プログラミングおよび実習 I・第 3 回

2003 年 4 月 23 日

1　関数呼び出し (その 2)

配列を引数として関数に渡す場合には、int 型や double 型などの単純なデータを渡す場合との違いに注意する必要があります。この違いを確認するために、まずは int 型の値を引数とする関数の場合を調べてみましょう。次は、入力された正の整数の階乗を出力するプログラムです。

```
 1 #include <stdio.h>
 2
 3 /* fact : 引数 n の階乗 n! を計算して返す関数 */
 4 int fact(int n)
 5 {
 6     int x = 1;
 7
 8     while (n > 1) {
 9         x *= n;
10         n--;
11     }
12     return x;
13 }
14
15 /* main : 入力された正の整数の階乗を出力する */
16 int main ()
17 {
18     int n;
19
20     printf("n = ");
21     scanf("%d", &n);
22     printf("n! = %d\n", fact(n));
23     return 0;
24 }
```

21 行目の scanf の呼び出しによって、 18 行目で宣言されている変数 n に、キーボードから入力された整数が格納されます。次の 22 行目では、この変数 n の値を実引数として関数 fact が呼び出されています。関数 fact が呼び出されると、 4 行目で宣言されている仮引数 n にこの実引数の値 (18 行目で宣言されている変数 n の値[1]) がまず格納されてから、 5 行目以降の関数定義本体が実行されていきます。

10 行目では、関数 fact の仮引数 n の値が 1 ずつ減されますが、これは、関数 main の方の変数 n の値には影響を与えません。関数 fact へ渡されたのは、 22 行目での (main の) 変数 n の**値** (変数 n に格納されている数値データ) であって、これが、別の変数 (fact の仮引数) n に代入 (コピー) されてから、関数 fact の定義本体が実行されているからです。

このように、実引数の**値**がコピーされて、呼び出される関数に渡されることを**値呼び** (call by value) と呼びます。 C 言語では、関数の呼び出しは基本的にすべて**値呼び**で行われます。

これに対して、実引数の値がどこに記憶されているかという情報 (このプログラムの場合は実引数となった変数 n の**アドレス**) を呼び出される関数に渡し、呼び出された側では、渡された場所を調べて実

[1]関数呼び出し (22 行目) での実引数が、関数 fact の仮引数 n と同じ名前の変数となっているのは単なる偶然です。 18 行目に宣言された変数 n と、 4 行目に宣言された仮引数 n は、 (名前は同じであるが) 異なる変数であって、そこに記憶される値はそれぞれ独立していることに注意してください。

引数の値を参照したり、そこに格納されている値を変更したりする方法のことを**参照呼び** (call by reference) と呼びます。参照呼びでは、呼び出された側で、呼び出した側の実引数の値を変更することもできます[2]。

配列を引数とした関数呼び出し　例えば

```
int a[10], b[10];
```

のように宣言された配列 a、b に対して、配列 b の内容を、すべて配列 a にコピーしたい場合は、

```
for(i = 0; i < 10; i++)
    a[i] = b[i];
```

のように配列の各要素をコピーすることになります[3]。配列全体を

```
a = b;
```

のような方法でコピーすることはできません。代入したり参照したりできるのは、あくまで配列を構成している 1 つ 1 つ要素です。これと同じことが関数呼び出しでも起ります。次のプログラムを見てみましょう。

```
 1 #include <stdio.h>
 2
 3 #define N        10
 4
 5 /* fact : 配列 a の要素 a[i] の階乗を返す関数 */
 6 int fact(int a[N], int i)
 7 {
 8     int x = 1;
 9
10     while (a[i] > 1) {
11         x *= a[i];
12         a[i]--;
13     }
14     return x;
15 }
16
17 /* main : 入力された10個の正の整数の階乗を出力する */
18 int main ()
19 {
20     int i, a[N];
21
22     for (i = 0; i < N; i++) {
23         printf("a[%d] = ", i);
24         scanf("%d", &a[i]);
25     }
26     for (i = 0; i < N; i++) {
27         printf("a[%d]! = %d\n", i, fact(a, i));
28     }
29     return 0;
30 }
```

[2] これは、一見便利そうにも見えますが、関数呼び出しが予期せぬ副作用を持ってしまう危険性もあります。

[3] もちろん i は「int i;」のように宣言された変数であるものとします

このプログラムは、 10 個の正の整数を入力し、それぞれの階乗を出力するものです。 C 言語では値呼びで関数呼び出しが行われますが、配列全体の代入ができないのと同様に、配列全体のコピーを関数に渡すこともできません[4]。

このプログラムの 27 行目では、配列名 a を第 1 引数として「fact(a, i)」という関数呼び出しが行われていますが、 C 言語では、配列名を式として使った場合、その配列全体のデータを表すのではなく、その配列の先頭要素が (メモリ中で) 格納されている場所 (**アドレス**) を意味する (その値として持つ) ことになっています。配列は、その要素データの並びとしてメモリ中に連続的に記憶されます。例えば、int 型のデータ 1 つを記憶するのに必要なメモリの大きさが 4 byte であった場合、 20 行目で宣言されている配列 a は、あるアドレス p を始まりとして、メモリ中に

p	p+4	p+8	p+12	p+16	p+20	p+24	p+28	p+32	p+36
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

のように配置されて記憶されます。

27 行目の「fact(a, i)」という関数呼び出しで a という式の意味する値は、配列 a の先頭要素 a[0] のアドレス p となります。この値 (アドレス) が関数 fact の仮引数 a に代入され (また、第 2 の実引数 i の値が、fact の仮引数 i に代入されてから) fact の関数定義本体 (7 行目以降が) 実行されます。

その配列の個々の要素のデータ型 (その要素の記憶に必要なメモリの大きさ) があらかじめ分かっていれば、その配列の先頭要素のアドレスから配列の各要素の記憶場所 (アドレス) が計算できますから、関数 fact の中で配列の要素 a[i] への代入やその参照を行うことができるわけです。

ここで注意しなければならないのは、呼び出された関数 fact へ渡されたのは (20 行目で宣言されている) 配列 a 全体のコピーではなく、その (先頭要素の) アドレスであり、関数 fact のプログラムがその仮引数 a を介して代入したり参照したりする配列の実体は、呼び出した側の配列と同じものであるということです。関数 fact は 12 行目で a[i] の値を変更していますが、これは、関数 main で宣言されている配列 a の内容を変更していることになります[5]。

配列型の仮引数の宣言　先のプログラムでは、関数 fact の仮引数の宣言は

```
int fact(int a[N], int i)
```

のようにされていましたが、配列型の仮引数 a の要素数を省略して、

```
int fact(int a[], int i)
```

のように書くこともできます。これは、先に述べたように、配列の要素のデータ型と先頭要素のアドレスさえ分かれば、その配列の各要素への代入や参照が可能となるためです。 C 言語では、配列要素へのアクセス (代入や参照) 時に、指定された添字が 0 から「その配列の要素数−1」までの範囲に収まっているかどうかは (自動的には) チェックされませんので、配列型の仮引数 a の要素数は何の役にも立たないことになります。

一方、 2 次元以上の配列では、最初の添字を除いて要素数を省略することはできません。例えば

[4] **構造体**や**共用体**と呼ばれるデータ型で配列を包めば、配列全体をコピーしたり、関数に渡したりすることが (実質的には) 可能となります。

[5] これは、たとえ 6 行目の a という仮引数名を別の名前に変えたとしても同じです。

```
int a[3][5];
```

のように宣言された 2 次元配列 a は、あるアドレス p を始まりとして、メモリ中に

p	p+4		p+16	p+20	p+24		p+36	p+40	p+44		p+56	
a[0][0]	a[0][1]	...	a[0][4]	a[1][0]	a[1][1]	...	a[1][4]	a[2][0]	a[2][1]	...	a[2][4]	

のように配置されて記憶されます[6]。最後 (2 次元配列の場合は 2 番目) の添字が先に変化することに注意してください。

2 次元配列では、指定された添字 i、j から配列要素 a[i][j] が格納されているアドレスを計算するためには、先頭要素 a[0][0] のアドレスの他に、2 番目の添字が動く範囲の大きさが必要となります。よって、この例のような配列 a を仮引数とする関数では、

```
int f(int a[3][5])
```

あるいは

```
int f(int a[][5])
```

のように仮引数を宣言することは可能ですが、

```
int f(int a[3][])
```

や

```
int f(int a[][])
```

のように、2 番目以降の添字部分の要素数を省略することはできませんので注意して下さい。

2 ゲーム盤の初期化部分の関数化

次のプログラム othello05.c は、前回の演習問題で作成した othello04.c でゲーム盤の状態 (配列 board の内容) を初期化していた部分を関数 reset として独立させ、この関数 reset を main から呼び出すようにしたものです。

othello05.c

```
 1 #include <stdio.h>
 2
 3 #define BLACK           (1)         /* 黒石を表す定数 */
 4 #define WHITE           (-BLACK)    /* 白石を表す定数 */
 5 #define NONE            (0)         /* 石がないことを表す */
 6
 7 #define BOARD_SIZE      (8)         /* オセロボードの縦横のマス数 */
 8
 9 /* 盤面を初期化する */
10 void reset(int b[][BOARD_SIZE])
11 {
12     int x, y;
13
14     /* すべてのマスをクリアする */
```

[6] int 型のデータの大きさは 4 byte としています。

```
15      for (x = 0; x < BOARD_SIZE; x++) {
16          for (y = 0; y < BOARD_SIZE; y++) {
17              b[x][y] = NONE;
18          }
19      }
20
21      /* 中央に4つの石を置く */
22      b[BOARD_SIZE/2-1][BOARD_SIZE/2-1] = WHITE;
23      b[BOARD_SIZE/2-1][BOARD_SIZE/2] = BLACK;
24      b[BOARD_SIZE/2][BOARD_SIZE/2-1] = BLACK;
25      b[BOARD_SIZE/2][BOARD_SIZE/2] = WHITE;
26  }
27
28  int main()
29  {
30      /* 盤面の状態 (BLACK/WHITE/NONE) */
31      int board[BOARD_SIZE][BOARD_SIZE];
32      int x, y;
33
34      reset(board);
35
36      while (1) {
37          /* 列番号を表示する */
38          for (x = 0; x < BOARD_SIZE; x++)
39              printf(" %d", x+1);
40          printf("\n");
41          /* 各行を表示する */
42          for (y = 0; y < BOARD_SIZE; y++) {
43              printf("%d", y+1);
44              for (x = 0; x < BOARD_SIZE; x++) {
45                  if (board[x][y] == BLACK)
46                      printf("●");
47                  else if (board[x][y] == WHITE)
48                      printf("○");
49                  else
50                      printf("□");
51              }
52              printf("\n");
53          }
54
55          /* 列番号を入力する */
56          do {
57              x = 0;
58              printf("列 = ");
59              scanf("%d", &x);
60          } while (x <= 0 || BOARD_SIZE < x);
61          /* 行番号を入力する */
62          do {
63              y = 0;
64              printf("行 = ");
65              scanf("%d", &y);
66          } while (y <= 0 || BOARD_SIZE < y);
67          /* 石を置く */
68          board[x-1][y-1] = BLACK;
69      }
70
71      /* プログラムを終了する */
72      return 0;
73  }
```

3 演習問題

3.1 例題プログラムを試す

講義で解説した othello05.c というプログラムを自分で作り[7]、コンパイル、実行してみなさい。完成したら、「mprog1 othello05.c」のようにして mprog1 コマンドを実行し、正しくできているかどうかチェックしてください。正しくできている場合は、そのプログラムが提出されたことになります。プログラムは Prog1 というディレクトリに作ることを忘れないようにしましょう。以下の 3 つの演習問題も同様です。

3.2 盤面の表示と入力部分の関数化

まず、othello05.c を othello06.c にコピーしなさい。その後 othello06.c の関数 main の定義を次のように変更するとともに、新しい 2 つの関数 showBoard と inputMove の定義を追加しなさい。

othello06.c の main

```
int main()
{
    /* 盤面の状態 (BLACK/WHITE/NONE) */
    int board[BOARD_SIZE][BOARD_SIZE];

    reset(board);
    while (1) {
        showBoard(board);
        inputMove(board);
    }
    return 0;
}
```

3.3 すでに石のあるマスには置けないようにする

まず、othello06.c を othello07.c にコピーしなさい。その後 othello07.c を変更して、次の実行例のように、すでに石のあるマスが指定された時には「そこには置けません」というメッセージを表示して、ユーザーにもう一度入力を促すようにしなさい。

othello07 の実行例

```
s1609h017% ./othello07
列 = 4
行 = 3
 1 2 3 4 5 6 7 8
1□□□□□□□□
2□□□□□□□□
3□□□●□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
列 = 5
行 = 4
そこには置けません
列 = 5
行 = 1
 1 2 3 4 5 6 7 8
```

[7]前回作成した othello04.c をコピーして変更すると簡単かも知れません

```
1□□□□●□□□
2□□□□□□□□
3□□□●□□□□
4□□□○●□□□
5□□□●●□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
列 = ^C
```

3.4 黒石と白石を交互に置くようにする

まず、othello07.c を othello08.c にコピーしなさい。その後 othello08.c を変更して、次の実行例のように、置かれる石が黒石、白石、黒石、白石、 ... と交互に変っていくようにしなさい。ただし、実行例に続く**実現の方針**にしたがってプログラムを作りなさい。

othello08 の実行例

```
s1609h017% ./othello08
 1 2 3 4 5 6 7 8
1□□□□□□□□
2□□□□□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
列 = 4
行 = 3
 1 2 3 4 5 6 7 8
1□□□□□□□□
2□□□□□□□□
3□□□●□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
列 = 5
行 = 2
 1 2 3 4 5 6 7 8
1□□□□□□□□
2□□□□○□□□
3□□□●□□□□
4□□□○●□□□
5□□□●●□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
列 = 4
行 = 4
そこには置けません
列 = 7
行 = 2
 1 2 3 4 5 6 7 8
1□□□□□□□□
2□□□□○□●□
3□□□●□□□□
4□□□○●□□□
5□□□●●□□□
6□□□□□□□□
7□□□□□□□□
```

```
8□□□□□□□□
列 = ^C
s1609h017%
```

実現の方針　関数 inputMove に int 型の引数を追加し、この追加された引数 (第 2 引数) で、置く石の色が指定できるように inputMove の関数定義を変更します。また、関数 main では、int 型の変数 turn の宣言を追加し、この変数で次に置かれる石の色 (BLACK または WHITE) を覚えておくようにし、turn の値を第 2 引数として、関数 inputMove を呼び出すようにします。

```
#define OPPONENT(t) (-(t))
```

というようなマクロ定義をしておくと、関数 main の中で置く石の色 (手番) を変えるのは

```
turn = OPPONENT(turn);
```

を実行するだけでできます。

次回の予定

次回は othello08.c をさらに発展させて、1 から 8 までの列番号を a から h までの英文字で表すように変更し、それに合わせて、d2 のような 2 文字の入力で石を置くマスを指定できるようにします。プログラムの実行例は次のようになります。

```
 a b c d e f g h
1□□□□□□□□
2□□□□□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = d2
 a b c d e f g h
1□□□□□□□□
2□□□●□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●●□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = g7
 a b c d e f g h
1□□□□□□□□
2□□□●□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●●□□□
6□□□□□□□□
7□□□□□□○□
8□□□□□□□□
位置 = ^C
```

プログラミングおよび実習 I・第 3 回・終り