14　多次元配列 (続き)

課題 10. 3×3 行列に関する基本的な関数 (表示・演算など) を作成して実習 14.3 のプログラムを書き直した次頁のプログラム matrix2.c を完成させよ。

解答例 10.1. 関数 printmatrix() の定義のみ示す。又、行列の加法の関数を 2 通り例示しておく。

```
void printmatrix(int m[SIZE][SIZE])
{
    int i, j;

    for( i=0; i<SIZE; i++ )
    {
        for( j=0; j<SIZE; j++ )
        {
            printf("%3d,",m[i][j]);
        }
        printf("\n");
    }

    return;
}

void addmatrix(int x[SIZE][SIZE],int y[SIZE][SIZE],int z[SIZE][SIZE])
{
    int i, j;

    for( i=0; i<SIZE; i++ )
        for( j=0; j<SIZE; j++ )
        {
            z[i][j] = x[i][j] + y[i][j];
        }

    return;
}

void addmatrix2(int x[SIZE][SIZE],int y[SIZE][SIZE])
{
    int i, j;

    for( i=0; i<SIZE; i++ )
        for( j=0; j<SIZE; j++ )
        {
            x[i][j] += y[i][j];
        }

    return;
}
```


行列の積を計算する関数 multmatrix() の実装例は保留する。更に考えられたい。

Check Point:

- 動作確認をきちんと行なったか。
- 表示・演算などを関数化しているか。
- 行列のサイズは固定したとは言え、将来の変更に耐え得るように書いておきたい。例えば、
  `#define SIZE 4`
  とした時に、そのまま 4×4 行列が扱えるようになっているか。

以上の点が正しくできていない場合は、**要再提出**。

addmatrix(x,y,z) が z=x+y に、
addmatrix2(x,y) が x+=y に、
それぞれ相当。実際に計算で使うには、両方を用意しておくと便利なことが多い。

関数内での配列要素への代入により、呼出側でも配列要素の値が変更されていることを次頁の例で確認し、またその理由を理解せよ。

機能拡張については、期限にこだわらず試みられたい。

例．「int 型を成分とする 3 × 3 行列」を新たな型として、Mat 型という別名を定義してみた。

```
/*   matrix2.c   2010-06-21   */
#include <stdio.h>

#define SIZE 3

typedef int Mat[SIZE][SIZE];

void inputmatrix(Mat);
void printmatrix(Mat);
void addmatrix(Mat, Mat, Mat);
void addmatrix2(Mat, Mat);

int main(int argc, char** argv)
{
    Mat a, b, c;

    inputmatrix(a);
    printf("a=\n"); printmatrix(a);

    inputmatrix(b);
    printf("b=\n"); printmatrix(b);

    addmatrix(a,b,c);
    printf("a+b=\n"); printmatrix(c);

    addmatrix2(a,b);
    printf("a+b=\n"); printmatrix(a);
}
```

```
void inputmatrix(Mat m)
{
    int i, j;

    for( i=0; i<SIZE; i++ )
        for( j=0; j<SIZE; j++ )
        {
            scanf("%d",&m[i][j]);
        }

    return;
}

void printmatrix(Mat m)
{
    関数 printmatrix() の定義の中身は同じなので略
}

関数 addmatrix(), addmatrix2() の定義も同様なので略
```

型の別名定義

typedef int Mat[SIZE][SIZE];

で Mat 型が「int 型の 3 × 3 配列」の型の別名として定義され、この後、既存の型と同じように書ける。このような型を定義した時には、その型のオブジェクトを扱う適切な関数を作成して、それを介してアクセスすることとし、その型の定義や関数の中身を知らなくても、利用する側 (関数の呼出側、ここでは main() 内) では、その機能とプロトタイプだけ判れば使える、という形にすべきである。

別名定義しても配列型であることには変わりないので、関数に渡した時の振舞いなどには注意する必要がある。

一連の関数の作成と main() の作成とで分業することを想定せよ。

15　構造体

15–1　構造体の宣言

複素数や有理数など、既存の型 (int, double など) では収まらないデータを扱いたい場合に、既存の型を組み合わせて新しい型を定義する**構造体**を利用することが出来る。構造体とは、1 つ以上の (一般には異なる型の) 変数を組にして、一まとまりのものとして扱うものである。構造体を利用するにはまず、その型の定義を (通常) プログラムの冒頭部分で行う。

構造体: structure

例．複素数型 Complex を構造体として定義しよう。ここでは、2 つの実数 (実部・虚部) の組として 1 つの複素数を扱う方針で、double 型変数 2 つの組を複素数型として定義してみる。

```
struct Complex
{
    double re;
    double im;
};                                  /* ここまでで struct Complex 型の定義 */

typedef struct Complex Complex;     /* 型名の別名定義 */
```

構造体の型名にはキーワード struct が付くので、struct Complex 型として定義されるが、一々書くのは煩わしいので、通常 typedef を用いて型名の別名定義をする。これで Complex が struct Complex と同義となり、Complex というキーワードをあたかも型名であるかのように用いることが出来る。

C 言語に元々ある型名と区別するため、大文字を用いる (一文字目だけ、又は全部) ことが多い。

この例では、全てのメンバが同じ型だが、異なる型のメンバの組でも良い。メンバとしてポインタ型・配列型・他の構造体型などを含むことも出来る。

これで「Complex 型」という新たな型が利用できる。既存の型と同様に

```
Complex c;
```

のように Complex 型の変数を宣言して用いる。re と im とをこの構造体の**メンバ**(member) と呼ぶ。構造体変数のメンバには、例えば c.re のように

構造体変数名. メンバ名

としてアクセスする。

構造体定義とその別名定義とをまとめて、次のように書いてしまうことも多い。

```
typedef struct Complex    /* 実はここの Complex はなくてもよい */
{
    double re;
    double im;
}
Complex;
```

15–2　構造体の利用

構造体変数を扱う場合、そのメンバに個別に直接アクセスすることは勿論出来るが、その構造体変数を扱う基本的な関数を作成して、それを介して扱うのが普通であり、推奨される。その型の定義や関数の中身を知らなくても (変更されても)、利用する側 (関数の呼出側) では、その機能とプロトタイプだけ判れば使える、という形にすべきである。

構造体変数にその型の値を代入演算子 = で代入することや、関数の引数・返値として構造体変数を渡すことが出来る。構造体へのポインタや構造体の配列を利用することも出来る。

前掲の複素数型の場合、当然それらの間の演算を行ないたい訳だが、出来合いで用意されている訳では勿論ない。そこで、複素数を扱う基本的な関数を作成しよう。尚、構造体同士の比較も比較演算子 (==,!=など) では出来ないので、必要なら比較の為の関数を作成することになる。

一連の関数の作成とそれを利用するプログラムの作成とで分業することを想定せよ。関数の中身を改良版で置き換えたり、仮に内部で絶対値と偏角との組として極座標表示で扱う方針に変更したとしても、利用者側のプログラムを変更しなくてもよい、という状態が望ましい。

実習 15.1. Complex 構造体を用いたプログラムの例。

コンパイル時にリンカオプション -lm を忘れずに。

```
/*   complex.c   2010-06-28   */
#include <stdio.h>
#include <math.h>

typedef struct Complex
{
    double re;
    double im;
}
Complex;

Complex c_set( double, double );
void c_print( Complex );
Complex c_add( Complex, Complex );
double c_abs( Complex );

int main(int argc, char** argv)
{
    Complex a, b, c;

    a = c_set(1.0, sqrt(2.0));
    b = c_set(-3.0, 2.0);

    printf("a = "); c_print(a);
    printf("b = "); c_print(b);

    c = c_add(a, b);
    printf("a+b = "); c_print(c);

    printf("abs(a+b) = %f\n", c_abs(c));
}
```

```
Complex c_set( double re, double im )
{
    Complex c;

    c.re = re;
    c.im = im;

    return c;
}

void c_print( Complex c )
{
    printf("%f%+fi\n", c.re, c.im );

    return;
}

Complex c_add( Complex a, Complex b )
{
    Complex c;

    c.re = a.re + b.re;
    c.im = a.im + b.im;

    return c;
}

double c_abs( Complex c )
{
    return sqrt(c.re * c.re + c.im * c.im);
}
```

例. 実部・虚部を与えて Complex 型変数の値を設定する関数 c_set() や、Complex 型変数の値を表示する関数 c_print() を、Complex 型変数へのポインタを渡す形で書いてみた。

前にも書いたが、一般には返値として返せるならその方が便利。但し、変数の値を渡す場合には値のコピーを作ることになるので、メンバの個数が多いなど複雑な型の場合には、ポインタ渡しの方が効率的なこともある。

こんな所で、printf 変換の + フラグが意外に便利。

```
void c_set( double re, double im, Complex *c )
{
    (*c).re = re;
    (*c).im = im;

    return;
}

void c_print( Complex *c )
{
    printf("%f%+fi\n", (*c).re, (*c).im );

    return;
}
```

考察 15.1.1. 関数 c_print() の 2 つの仕様で、引数として関数に渡すデータのバイト数を比較せよ。

注. 演算子の優先順位の関係で (*c).re の括弧が必要なことに注意。構造体へのポインタはしばしば用いられるので、(*c).re のことを c->re と書く別記法も用意されている。

「ポインタ変数 c の指し先の re というメンバ」という感じが判り易いかと。

実習 15.2. 前頁の complex.c の関数 c_set() , c_print() を、上の例のものに置き換えよ。

実習 15.3. 前頁の complex.c に減乗除算の関数 (c_sub(), c_mul(), c_div()) を追加し、それを用いた計算を main() に追加して動作を確認せよ。

課題 11 (〆切 学期末迄 (詳しい期日は追って連絡する)). 構造体を用いて「有理数型」Rational を定義した上で、代入・表示・加減乗除・比較などの一連の基本的な関数を作成し、main() 内でその動作を確認できるような計算をせよ。

有理数を扱うライブラリを自作することに相当する「大きい」課題である。型そのものの設計や関数プロトタイプの設計など「仕様」の決定や、或はそもそもどのような関数 (機能) が必要か、などの判断が要求されている。