情報処理B 2017　　第9回　配列

反復（繰り返し）が用いられる場面としては、前回のような「ユーザからの入力に対して随時応答する」ような例もありますが、最も多いのが「同じ種類のデータ列に対して同様の処理を行う」例です。

このような処理を実現するためのデータ構造が、**配列** (array) と呼ばれるものです。

9.1　配列を使う

9.1.1　配列を作る

例として、3 次元ベクトルの内積を考えてみましょう。ベクトル $\boldsymbol{x} = (x_1, x_2, x_3)$ と $\boldsymbol{y} = (y_1, y_2, y_3)$ の内積 $\boldsymbol{x} \cdot \boldsymbol{y}$ は、以下のように表されます。

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_1 y_1 + x_2 y_2 + x_3 y_3 = \sum_{i=1}^{3} x_i y_i \tag{1}$$

この計算を行うプログラムを書くためには、もちろん今までの講義内容が理解できていれば何とかなるはずです。しかし、2 つのベクトル $\boldsymbol{x}$ と $\boldsymbol{y}$ の各要素を別々の変数として宣言するのは、（ベクトルという形でグループ化されていることが）わかりにくくなります。さらに問題なのが、せっかく前回学んだ `while` 文が使えない、という点です。3 次元ベクトル程度ならまだしも、実際の研究開発ではもっと大きな次元のベクトル計算を行う必要があり、それを逐次処理で書くのは大変な作業です。

そこで、配列を用います。配列の宣言は、各要素の（共通の）データ型を最初に書き、次に配列の名前、続いて角括弧（`[]`）で括って要素数を記述します。

```
データ型 配列名[要素数];
```

たとえば、`double` 型の要素 3 つを持つ配列 `x` を宣言する場合は、次のように書きます。

```
double x[3];
```

このとき、配列の各要素は**配列名**の後に要素の番号（**添字**, subscript）を書くことで、変数と同様に参照することが可能になります。ただし、いくつか注意しなければならない点があります。まず、宣言で用いる配列要素数、および参照で用いられる添字は整数でなければなりません。さらに、

C 言語では配列添字はゼロ（0）から始まる

という点に注意する必要があります[1]。すなわち、上記のような宣言で配列 `x` を作成した場合、利用できる配列要素は `x[0]`, `x[1]`, `x[2]` の 3 つということになります。

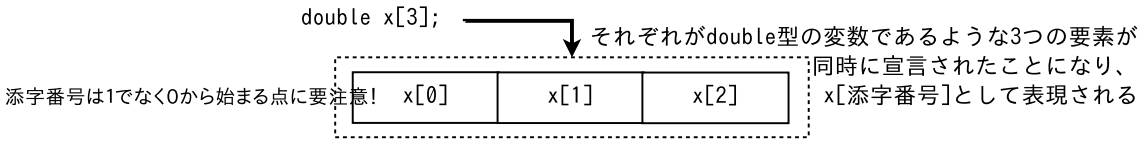

図 1: 配列宣言の概念

このような仕様に合わせてプログラムを書くには、まずベクトルの定義を $\boldsymbol{x} = (x_0, x_1, x_2)$ と $\boldsymbol{y} = (y_0, y_1, y_2)$ と読み替え、さらに式 (1) を

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_0 y_0 + x_1 y_1 + x_2 y_2 = \sum_{i=0}^{2} x_i y_i \tag{2}$$

[1]科学技術計算分野で広く用いられるプログラム言語である FORTRAN は、添字が 1 から始まる例です。また Visual Basic のように、宣言の時に要素数ではなく「最大の添字番号」を指定するような言語もあります。

と書き換える必要があります。式 (1) のように、数学では添字が 1 から始まるケースが多いのですが、プログラミングの際には混乱しないように気をつけてください。

ここまでの説明を踏まえて、例 9-1.c を見てください。前回学んだ while 文やインクリメント演算子、算術代入演算子をふんだんに使ったプログラムです。scanf() 関数を見ると、配列要素は事実上 1 つの変数と同じように使えることがわかってもらえるでしょう。

また、内積を求めるループの部分は、$\sum_{i=1}^{n}$ という形の計算を行う場合の常套手段です。総和を格納する変数を 0 にセットしておき、ループの中で要素を順に足していく、という流れになっています。

例: 9-1.c

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

int main(void)
{
  double x[3], y[3];
  double inner_product;
  int i;

  i = 0;
  while (i < 3) {                  /* xの要素を読むループ */
    printf("Input x[%d]: ", i);
    scanf("%lf", &x[i]);
    i++;
  }
  i = 0;
  while (i < 3) {                  /* yの要素を読むループ */
    printf("Input y[%d]: ", i);
    scanf("%lf", &y[i]);
    i++;
  }

  i = 0;
  inner_product = 0;
  while (i < 3) {                 /* 内積を求めるループ */
    inner_product += x[i] * y[i];
    i++;
  }
  printf("inner product is %f.¥n", inner_product);
  exit(0);
}
```

9.1.2 配列の位置づけ

今まで用いてきた定数や変数は、ある特定の型を持った 1 つのデータ、あるいはデータのための器でした。こうした定数や変数を、**スカラー型** (scalar type) のデータと呼びます [2]。これに対して、スカラー型のデータを複数まとめたデータを、**集合体型** (aggregate type) と呼びます。配列は、「同じ種類のデータ型を、指定した数だけまとめて一つのものとして扱う」集合体型の一種です [3]。

配列は、上に示したベクトルのようなものと考えてもよいですし、数列のようなものと捉えることもできます。あるいは、学生全員のレポート成績のように、個人の持つ同種の情報を並べたデータベースを想像してもらってもよいでしょう。

[2] スカラー型のデータには、第 3 回で紹介した char, short, long, int, float, double の 6 種類からなる**基本型** (basic type) の他に、これらを参照する**ポインタ型** (pointer type) が含まれます。ポインタは（重要ですが）非常に難しい概念なので、本講義では扱いません。

[3] これに対して、異なる種類のデータ型をまとめたものとして、**構造体型** (structure type) というものがあります。ポインタほどではないにしても、やや難しい内容ですので、やはり本講義では扱いません。

9.2　2 次元配列

配列の基本は 1 次元、すなわちベクトルや数列のように「1 つの添字」で要素を特定するようになっています。これを応用すると、2 つ以上の添字を使って要素を特定することができるようになります。このような多次元の配列が利用できれば、線形代数学で用いられる行列を表現することができるようになります。また、表計算ソフトの 1 つのシートも、行番号と列番号という 2 つの添字で要素（セル）を特定する 2 次元配列の一種です。

たとえば、`int` 型の 2×3 要素を持つ 2 次元配列 `a` を宣言するには、以下のように書きます。

```
int a[2][3];
```

この宣言の結果、以下の合計 6 つの `int` 型の要素が使えるようになります。

```
a[0][0], a[0][1], a[0][2],
a[1][0], a[1][1], a[1][2]
```

この宣言は、実際には「『`int` 型の要素を 3 つ持つ 1 次元配列』という要素を 2 つ持つ配列」という意味になっています。すなわち、2 次元配列とは「1 次元配列を要素として持つ配列」です。最初は集合体型を、スカラー型データの集まりと表現しましたが、実は集合体型を集めて新たな集合体型を作ることも可能です。その実例が、この 2 次元配列なのです。このように考えれば、3 次元配列、4 次元配列……と、何次元配列でも同じように宣言できることがわかるでしょう [4]。

線形代数で学ぶように、2×3 の行列 A と 3×2 の行列 B の積 $C = A \times B$ は 2×2 の行列です。C の (i, j) 要素である c_{ij} は

$$c_{ij} = \sum_{k=1}^{3} a_{ik} b_{kj}$$

で求められます。例として、

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$$

の場合に C を求める例 `9-2.c` を示します。

ここでは、新しい機能として配列の**初期化** (initialization) を行っています。変数も、宣言と同時に代入することで初期化できるということは、以前に例で示しました。配列の場合、代入するべき初期値を波括弧（`{}`）で囲んで、個々の初期値をカンマで区切ることで、全部または一部の要素をまとめて初期化することが可能です。2 次元配列は「配列の配列」ですから、初期値は二重に波括弧で囲まれることになります。このサンプルプログラムでの初期化によって、たとえば `a[][]` の各要素は次のように値が代入されたのと同じことになります。

```
a[0][0] = 1, a[0][1] = 2, a[0][2] = 3;
a[1][0] = 4, a[1][1] = 5, a[1][2] = 6;
```

[4] 次元数の上限には、特に定めがありません。C 言語の仕様では、コンパイラが（文法上）最低 12 次元の配列を解釈できるよう義務づけています。実際にある計算機で試してみたところ、$2 \times 2 \times \cdots \times 2$ の 23 次元配列までは問題なくコンパイルし実行できました。24 次元配列もコンパイルできましたが、「メモリが足りなくなったため」実行に失敗しました。

例: 9-2.c

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

int main(void)
{
  double a[2][3] = {{1, 2, 3},
                    {4, 5, 6}};
  double b[3][2] = {{7, 8},
                    {9, 10},
                    {11, 12}};
  double c[2][2];
  int i, j, k;

  i = 0;
  while (i < 2) {
    j = 0;
    while (j < 2) {
      c[i][j] = 0;
      k = 0;
      while (k < 3) {
        c[i][j] += a[i][k] * b[k][j];
        k++;
      }
      printf("%7.1f ", c[i][j]);
      j++;
    }
    printf("¥n");
    i++;
  }
}
```

9.2.1 2 次元配列を表のように使う

第 7 回で例を挙げた親子の血液型判定プログラムでは、話を簡単にするために、片方の親の血液型だけから子供の血液型の可能性を表示していました。今回は、両親の血液型に基づいて、子供の血液型の可能性を表示してみましょう。

if 文でまともに場合分けを考えると、16 分岐を作ることになり、現実的とは言えません。一方「子供の側」を基準にすると、表 1 に書いた (1)〜(6) の 6 パターンであることが分かります。

表 1: 両親と子供の ABO 式血液型の組み合わせ

		母			
		A	B	AB	O
父	A	A, O (1)	A, B, AB, O (2)	A, B, AB (3)	A, O (1)
	B	A, B, AB, O (2)	B, O (4)	A, B, AB (3)	B, O (4)
	AB	A, B, AB (3)	A, B, AB (3)	A, B, AB (3)	A, B (5)
	O	A, O (1)	B, O (4)	A, B (5)	O (6)

すなわち、

(1). A または O が生まれる

(2). A, B, AB または O が生まれる

(3). A, B または AB が生まれる

(4). B または O が生まれる

(5). A または B が生まれる

(6). O が生まれる

この 6 パターンに合わせて if 文を作ってやることも可能ですが、やはり条件文を書くのは大変です。そこで、表 1 をそのまま 2 次元配列として作ることにします。そして、たとえば表の値が (5) であるならば「A 型もしくは B 型」というように表示してやればよいのです。

例: 9-3.c

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

int main(void)
{
  int blood[][4] = {{1, 2, 3, 1},
                    {2, 4, 3, 4},
                    {3, 3, 3, 5},
                    {1, 4, 5, 6}};
  int father_blood, mother_blood;

  printf("Input blood type of your father (A=1, B=2, AB=3, O=4) : ");
  scanf("%d", &father_blood);
  printf("Input blood type of your mother (A=1, B=2, AB=3, O=4) : ");
  scanf("%d", &mother_blood);

  if (blood[father_blood - 1][mother_blood - 1] == 1)
    printf("You may have blood type A or O.¥n");
  if (blood[father_blood - 1][mother_blood - 1] == 2)
    printf("You may have bloopppd type A, B, AB or O.¥n");
  if (blood[father_blood - 1][mother_blood - 1] == 3)
    printf("You may have blood type A, B or AB.¥n");
  if (blood[father_blood - 1][mother_blood - 1] == 4)
    printf("You may have blood type B or O.¥n");
  if (blood[father_blood - 1][mother_blood - 1] == 5)
    printf("You may have blood type A or B.¥n");
  if (blood[father_blood - 1][mother_blood - 1] == 6)
    printf("You may have blood type O.¥n");
  exit(0);
}
```

第 7 回の応用トピックで触れた switch 文を使うと、もう少し記述はシンプルになるかもしれません。興味のある人は自分で書き換えてみましょう。

例 9-3.c でも、ここまでに説明していないテクニックを用いています。if 文の条件式に指定した配列要素に注目してください。先に注意したように、C 言語の添字番号は 0 から始まるのですが、入力しているのは 1 から始まる値です。このズレを埋めるために、変数 father_blood に 1 が代入されたら配列添字を 0 に、2 が代入されたら添字を 1 に……と、常に 1 つずらした参照を行うようにしています。式 (1) を式 (2) に変換したように、あらかじめ入力する値を 0 から 3 にするという方法もありますが、変換をプログラムの中で行うという方法もある、という例だと考えてください。

このサンプルプログラムでは、while ループを使っていませんし、論理演算子や等値演算子も使っていません。配列は復と組み合わせた時に最も力を発揮するのは確かですが、この例のように「一覧表として使う」ことでプログラムを簡単にすることができる効能もあります[5]。

9.3 応用トピック：do-while 文

ここで、もうひとつ別の繰り返し文である、do-while 文を紹介しましょう。do-while 文は以下のように記述されます。

[5] このような配列の使い方は、ルックアップテーブル (lookup table) と呼ばれます。今回示した例の他、その場で計算するのに時間がかかるような（限られた数の）関数値を必要とする場合など、様々な場面で用いられます。

```
do
  文
while (条件式);
```

while 文が最初に条件式を持っているのに対して、do-while 文では条件式が最後に置かれています。その結果、2 つの繰り返し文は以下のように異なる動作をします。

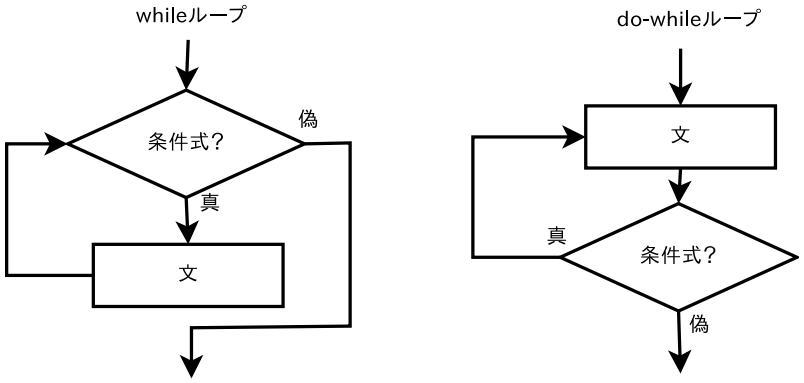

図 2: while 文と do-while 文の違い

do-while 文では、指定されている文は最低でも 1 回実行されることに注目してください。一方の while 文は、条件式の値によっては 1 回も実行されない可能性があります。そんな無駄なプログラムはあるだろうか、と疑問に思うかも知れませんが、実際に入力された値や計算結果次第では、1 回もループを行う必要がない（あるいはループするとまずい）こともあり得ます。こうした時には、当然 while 文を使う方がよいでしょう。一方で、とにかく文を最低 1 回は実行しないと、条件式のために必要な値が得られないようなこともあります。こうした場合には do-while 文を使う必要があります。

実際には、ほとんどの繰り返し処理は while でも do-while でも記述することができます。ただし、何度も言うようですが「プログラムの見通しの良さ」はどちらを用いるかで変わってきます。

9.3.1 数当てゲームの改良

前回のサンプルプログラムにあった数当てゲームを、do-while 文で書き換えてみましょう（例 9-4.c）。

もちろん、単に条件式を 1 にしてまえばやはり無限ループができるので全く同じ動作になりますが、ちょっと工夫して「0 を代入した時は（数を当てていなくても）プログラムを終了する」ようにしています。これと同じ処理を while 文で書く場合、最初に変数 guess_number を 0 でない値に初期化しておく必要があります。このように、入力の内容に応じて処理を行うという繰り返しは、do-while 文で書くと良いケースがしばしばあります [6]。

[6] この他に、1 回の計算ごとに「1 つ前の値と最新の値を比較して、その差（または比）が一定以上小さくなったら終了する」ような処理でも、最低 1 回の計算を必要とするため、do-while 文で書く方が見通しがよくなることがあります。こうした計算処理には、よく知られた**ニュートン法** (Newton's method) などがあります。

例: 9-4.c

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

int main(void)
{
  int secret_number;
  int guess_number;

  srand(time(NULL));              /* 乱数の生成 */
  secret_number = rand() % 9 + 1;

  do {
    printf("Make a guess the number: ");
    scanf("%d", &guess_number);
    if (guess_number != secret_number) {
      printf("You missed.¥n");
    } else {
      printf("Bingo!¥n");
      break;
    }
  } while (guess_number != 0);     /* 0が代入された場合は終了 */
  exit(0);
}
```