Title	計算機プログラミングI・同演習 講義ノート2007
Author(s)	井上, 純一
Description	2007年度前期に開講された工学部情報エレクトロニクス学科2年生を対象としたLinuxシステム、C言語プログラミングに関する入門的な講義・演習の講義ノートです。この講義・演習で扱わない、より進んだ内容は後期に開講される「計算機プログラミングII」にて学習します。
Issue Date	2007-08-22T04:23:05Z
Doc URL	https://hdl.handle.net/2115/28047
Rights(URL)	https://creativecommons.org/licenses/by-nc-sa/2.1/jp/
Type	learning object
File Information	ProgI2007_5.pdf, 第5回講義・演習ノート

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

計算機プログラミングI 講義ノート #5

担当：井上 純一 (情報科学研究科棟 8-13), 赤間 清 (情報基盤センター)

URL：http://chaosweb.complex.eng.hokudai.ac.jp/~j_inoue/PROG2007/PROG2007.html

平成 19 年 5 月 11 日

目次

13.3 do-while 文 37

14 プログラムを書く際の注意点 39

14.1 読みやすいプログラム 39

14.1.1 段付け (indent) 39

14.1.2 簡潔な記述 40

14.1.3 コメントの活用, 変数名の工夫 41

14.2 修正しやすいプログラム 41

14.3 効率の良いプログラム 42

14.3.1 記憶効率 42

14.3.2 実行効率 42

14.3.3 実行時エラーの少ないプログラム 43

15 数値計算 I 43

15.1 素朴な 2 分法で解く 44

15.2 ニュートン法 45

13.3 do-while 文

繰り返し制御の最後に **do-while 文**を学びます．do-while 文の一般形は次のようになっています．

```
do
{
        文 1;
        文 2;
        ............
        ............
        ......
        文 n;
}while(条件式);
```

そして, 具体的には次のように実行されることになります.

(1) **ループ本体の実行：** ループ本体を実行し, 次のステップへ移る.
(2) **繰り返しの判定：** while 文の「条件式」を評価し, 0(偽) であれば do-while 文全体の実行を終了し, 0 でなければ (1) へもどる.

これも次の練習問題にあたることで実際に動作を確かめてみましょう.

練習問題 13.3

次のプログラムをファイルに書き込み, コンパイル・実行することによって動作を確かめよ.

```
/* do-while 文の動作を確認するためのプログラム */
/* 最大公約数を求める */
#include<stdio.h>
#include<math.h>

main()
{
        int a,b,m,n,r;
        scanf("%d %d",&a,&b);
        m=a;
        n=b;

        do{
             r=m%n; /* m を n で割った余りを求め, その値を変数 r に代入 */
             m=n; /* n の値を m に代入 */
             n=r; /* r の値を n に代入 */
        }while(r!=0);

        printf("G.C.D. of %d and %d is %d\n",a,b,m);
}
```

※注意事項：

(1) 条件式の値がはじめから 0(偽) であっても, **ループ本体の文は少なくとも 1 回は実行される**.
(2) while 文, for 文はループ本体の実行前に条件式の評価を行うが, do-while 文は**最後**に行う. 従って, 最低でも 1 回はループ本体を実行する必要のある場合には do-while 文を用いるべきである.
(3) 次のような記述は無限ループを生成する. 注意されたし.

```
/* 無限ループ */
do
 {
        printf("beautilful life !\n");
 }while(1);
}
```

14　プログラムを書く際の注意点

ここからはC言語プログラミングの本題をやや外れて, 良いプログラムとはどんなものか, ということを少し考えてみたいと思います. まず, 良いプログラムかどうかというのは様々な観点から評価されるべきであり, 例えば次のような見方があげられます.

- 読みやすいプログラム
- (自分もしくは他人が) 修正しやすいプログラム
- 効率の良いプログラム
- 実行時のエラーが少ないプログラム

これらのそれぞれを以下で詳しくみて行くことにしましょう.

14.1　読みやすいプログラム

プログラムとしての「読みやすさ」とは何を指すのかを見ていきます.

14.1.1　段付け (indent)

次の例に見るように適切な**段付け (indent)** をつけるとプログラムが格段に読みやすくなります.

```
/* 段付けの例 */
for(i=0;i<=9; i=i+1){
       n = n + 1;
       printf("%d\n",n);
 }
```

しかし, 次のようにしてしまうと段付け自体が本質的な間違いを犯してしまいますので, 段付けにも細心の注意が必要です.

```
/* 段付けが問題を引き起こす例 */
#include<stdio.h>
#include<math.h>
main()
{
        int t;
        t=0;
        while(t < 10)
        printf("t=%d\n",t);
        t = t + 2;
}
```

練習問題 14.1 (※ LMS 入力問題)
上記の「段付けが問題を引き起こす例」にあげたプログラムは無限ループを生成してしまう. このプログラムを修正し, 無限ループを回避せよ.
※ 万が一, 無限ループが生じてジョブが止まらなくなった場合には, Ctrl-C で強制終了させること.

14.1.2 簡潔な記述

C 言語には**特殊代入演算子**や**インクリメント演算子**, **デクリメント演算子**があり, これらを用いることで簡潔な記述が可能になります. まず, 以下に特殊代入演算子について表にまとめておきましょう.

特殊演算子	説明	意味
$+=$	加算する	$x = x + \mathrm{value}$
$-=$	減算する	$x = x - \mathrm{value}$
$*=$	乗算する	$x = x * \mathrm{value}$
$/=$	除算する	$x = x/\mathrm{value}$

また, インクリメント, デクリメントに関しては以下の通りです.

○ インクリメント：
意味：整数変数に 1 を加えることを意味する.

表現例 1：count + +;
区別：count の値を使用した後にインクリメントを行う.
表現例 2：+ + count;
区別：count の値を使用する前にインクリメントを行う.

○ デクリメント：
意味：整数変数から 1 を引くことを意味する.

表現例 1：count − −;
区別：count の値を使用した後にデクリメントを行う.
表現例 2：− − count;
区別：count の値を使用する前にデクリメントを行う.

次の練習問題で確認しておきましょう.

練習問題 14.2
インクリメント演算子の使用法を確認するために, 各自が次の 2 つのプログラムを書き, コンパイル・実行することによって出力結果を比較せよ.

```
/* インクリメント (区別 1) */
#include<stdio.h>
#include<math.h>
main()
{
        int x=6,y;
        y=x++;
        printf("%d\n",y);
}
```

```
/* インクリメント（区別 2）*/
#include<stdio.h>
#include<math.h>
main()
{
        int x=6,y;
        y=++x;
        printf("%d\n",y);
}
```

14.1.3 コメントの活用, 変数名の工夫

コメントに関しては既に述べてますのでここでは繰り返しません. 変数名に関してですが, 基本的にプログラムの中に現れる全ての変数は何らかの意味を持っているはずです. 例えば, 既に見てきましたが, 2 次方程式の解, 三角形の辺などを変数を使って表す場合に, x,a,b,c などを用いました. 簡単なプログラムで変数の数も少ない場合にはプログラムを読めば大体の変数の意味までわかってしまいますが, プログラムが長くなり, 変数の及ぶ範囲も広い場合, その変数の名前を見ただけで何を表しているのか, がある程度わかるものが望まれることになります. 例えば, 2 次方程式の解の場合には kai1,kai2 あるいは, solution1, solution2 を用いたり, 辺の場合には side1,side2,side3 などです. また, これ以外にループなどをまわす際の変数は通常, アルファベットの小文字の一文字を使う場合が多いようです. つまり

```
for(i=0; i < N; i++){
      ........... ;
}
```

として i あるいは j を用いるように. ただし, 小文字の l (エル) は数字の 1 と間違いやすいので, l は単独で用いずに何か別のアルファベットと一緒に組み合わせたりして, 数字の 1 と明確に区別できるようにすると良いかも知れません.

14.2 修正しやすいプログラム

プログラムは未知の修正に対しても, 可能な限り修正ができるように作成することが必要です. 例えば, 次のプログラムでは **define** を用いて変数の値を変更しています.

```
/* #define を用いた変数の変更 */
/* まずは #define を使わない場合で書いてみましょう */
#include<stdio.h>
#include<math.h>
main()
{
        int i;
        for(i=1; i <= 100; i++){
                n = n + i;
               printf("%d\n",n);
```

```
        }
}
/* 続いて #define でループの上限を与える書き方 */
#include<stdio.h>
#include<math.h>
#define N 100
main()
{
        int i;
        for(i=1; i <=N; i++){
               n = n + i;
               printf("%d\n",n);
        }
}
```

14.3 効率の良いプログラム

一口に「効率」と言っても様々な観点があります.

14.3.1 記憶効率

言うまでもなく, 記憶効率を高めるにはあまりメモリを使わないことですが, 例えば用いる変数にしても次の表のような違いがあります.

変数型	必要バイト数	表現範囲
int	4 (UNIX 高速)	$-2^{31} \sim 2^{31}-1$
short in	2 (UNIX 低速)	$-2^{15} \sim 2^{15}-1$
float	4 (UNIX 低速)	$\pm 3.4 \times 10^{-38} \sim \pm 3.4 \times 10^{38}$
double	8 (UNIX 高速)	$\pm 1.7 \times 10^{-308} \sim \pm 1.4 \times 10^{308}$
long double	10 (UNIX 低速)	$\pm 3.4 \times 10^{-4932} \sim \pm 3.4 \times 10^{4932}$

一般的に言って, メモリを低く抑えると表現精度は落ちることになります.

14.3.2 実行効率

変数の選択では高速演算のできる変数型を選択することが重要になります (上の表を参照). UNIX(LINUX) のプログラムを書く際, 大量メモリが必要な場合には整数型は short int, 浮動小数は float を用いるけれども, 一般的には精度と演算効率とを考慮に入れて int と double を用いることが適切です. また, 整数演算と実数演算は使用される回路が異なるので演算速度も異なります.

また, これ以外にも計算アルゴリズムを工夫することで実行効率を高めることもできます. 例えば, 変数 y の値に x の多項式を代入するような場合, 具体的には

$$y = 3x^3 + 2x^2 - 4x + 0.5 \tag{2}$$

を C 言語で書くとなると

```
y=3*x*x*x+2*x*x-4*x+0.5 /* (式 1) */
```

となりますが, これは $y = \{(3x+2)x-4\}x+0.5$ のように式変形できますから, 次のように書くこともできます.

```
y=((3*x+2)*x-4)*x+0.5 /* (式2) */
```

この (式 1) と（式 2）の乗算回数と加算回数を勘定してみると次のようになります.

	乗算回数	**加減算回数**	**計**
式 1	6	3	9
式 2	3	3	6

従って, どちらとも加算回数は同じだけれども, (式 2) のように式変形を施してから計算機にジョブを渡した方が乗算回数が少なくて済み, 従って実行効率も上がるというわけです. このように, 計算回数を考えて効率の良い方の演算順序を選ぶことでも実行効率は向上します. 前に述べたように, 一般的に言って, 記憶効率と実行効率はトレードオフのようなところがありますが, この演算順序の交換や式変形で計算回数を減らす工夫は記憶効率にほとんど影響しませんから, とりわけ大容量のメモリを使う計算機シミュレーションなどではとても有効になってきます.

14.3.3 実行時エラーの少ないプログラム

実行時にエラーが生じると, 多くの場合, 「エラー・コード」と呼ばれる記号と簡単なエラー・メッセージが出力され, エラーの発生したソースプログラムの行数は表示されません. 従って, コンパイル時のエラーと比べ, その原因を探るのが格段と厄介になりますから, この実行時エラーを少なくするようなプログラムの書き方をすることが望まれます.

それでもやはり実行時にエラーが出る場合にはどうするかですが, 例えば, 変数の中にとても大きな数が代入されてしまったり, あるいはゼロで割ってしまうことに気づかないでプログラムを実行してしまう場合があります. その際には printf 文を用いてその変数の値を逐次表示させ, 途中で変数に何かおかしな値を代入していないかどうかをチェックすることはできます. この手のデバックの方法に関しては回を改めて後日見ていくことにしましょう.

15 数値計算 I

ここまで学んだ事柄を用いるとある程度までのプログラミングはできるようになっていると思いますので, ここでは実際の科学技術計算で用いる数値計算上の技法のいくつかを学んでいくことにします. その中でも, ここでは

$$f(x) = 0 \tag{3}$$

のタイプの代数方程式を数値的に解くにはどうするのかということを考えてみたいと思います.

そのような代数方程式として, 例えば $f(x) = x^2 - a^2$ の場合には, $x = \pm\sqrt{a}$ のようにその解が手で求まってしまいますが, 特殊なケースとして因数分解ができない限り, 多項式 : $f(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ に関する $f(x) = 0$ の解は求まりません. こういう場合に解の候補 (近似解) を探すにはどうするのかを, ここでは具体的に見て行こうというわけです.

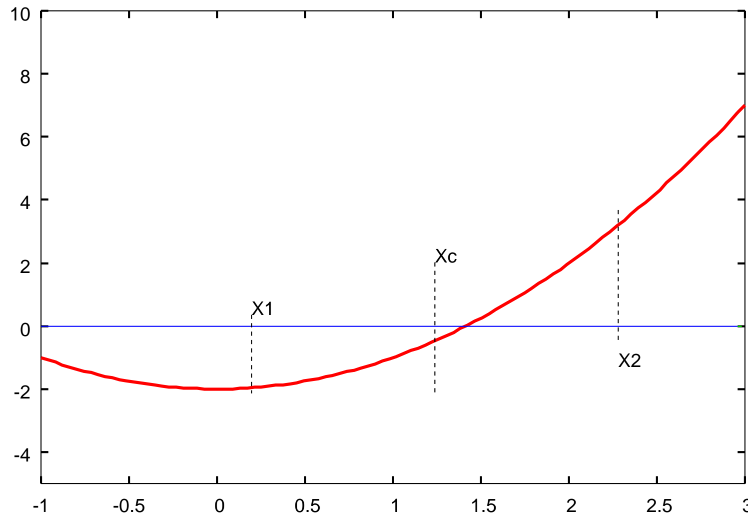

図 11: ここで考える 2 分法の概念図.

15.1　素朴な 2 分法で解く

ここでは簡単のため, $f(x) = x^2 - a^2$ に選び, この解 (もちろん, a が正の実数なら $x = \pm\sqrt{a}$) を数値的に求めることを考えます. この際, $f(x) = 0$ の解が範囲 : (x_1, x_2) の間に含まれていることがわかっているとしましょう. これは $f(x)$ の形があまり複雑でなければ割と難なく見積もることができるでしょう. このとき, $x = x_1$ と $x = x_2$ の中点を

$$x_c \quad = \quad \frac{1}{2}(x_1 + x_2) \tag{4}$$

と置きます. これをグラフで表すと図 11 のようになります. $f(x_c)$ の値の符号が $f(x_1)$ と同じであれば, $f(x) = 0$ の解は区間 (x_c, x_2) の中のどこかにあるわけですから, 区間 (x_1, x_2) の左端 x_1 を x_c で置き換えます. 逆に, $f(x_c)$ の符号が $f(x_2)$ と同じであれば, 区間の右端 x_2 を x_c で置きなおします. このようにして得られる新しい区間に対して, 上記の操作を繰り返して行くわけです. すると, この繰り返し回数が十分であれば, 最終的には両端 x_1, x_2 がともに $f(x) = 0$ の解 x に近づいて行くことになり, 結果として, この方程式 $f(x) = 0$ の良い近似値が得られるはずです. この手の数値解法を **2 分法**と呼んでいます.

それでは実際に上記のアルゴリズムを C 言語でコーディングし, 結果を見てみるために次の練習問題をやってもらいましょう.

練習問題 15.1 (※ LMS 入力問題)
$f(x) = x^2 - 2 = 0$ の解のうち, 正の値を持つものを 2 分法により求めたい. このとき次の 2 点に注意してプログラムを作成し, 実行せよ.

(1) 区間の両端の差の絶対値 $|x_1 - x_2|$ が 10^{-5} 以下になったときを収束点を x とし, 解 x を

```
x=******
```

の形式で表示せよ. ただし, はじめの区間を $(0, 2006)$ に選ぶこと.

(2) 繰り返し回数 i とそのときの x_1 の値 (区間の左端) と真の解 $\sqrt{2}$ とのズレ : $d = |\sqrt{2} - x_1|$ を

```
i=1 d=******
```

```
i=2 d=******
i=3 d=******
...............
............
..............
```

のような形式で表示し, 繰り返し回数とともに, この「誤差」がどのように減少していくのかを確かめよ.

※ LMS に投稿するプログラムは (1) の要求を満たすものだけでよい.
※ この手の漸化式の繰り返しに関しては前回 (第 4 回) の **LMS 入力問題** を思い出すと良い.

15.2　ニュートン法

この手の方程式の数値解を求めたい場合, **ニュートン法**と呼ばれる手法を用いることもできます.
この方法は方程式 $f(x) = 0$ の解の候補 $x = x_0$ で曲線 $y = f(x)$ に接する接線の方程式:

$$y = f(x_0) + f^{'}(x_0)(x - x_0) \tag{5}$$

の x 軸との交点：

$$x_1 \equiv x_0 - \frac{f(x_0)}{f^{'}(x_0)} \tag{6}$$

を解候補 x_0 の改良版とするもので, これを推し進めて n-番目の解候補と $n+1$-番目の解候補の間に成り立つ漸化式

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{7}$$

を反復的に解き, この収束点を $f(x) = 0$ の解とするものです.
$f(x) = x^2 - a^2$ に対する具体的なプログラムが教科書の 47 ページに載っています. これを参考にして次の練習問題をやってもらいましょう.

練習問題 15.2 (※ LMS 入力問題)
$f(x) = x^2 - 2 = 0$ の解のうち, 正の値を持つものをニュートン法により求めたい. このとき次の 2 点に注意してプログラムを作成し, 実行せよ.

(1) x_n, x_{n+1} の差の絶対値 $|x_n - x_{n+1}|$ が 10^{-5} 以下になったときを収束点 x とし, 解 x を

```
x=******
```

の形式で表示せよ. ただし, x の初期値を 2006.0 に選ぶこと.
(2) 繰り返し回数が i のときの x_i と真の解 $\sqrt{2}$ とのズレ：$d = |\sqrt{2} - x_i|$ を

```
i=1 d=******
i=2 d=******
i=3 d=******
...............
............
..............
```

のような形式で表示し, 繰り返し回数とともに, この「誤差」がどのように減少するのかを確かめよ. また, 前問 **練習問題 15.1** (2) で調べた 2 分法の場合と比べて, この誤差の減少の振る舞い方に違いが出るか否かを確認せよ.

※ LMS に投稿するプログラムは (1) の要求を満たすものだけでよい.

今週の LMS 入力問題

方程式

$$f(x) \quad = \quad x^2 - 4x + 1 = 0 \tag{8}$$

の小さな方の解をニュートン法で求めるプログラムを作成し, 結果を

```
x=*****
```

として表示するプログラムを作成せよ.