物理学情報処理演習

7. C言語の基礎 その２

配列、多次元配列

関数

標準ライブラリ

参考文献

“明解C言語　入門編”

ソフトバンククリエイティブ　柴田望洋

“プログラミング言語C 第２版”

共立出版　B. W. カーニハン／D. M. リッチー

大久保晋

E-mail: buturi-johoshori@tiger.kobe-u.ac.jp

http://extreme.phys.sci.kobe-u.ac.jp/staffs/okubo/lectures/Programming/index.html

配列　ー定義ー

- （同じ型の）変数の集合は配列として定義できる
- 配列の宣言
  - 配列名と型、及び配列の大きさを指定する
    - 例

```
int array[10];          /* 10の整数からなる配列 */
char moji[128];         /* 128個の文字からなる配列 */
```

  - 配列の大きさは定数でなくてはならない
    - よい例

```
#define SIZE 100
…
int array[SIZE];        /* 100個の整数からなる配列 */
```

    - 悪い例

```
int n = 100;
…
int array[n];           /* 100個整数からなる配列 */
```

配列　ー代入・参照ー

- 配列への代入・参照
  - 配列の各要素を参照（代入）するには、”[ ]”を使い添字で指定する
    - 添字は0から始まる
    - N個の要素の配列を指定するには、0… N-1

      例

```
#define SIZE 10
int array[SIZE];      /* 10個の整数からなる配列 */
int index;                          /* 要素を指定する添字 */
for (index=0; index<SIZE; ++index) {
      array[index] = index * 2 - 1;
}
```

  - 添字が配列の大きさを超えてもコンパイルエラーとはならない
    - 例

```
int a[10];                  /* 10個の整数からなる配列 */
a[100] = 100;               コンパイルエラーとはならない
```

    実行時に、セグメンテーション違反やバスエラーがでたり、おかしな振る舞いをする。もちろん計算結果も正しくなくなることがある

配列　ー代入・参照ー

- 配列への代入・参照
  - 配列の各要素を参照（代入）するには、”[ ]”を使い添字で指定する
    - 添字は0から始まる
    - N個の要素の配列を指定するには、0… N-1

      例

```
#define SIZE 10
int array[SIZE];      /* 10個の整数からなる配列 */
int index;                          /* 要素を指定する添字 */
for (index=0; index<SIZE; ++index) {
      array[index] = index * 2 - 1;
}
```

  - 添字が配列の大きさを超えてもコンパイルエラーとはならない
    - 例

```
int a[10];                  /* 10個の整数からなる配列 */
a[100] = 100;               コンパイルエラーとはならない
```

    実行時に、セグメンテーション違反やバスエラーがでたり、おかしな振る舞いをする。もちろん計算結果も正しくなくなることがある

配列　ーコピーー

- 配列全体の代入はできない
  - 悪い例

```
int a[100], b[100];        /* 100個の整数からなる配列 */
…
a = b;                     /* 配列全体の代入はできない */
```

- 配列のコピー
  - 配列をコピーするには要素毎に参照し代入する
    - 例

```
#define SIZE 100
int a[SIZE], b[SIZE];      /* 10個の整数からなる配列 */
int index;                 /* 要素を指定する添字 */
for (index=0; index<SIZE; ++index){
      a[index] = b[index];
}
```

オブジェクト形式マクロ -1-

- #define 指令を使って定数を名前で置き換える
  - マクロ名と定数を指定する

    例

```
#define NUMBER 5            /* NUMBER という名で5を置き換える */
#define GREETING “Hello”    /*GREETINGの名で”Hello”を置き換える*/
```

- 定数の意味をはっきりさせるため、名前をつける
  - よい例

```
#define SIZE 100                /* 100個の配列の大きさ */
#define PI 3.141592654          /* 定数 PI は 3.141592654 */
…
int array [SIZE];
…
for (i=0; i<SIZE; i++){
…
}
```

  - 悪い例

```
…
int array[100];
…
for (i=0; i<100; i++){
…
}
```

- マクロで定義しておけば配列の大きさを変えるときも間違えない。悪い例だとプログラム中の配列を何カ所も変更する必要がある。

オブジェクト形式マクロ -2-

- #define 指令は、プロプロセッサによってマクロ名をその後の文字列で置き換えるよう処理される
  - 文ではないので　セミコロン“ ; ”は付けてはいけない
  - マクロ名は変数ではないので代入できない

例

```
#define NUMBER 100    プリプロセッサ後
…                                …
NUMBER = 10;                100 = 10; コンパイルエラーとなる
```

- C++では、#define 指令でなく const 修飾子を使う

例

```
const int SIZE = 100;
…
int array[SIZE];
…
for (i=0; i<SIZE; i++){
    …
}
```

配列　―初期化―

- 配列の初期化
  - 配列を初期化するためには、各要素の初期値を”, ”で区切り、”{ }”でくくる
    - 例
      - int array[3] = { 0, 10, 20 };
  - 配列の代入には”{ }”は使えない（各要素毎に代入する）
    - 例
    - int array[3];
    - …
    - array = { 0, 10, 20, 40 };　　コンパイルエラーとなる
  - 初期化子が足りない配列要素は0に初期化
  - 全く初期化していない配列の全ての要素は不定
  - 大きさが与えられていない配列に初期化が行われると、その配列の大きさは初期化の数と一致する
    - 例
    - int array[ ] = { 0, 10, 20, 40 };　　arrayの大きさは4

演習７－１

- １０人分のテストの成績を読み込んで、その平均と標準偏差を求めるプログラムを作ろう
  - 成績を入れる整数配列をつくる、大きさはマクロで定義する
  - 成績はscanfで読み込む

```
#include <stdio.h>
#define NUMBER 10
int main(void){
int i;
int seiseki[NUMBER];

for (i=0;i<NUMBER; i++){
  printf(“%d-th point ?”,i);
  scanf(“%d”, &(seiseki[i]) );
  }
}
```

  - 平均と標準偏差を計算してみよう

    n人の集団でそれぞれがx_iの値を持つとき

    平均は $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

    標準偏差は $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}$

多次元配列

- 添字を複数使って、配列の配列を作ることができる
  - 例

```
int array[10][100];          /* 100の大きさの整数配列による */
                               /*   10の配列 */
array[i][j] = x;             /* i番目の配列のj番目の要素に代入 */
```

- 多次元配列の初期化
  - 多次元配列を初期化するときも、各要素の初期値を”, ”で区切り、”{ }”でくくる
  - 初期化子た足りない要素は0に初期化
    - 例

```
int array[2][4] = { { 0, 10, 20, 30}, {1, 12, 23, 34} };
```

演習７−2

- 次の２つの配列A, Bの和A + Bと差A - Bを求めよ

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}$$

行列A, Bの値は、初期化で与えよう。
A[i][j] を Aの i行 j列だと考える。

```
#define SIZE 3
int a[SIZE][SIZE] = { {1,0,1}, {0,0,0}, {1,0,-1}};
```

- 計算結果を見やすく出力するため行列c[3][3]を以下の様に出力しよう

```
printf("matrix c= \n");
for (i=0; i<SIZE; i++){
  for (j=0; j<SIZE; j++){
        printf("%d\t",c[i][j]);
  }
  printf("\n");
}
```

演習７−３

- N以下の素数の総数を調べるプログラムを次の方針で以前作った
- A) ある数 x が素数であるかどうかは、x未満の全ての数y=2..x-1で割り切れなければよい。つまり

```
for (y=2; y<x; ++y){
    if ( (x%y)== 0 ) break;    /* 割り切れた */
}
```

のループの直後でyがxと等しければ(breakされてなければ)素数である。
これを x = 2..N まで繰り返して計算

- B) Aの方法は無駄がある！　ある数 x が素数かどうかは、（x未満の）全ての数を試す必要はなく、素数だけ試すだけでよい

そこで、見つかった素数を配列 sosuu に順に入れていく。また、見つかった素数の数を n_sosuu としよう。
すると素数かどうかの判定は

```
for (i=0; i<n_sosuu; ++i){
  if ( (x % sosuu[i]) == 0 ) break;  /* 割り切れた */
}
```

のループの直後で i が n_sosuu と等しければ (breakされていなければ）素数であり、sosuu[n_sosuu] に x を代入し、n_sosuuを１つ増やせばよい。
この方針でプログラムを作ってみよう。

関数

- プログラムは関数の集まりである
  - プログラムの単位を関数と考える
  - 実行モジュールは必ずmain関数を持つ
- 関数を使う利点
  - 同じ処理を繰り返すときは関数呼び出しを使うことでソースコードの大きさや実行モジュールの大きさを小さくすることができる
  - 具体的な処理の詳細を見せずにプログラム全体の処理の流れを追うことができる
    - 例 1…10の階乗の計算

```
#include <stdio.h>
int factorial(int n)
{
  int i, x=1;
  for (i=1; i<=n; i++){
    x = x * i;
  }
  return(x);
}

int main(void)
{
  int n;
  printf("input n= ");
  scanf("%d", &n);
  printf("%d! = %d ¥n", n, factorial(n));
  return (0);
}
```

関数 factorial( )

main関数で
関数 factorial( )の呼び出し

関数

- 処理は関数呼び出しによって行われる
- 関数は
  - 名前
  - 返り値：型
  - 引数：型　＋　仮引数
    をもつ

  - void型　の使い方
    返り値がないときは、返り値の型をvoidにする
    引数がないときは、引数の型をvoidにする

- 関数を使用するには
  - 宣言：　関数名、返り値の型、引数の数と型
  - 定義（実装）：処理の内容
    が必要

関数の使用宣言と定義

- 関数（定義部）の構造
  - 宣言文
    - 変数の宣言
  - 実行文
    - 実際の処理を行う

返り値の型

引数の型

```
#include <stdio.h>                    プリプロセッサ処理

int main( void )                      主関数 main () の定義
{                                     main () の始め
   double menseki;
   double teihen, takasa;                          宣言文

   teihen = 2.0;
   takasa = 3.2;                                   実行文
   menseki = teihen * takasa/2.0;

   return (0);
}                                     main () の終わり
```

関数の使用宣言と定義 2

- 関数の呼び出しより先に、定義または使用宣言が無くてはいけない
- 関数の定義は、呼び出し部と別のファイルでも可

- 使用宣言(forward declaration)
  - 関数名
  - 返り値の型
  - 引数の数とその型

- 関数の定義（と実装）
  - 関数名
  - 変数の型
  - 引数の数とその型
    ＋
  - 引数の名前
  - 実際の処理

```
double f(int a);          使用宣言


int main(void)
{
  …
  a = f(I);       関数の呼び出し
  …
}

double f(int x)           関数 f( )の定義
{
  x = x*x;                関数 f( )の実装
  return x;
}
```

演習７－４：関数の定義

３辺の長さが与えられたとき、その三角形の面積を計算する関数

```
double Menseki( double teihen, double takasa )
```

を作成し、底辺と高さを与えて三角形の面積を計算するプログラムを作ろう！

詳細は次頁

演習７－４：関数の定義 program

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

double Menseki(double teihen, double takasa);

int main(void)
{
    double area;    /* area of a triangle */
    double takasa;  /* height of a triangle */
    double teihen;  /* base of a triangle */

    /* input teihen from standard input */
    printf("Input base of a triangle ");
    scanf("%lf", &teihen);

    printf("Input height of a triangle ");    /* input takasa */
    scanf("%lf", &takasa);

    /* area is calculated by formula */
    area = Menseki(teihen, takasa);

    printf("S = %f ¥n", area);
    return (0);
}

double Menseki(double teihen, double takasa)
{
    double area;
    area = (teihen * takasa)/2.0;
    return(area);
}
```

ライブラリ

- 標準Cライブラリ
- ANSI標準で用意されている関数群
  - ‘man -S 3 xxxx’でマニュアルを見ることができる。
  - 例　man -S 3 printf
- ヘッダーファイル(xxxx.h)に、これらのライブラリ関数の宣言が書かれている
- 関数の中身は、ライブラリ（libxxx.a）にある
  - 入出力　stdio.h
    例: int printf(const char* *format*, …)
  - 文字操作　ctype.h, string.h
    例:　int tolower(int *c*)
  - ユーティリティ　stdlib.h
    例:　atoi(const char* *s*)
  - 日付　time.h
    例:　time_t time(time_t* *tp*)
  - 算術　math.h
    例:　double sqrt(double a)
    Math Library のみ標準ライブラリではないので -lm オプションが必要（gcc 4.0 以降では不要?）

関数の引数

- Cでは、関数は"値渡し "
  - 呼び出した側の引数に入っている"値"が、呼び出された側の引数に渡される

例

```
double x = 1.0;
func (x);      /* 関数の呼び出し x の値を渡す */

void func(double y)
{
  y = y + 5.0;
}
```

呼び出した側の引数 x に入っている値 "1.0" が、呼び出された側の引数 y の値となる

呼び出された側の変数 y の値を変更しても、呼び出した側の変数 x は変化しない

関数の引数

- 配列を引数としてとる関数
  - 配列の宣言は
    - 呼び出す側では大きさを指定
    - 関数側では大きさ不定の配列
    - 型は一致していなければならない
    - 例

```
int array[10];    /* 10個の整数からなる配列 */

func(array);      /* 整数配列を引数にとる関数の
  */                                  /* 呼び出し */
void func(int a[])          /* 整数配列を引数に */
                                      /* とる関数の定義 */

{
…
}
```

    - 関数内で、配列の要素が変更された場合、関数から戻ってもその変更は有効です。

変数

- 変数は、値を格納する場所
  - 変数には、名前と型がある
    - `int I;`
    - `double x;`
  - 値を変えることができない変数（定変数）もある
    - `const double pi=3.141592;`
  - （同じ型の）変数の集合は配列として定義できる
    - `int array[100];`
    - `double space_vector[3];`

スコープ・ルール　*Advance*

- 名前（変数名・定数名・関数名）には、適用範囲がある
  - 局所
    - 関数内
  - 外部
    - ファイル内
    - 全ファイル

```
int           ext_variable;
static int int_variable;
extern int value[];
```

外部（全ファイル有効）

```
int func(double value)
{
double x, y;
static int count;
char* heap;
…
}
```

局所変数（関数内）

- 同じ名前がある場合は、局所変数の宣言が適用される。
- 全ての関数名は、全てのファイルに有効
  - 同じ関数名が、複数で定義されてはいけない

寿命、記憶域　*Advance*

- 変数には、寿命（有効期間）がある
  - 自動変数
    - 一回の関数呼び出し内でのみ有効
    - 関数の呼び出し毎に
      記憶域確保、初期化 が行われる
  - 静的変数
    - プログラム実行開始時に
      記憶域確保、初期化 が行われる
  - 動的変数
    - プログラム中で、明示的に
      記憶域確保、初期化 が行われる

```
int                     ext_variable;
static int    int_variable;
extern int    value[];
int func(double value)
{
    double              x, y;
    static int          count;
    char*               heap;
    heap = malloc(…);
    …
}
```

静的変数

自動変数

動的変数

プログラムの書き方：変数の名前、型

変数名、型をルールをもって付けよう！

変数名

- 全て英小文字（ +数字+_ ）で書く
  - 一文字目には数字や_を避ける
- 重要な変数は意味の分かる名前にする

型

- Cの組み込み型には
  - 整数： long, int, short, unsigned etc…
  - 実数： double, float
  - 文字： char
- 単精度実数”float”は使用しない
  - 現在ほとんどマシンの内部計算は倍精度で行われている
- 倍精度整数”long”は使用しない
  - 現在ほとんどのマシンでintとlongは同精度

プログラムの書き方：定数

- よく使用する定数、あるいは重要な定数には名前をつけて使用する
  - 列挙定数
  - マクロ
    - #define MAX 1000000
    - #define BELL ‘¥x7’

  を使用する
  - 定数名は
    - 全て英大文字（+数字+_）で書く
    - 一文字目には、数字や__を避ける
    - 重要な変数は、意味の分かる名前にする
  - マクロ#defineは、プリプロセッサで処理され、単純な文字列の置き換えのみ行われる

プログラムの書き方：実行文

- 実行文を書く際には、制御の流れが見えるようにする
  - 括弧
  - 改行
  - 字下げ

  を正しく使う
    - miエディターのCモードでは、tabを使うことで字下げをそろえることができる。

  - 一行に複数の文を入れない

    悪い例:　i = 5*j; j = j*j;

    よい例:　i = 5*j;

    　　j = j*j;

  - コメントは１行毎に行う

    悪い例:　/* a comment

    　　next comment */

    良い例:　/* a comment */

    　　/* next comment */

プログラムの書き方：実行文

- 実行文を書く際には、制御の流れが見えるようにする
  - 括弧
  - 改行
  - 字下げ

  を正しく使う
    - miエディターのCモードでは、tabを使うことで字下げをそろえることができる。

  - 一行に複数の文を入れない

    悪い例:　i = 5*j; j = j*j;

    よい例:　i = 5*j;

    　　j = j*j;

  - コメントは１行毎に行う

    悪い例:　/* a comment

    　　next comment */

    良い例:　/* a comment */

    　　/* next comment */

プログラムの書き方：実行文

- 字下げ・括弧を使ってループの深さを明らかにする
  - 推奨例

```
int func( double a )
{
    …
    return j;
}


if (…){
    i = j * k;
    …
} else {
    …
}


for ( i=0; i < A_MAX; ++i){
    …
}
```

プログラムの書き方：実行文

- まとまった処理は、関数として定義することで、プログラムの流れを明確にする
  - ループの深さは、せいぜい４～５段
  - １つの関数の定義は、どんなに大きくても200～300行に押さえる。
  - コメントをいれることで
    - 処理の内容
    - 事前条件
    - 事後条件

    等を明らかにする