
    return EXIT_SUCCESS;
}
```

main() 関数の引数 argc, argv が重要。
本来は引数なしの場合の処理も必要。

解答例(2) その1 (Java)

- 回数を コマンドライン引数 から受け取る

```
public class S02a {  
    public static void main(String[] args) {  
        int num = Integer.parseInt(args[0]);  
        for (int i=0; i<num; i++) {  
            System.out.println("Hello, world!");  
        }  
    }  
}
```

main() 関数の引数 args が重要。
本来は引数なしの場合の処理も必要。

解答例(2) その1 (Python)

- 回数を コマンドライン引数 から受け取る

```
import sys
num = int(sys.argv[1])
for i in range(num):
    print("Hello, world!")
```

sys.argv が重要。
本来は引数なしの場合の処理も必要。

解答例(2) その2 (C)

- 回数を 標準入力 から受け取る

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

int main(int argc, char *argv[]) {
    int num;

    scanf("%d", &num);
    for (int i=0; i<num; i++) {
        printf("Hello, world!\n");
    }
    return EXIT_SUCCESS;
}
```

本来はエラー処理やプロンプト表示も必要。

解答例(2) その2 (Python)

- 回数を 標準入力 から受け取る

```
import sys
num = int(input())
for i in range(num):
    print("Hello, world!")
```

本来はエラー処理やプロンプト表示も必要。

例題 (3)

問題 ファイルの中に数字のキャラクタ (0~9) がいくつ現れるかを数えるプログラムを作れ。

前提 入力ファイルには ASCII 文字のみが含まれているとする。使用言語は問わない。

ポイント

- 標準入出力が使えるか (フィルタ)
- C で書くなら `isdigit()` を正しく使えるか

解答例 (3) (C)

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

int main() {
    int c, count = 0;

    while ( (c = getchar()) != EOF ) {
        if ( isdigit(c) ) count++;
    }
    printf( "%d\n", count );
}
```

isdigit(c) は比較しないのがポイント

解答例 (3) (Python)

```
import sys
import re

count = 0
for line in sys.stdin:
    count += len(re.findall('\d', line))
print count
```

正規表現を利用した

例題 (4)

問題 サイコロを振って8回連続して偶数の目が出るかどうか、2560000回試行するプログラムを作れ。

ポイント

- 8 や 2560000 は magic number?
- `srand()` は最初に一回のみ
- `srand(time(NULL))` で再現できない乱数系列を
- `atoi(argv[1])` で種を受け取るとよい
- `argc` を見て `time(NULL)` と `atoi(argv[1])` を振り分ける
- `rand()` はコンパイラによって違うので Mersenne Twister を使おう
- [1..6] の整数乱数の作り方

解答例 – 独立した関数に

```
int rand2(void) {  
    return rand() % 2; // 乱数の使い方の悪い例  
}
```

```
int is_continuous_even(int n) {  
    for (int i=0; i<n; i++) {  
        if (rand2() == 0) return FALSE;  
    }  
    return TRUE;  
}
```

解答例 – コマンドラインオプション

```
int main( int argc, char *argv[] ) {
    int seed;

    if ( argc > 1 ) {
        seed = atoi(argv[1]); // コマンドライン引数
    } else {
        seed = time(NULL); // 引数なしなら時刻を使う
    }
    srand(seed); // 乱数の種は 1 度だけ設定すれば十分

    .....
    return EXIT_SUCCESS;
}
```


解答例 – 構成変更

```
int tries; // 試行回数
int seed;  // 乱数の種

void get_arg( int argc, char *argv[] ) {
    if ( argc == 1 ) {
        printf("usage: %s tries [seed]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    tries = atoi(argv[1]);

    if ( argc > 2 ) {
        seed = atoi(argv[2]);
    } else {
        seed = time(NULL);
    }
    srand(seed);
}
```

解答例 – 一応完成

```
#define CONTINUOUS 8

int main( int argc, char *argv[] ) {
    get_arg( argc, argv );

    int success = 0;
    for (int i=0; i<tries; i++) {
        if ( is_continuous_even(CONTINUOUS) ) success++;
    }

    printf("seed = %d, %d/%d\n", seed, success, tries);
    return EXIT_SUCCESS;
}
```

解答例 – Mersenne Twister を使う

```
#include "mt19937ar.h"

/* [0,1] の整数乱数 */
int rand2(void) {
    // return rand() % 2; // コンパイラ標準
    return genrand_int32() % 2; // mt19937ar
}

/* [0,5] の整数乱数 */
int rand6(void) {
    return (int)(genrand_real2() * 6);
    /* 半開区間バージョン [0,1) の乱数を6倍して切り捨て */
    /* genrand_int32() % 6 よりもよい乱数のはず */
}
```

Mersenne Twister を使うメリット

- コンパイラに依存しない乱数系列が得られる
- 各種言語の実装が公開されている
C, C++, Java, Python, MS-EXCEL, ...
- C 版は商用利用も可能（変更済み BSD ライセンス）
- よい疑似乱数（GFSR 法の改良）
 - ◆ きわめて長い周期 $2^{19937} - 1$
 - ◆ 623 次元以下で一様に分布
- 十分に高速（線形合同法と同等）