プログラミングおよび実習II・第 4 回

2003 年 10 月 15 日

1 バブルソートの復習

1.1 実力チェック

前回、バブルソートのアルゴリズムで配列の内容を昇順に整列するプログラムとして、次のような関数 bubblesort を考えました。

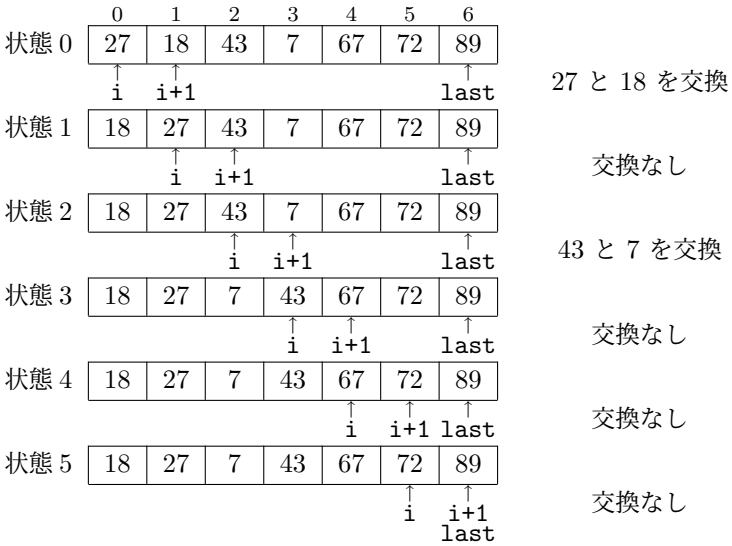
前回 2 ページの bubblesort のプログラム

```
 1 void bubblesort(int data[], int num)
 2 {
 3     int last, i;
 4     int tmp;
 5
 6     for (last = num-1; 0 < last; last--) {
 7         for (i = 0; i < last; i++) {
 8             if (data[i] > data[i+1]) {
 9                 tmp = data[i];
10                 data[i] = data[i+1];
11                 data[i+1] = tmp;
12             }
13         }
14     }
15 }
```

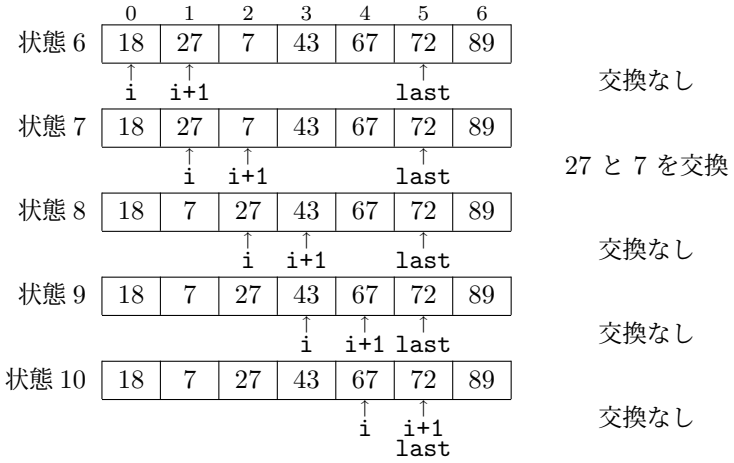
ところが、この関数が行っている処理をよく吟味してみると、かなり無駄な処理を行っていることに気づきます。たとえば、次のような内容の配列 data が与えられた時のことを考えてみましょう。要素の個数 num は 7 です。

	0	1	2	3	4	5	6
data	27	18	43	7	67	72	89

まず、変数 last の値を 6 として、添字 0 から添字 last まで、隣り合った 2 つのデータ (添字 i と i+1) を比較し、小さい方が前に来るように (必要なら交換) するという操作を、i を 1 ずつずらしながら行っていきます。 このときの様子は以下のようになります。



この後、変数 last の値を 1 減らして、添字 0 から添字 last まで、隣り合った 2 つのデータを比較し、小さい方が前に来るように (必要なら交換) するという操作をさらに行います。



ここで、状態 3 から状態 5 に至るときの 2 組の要素 (43 と 76、および 67 と 72) の比較が、もう 1 度、状態 9 以降で行われていることに気づきます。

変数 last の値が 6 のときの (状態 0 から状態 6 へ至るまでの一連の) 処理では、状態 3 以降では要素の交換は行われていません。状態 3 では、配列 data の添字 3 の要素に、それより先頭側の要素の最大値が移動してきているはずですので、ここから状態 6 に至るまでに要素の交換がなかったということは、状態 6 では、この配列の添字 3 以降の要素はすでに整列が完了してしまっていることを意味します。前回の 2 ページのプログラムでは、それにも関わらず変数 last の値を (1 だけ減らして) 5 として、同様の処理を続けていますが、本来なら状態 6 での last の値は 2 になっていてもよいわけです。

以上のような考察に基づく次の実現の方針に従い、関数 bubble ソートの定義をより効率のよいプログラムに変更しなさい。ただし、解答用紙のプログラム名は「prog4-1.c」とし、関数 bubblesort の (変更後の) 定義のみを書きなさい。

実現の方針 (1) 整数変数 last を用意し、これを配列の末尾の要素の添字で初期化しておく。

(2) 変数 last の値が 0 より大きい間、次の 2 つの作業を繰り返せば、ついには配列全体を昇順に整列させることができる。

(2-1) 配列の先頭から末尾に向けて、隣り合った 2 つのデータを比較し、小さい方が前に来るように (必要なら交換) するという操作を、添字を 1 ずつずらしながら、添字 last−1 と last の組まで繰り返す。もし、この中で 1 度も交換が必要なかった場合は、すでにこの配列の整列は完了していることになる。交換が必要だった場合は、最後に交換を行った組の前方側 (添字が若い側) の要素の添字 i を記憶しておく。この操作で、添字 i 以降の要素はすべて、添字 i 未満のどの要素より大きいものばかりとなり、しかも、配列の添字 i 以降は昇順に整列している状態となる。

(2-2) 変数 last の値を (2-1) で得られた i で置き換える。

2　ソーティングプログラムのテスト

これまで、選択ソートとバブルソートという 2 つのアルゴリズムに基づいてソーティングを行うプログラムを何通りか作ってきました。もっと複雑なアルゴリズムのプログラミングに進む前に、この節では、自分の書いたソーティングプログラムが、目標通りの仕事を行っているかをテストする方法を考えてみます。

もっとも単純な方法は、これまでの実行例で見てきたように、適当に配列内容の初期値を入力して、その出力結果が入力の内容に見合ったものになっているかを自分で調べることです。ですが、この方法だと、そのとき入力してみたデータの並びに対して偶然うまく行っただけで、別のデータに対してはプログラムがうまく働かないかも知れないという疑念を拭いさることができません。また、人間が結果を調べていますので、何らかの見落しや見間違いがあるかも知れません。もう少し信頼性の高いテストを行う方法はないでしょうか。

2.1　ソーティングプログラムの出力のチェック

ソーティングプログラムにある入力 (配列中に格納されたデータ) を与えたとして、プログラムの出力が期待したものになっているかどうかを調べるには次の 2 点をチェックすればよいはずです。

(1) 出力が入力の並び替えになっていること — なくなってしまったデータや突然現れたデータがないこと。

(2) 出力が昇順 (降順) に並んでいること — 隣り合った 2 つの要素の組それぞれについて、先頭側の要素が末尾側の要素以下 (以上) になっていること。

これを人間の目で行うのでは大変ですし間違いを避けられませんから、このチェックをプログラムにさせることを考えます。たとえば、以下のような関数群を定義して利用しましょう[1]。

関数 checkperm と 関数 checkorder

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 typedef int Boolean;
 5 #define TRUE            (1)
 6 #define FALSE           (0)
 7
 8 #define MAX_DATA        (1000)
 9
10 /* 配列 new[] の先頭 num 個の要素が、配列 old[] の先頭 num 個
11    の要素の並び替えになっているかどうかをチェックする。 */
12 void checkperm(int new[], int old[], int num)
13 {
14     Boolean used[MAX_DATA];
15     int i, j;
16
17     /* used[] をクリアする */
18     for (i = 0; i < num; i++)
19         used[i] = FALSE;
20     /* 各 old[i] を new[] の中から探す */
```

[1] このプログラムの 4 行目では、Boolean を int の別名として型定義しています。typedef を使った型定義に関しては付録を参照してください。

```
21      for (i = 0; i < num; i++) {
22          for (j = 0; j < num; j++) {
23              if (!used[j] && old[i] == new[j]) {
24                  used[j] = TRUE;
25                  break;
26              }
27          }
28          if (j >= num) {
29              printf("整列前の添字%dに対応する要素が整列後に"
30                     "ありません。\n", i);
31              printf("整列前の配列内容:");
32              showdata(old, num);
33              printf("整列後の配列内容:");
34              showdata(new, num);
35              exit(EXIT_FAILURE);
36          }
37      }
38  }
39
40  /* 配列 new[] の先頭 num 個の要素が昇順に並んでいるかどうかを
41     チェックする。 */
42  void checkorder(int new[], int old[], int num)
43  {
44      int i;
45
46      for (i = 1; i < num; i++) {
47          if (new[i-1] > new[i]) {
48              printf("整列後の添字%dと%dが昇順になっていません。\n",
49                     i-1, i);
50              printf("整列前の配列内容:");
51              showdata(old, num);
52              printf("整列後の配列内容:");
53              showdata(new, num);
54              exit(EXIT_FAILURE);
55          }
56      }
57  }
```

関数 checkperm は、int 型の配列 new の先頭 num 個の要素が、配列 old の先頭 num 個の要素の並び替えになっているかどうかをチェックします[2]。また、関数 checkorder は、int 型の配列 data の先頭から num 個の要素が昇順に並んでいるかどうかをチェックします。これらの関数がおかしな所を発見した場合は、関数 showdata [3]を呼び出して整列前と整列後の配列の内容を表示しています。

次の関数 checksort は、与えられた入力を関数 bubblesort が正しく整列するかどうかを、これら 2 つの関数を利用してチェックしています。このプログラムでは、1 行目の SORT のマクロ定義を書き換えることで、チェックしたい関数を変更することができるようになっています。

関数 checksort

```
1  #define SORT(d, n)      (bublesort(d, n))
2
3  void checksort(int old[], int num)
4  {
5      int new[MAX_DATA];
```

[2]C 言語では、このプログラムの 29 行目から 30 行目のように、複数の文字列リテラルを並べて書いておくと、それらを 1 つの文字列リテラルに連結して処理してくれます

[3]第 2 回の 6 ページ参照

```
 6     int i;
 7
 8     /* すべてのデータを new へコピーする */
 9     for (i = 0; i < num; i++)
10         new[i] = old[i];
11     /* new の整列を行う */
12     SORT(new, num);
13     /* 整列結果が元の内容の並び替えになっているかをチェックする */
14     checkperm(new, old, num);
15     /* 整列結果が昇順になっているかをチェックする */
16     checkorder(new, old, num);
17 }
```

2.2 いろいろな入力を生成する

前節の関数 checksort を利用すれば、与えられたデータが正しく整列されたかを簡単にチェックすることができます。後は、いろいろなデータを与えてみて、どんな場合でもうまく整列できるかを確認してみるだけです。これもプログラムにやらせてみることにします。

```
 1 #define N_TESTS         (100000)
 2 #define TEST_SIZE       (20)
 3
 4 int main()
 5 {
 6     int data[MAX_DATA];
 7     int range;
 8     int num, i, j;
 9
10     for (i = 0; i < N_TESTS; i++) {
11         /* データの個数を[1 ... TEST_SIZE]の範囲の乱数で決める */
12         num = rand() % TEST_SIZE + 1;
13         /* データの拡がり具合を[1 ... TEST_SIZE]の範囲の乱数で決める */
14         range = rand() % TEST_SIZE + 1;
15         /* 各データを[-range/2 ... range/2]程度の範囲の乱数で生成する */
16         for (j = 0; j < num; j++)
17             data[j] = rand() % range - range/2;
18         /* このデータでテストする */
19         checksort(data, num);
20     }
21     printf("すべてのチェックに合格しました。\n");
22     return EXIT_SUCCESS;
23 }
```

このプログラムでは以下のようなテストを 100000 回行っています。

(1) まず、テスト用のデータの個数を 1 から 20 までの乱数で決める。

(2) 次に、データの拡がり具合 (最大値と最小値の差 +1) を、 1 から 20 までの乱数で決める。

(3) (1) と (2) で決った条件に合わせて配列中のデータを生成する。この時、正負のデータがほぼ均等に現れるようにする。

(4) (3) で生成されたデータに対して関数 checksort を呼び出して、正しく整列されるかどうかをチェックする。

3 演習問題

以下の 3 つの演習問題に取り組みなさい。プログラムが完成したら、前回と同様に mprog2 コマンドを実行して、自分が作成したプログラムを提出してください。

3.1 prog4-1.c

前回作成した prog3-2.c を prog4-1.c にコピーし、「1.1 実力チェック」で行ったのと同様の変更をこのプログラム prog4-1.c に加え、実際に計算機上で作成、コンパイル、実行して、次の実行例のように動作することを確認しなさい。

prog4-1.c の実行例

```
s1542h017% ./prog4-1
 1番目のデータ => 27
 2番目のデータ => 18
 3番目のデータ => 43
 4番目のデータ => 7
 5番目のデータ => 67
 6番目のデータ => 72
 7番目のデータ => 89
 8番目のデータ => .
 27, 18, 43,  7, 67, 72, 89
 18, 27,  7, 43, 67, 72, 89
 18,  7, 27, 43, 67, 72, 89
  7, 18, 27, 43, 67, 72, 89
s1542h017%
```

注意: このプログラムは mprog2 コマンドで提出すると、教員か TA のチェックを受けるように要求されます。mprog2 コマンドが表示する指示に従ってください。

3.2 prog4-2.c

まず、prog4-1.c を、prog4-2.c にコピーしなさい。次に、 2 ページの「実現の方針」での last が変化していく過程が次の実行例のように表示されるように prog4-2.c を変更しなさい。

prog4-2.c の実行例

```
s1542h017% ./prog4-2
 1番目のデータ => 27
 2番目のデータ => 18
 3番目のデータ => 43
 4番目のデータ => 7
 5番目のデータ => 67
 6番目のデータ => 72
 7番目のデータ => 89
 8番目のデータ => .
 27, 18, 43,  7, 67, 72, 89
                        ^last
 18, 27,  7, 43, 67, 72, 89
         ^last
 18,  7, 27, 43, 67, 72, 89
     ^last
  7, 18, 27, 43, 67, 72, 89
s1542h017%
```

第 3 回の付録「prog2-5.c のプログラム例」や、今回の付録 C 「prog3-4.c のプログラム例」が参考になるはずです。

3.3 prog4-3.c

まず、prog4-1.c を prog4-3.c にコピーし、今回「2 ソーティングプログラムのテスト」の節で解説したテスト用の関数群 (checkperm、checkorder、checksort) の定義を書き加えるとともに、関数 main の定義を 5 ページのもので置き換え、自分の作った関数 bubblesort をテストするプログラムに変更しなさい。

prog4-3.c の実行例

```
s1542h017% ./prog4-3
すべてのチェックに合格しました。
s1542h017%
```

プログラミングおよび実習Ⅱ・第 4 回・終り

付録 A. typedef による型定義

C の予約語 typedef を使うと、自分で型に名前を付けることができます。typedef による型定義は

```
typedef 型指定 新しい型名;
```

のような書式で書きます。たとえば、

```
typedef int Boolean;
```

と型定義をしておくと、これ以降 Boolean という語を int と同じ意味で使うことができます。つまり「int x;」と書いて変数 x を宣言する代りに、「Boolean x;」と書くことができるようになります。int の代りに、Boolean を定義して用いることで、その変数などで扱われるデータが真理値であることが明確になり、プログラムが理解しやすくなるとともに、間違いを減らすことができます。

このような用途以外にも、たとえば、構造体を使ったプログラムに何度も「struct 構造体タグ」という型指定が現れて煩わしい場合に、

```
struct DATE {
    int year;
    int month;
    int day;
};

typedef struct DATE Date;
```

のように書いておき、これ以降 Date を struct DATE という構造体型を意味する型名として利用することもできます。構造体の宣言と一緒にして、

```
typedef struct DATE {
    int year;
    int month;
    int day;
} Date;
```

のようにすることも可能です。構造体タグの DATE は特に必要がなければ省略することもできます。

この typedef による型定義は、typedef を無視すると Date という名前の変数の宣言のように見えることに注意してください。そのように見たときの Date という変数の型が、typedef で定義される型名 Date の意味するデータ型となります。たとえば、

```
typedef struct DATE History[100];
```

と書くと、History は「100 個の struct DATE 型の要素からなる配列型」を意味する型名となります。つまり、この型定義以降で、たとえば

```
History h;
```

のように h を宣言すると、

```
struct DATE h[100];
```

と宣言したのと等価になります。

付録 B. 乱数の発生

ソーティングプログラムのテストのためのデータ生成以外にも、シミュレーションやゲームなどのプログラミングで、**乱数**と呼ばれるでたらめな数が必要となることがよくあります。C では rand という関数を使うことで、このでたらめな数を生成することができます。rand 関数は、それが呼ばれるたびに、0 以上のでたらめな int 型の整数[4]を返します。ただし、関数 rand が生成する整数は、見かけは乱数のように見えますが、実はある規則にしたがって順に決った整数が生成されているに過ぎません。プログラムを実行するたびに rand が生成する乱数の系列を変えたい時には、srand という関数を使って、プログラムが実行される度に変化するような数を**乱数の種**として設定します。rand が生成する乱数列は、この srand によって設定された整数 (乱数の種) によって変りますので、プログラムが実行される度に生成される乱数の系列も変ってくれるようになります。srand の引数の型は unsigned int 型です。プログラム中で関数 rand や srand を使う場合は「#include <stdlib.h>」が必要となります。

次のプログラムは、現在時刻を乱数の種として、0 〜 9 までの数を 100 個表示するプログラムです。関数 time は NULL[5]を引数とて呼び出すと、ある特定の日時 (たとえば、グリニッジ標準時で 1970 年 1 月 1 日 0 時 0 分 0 秒) から経過した秒数を、ある整数型[6]として返してくれますので、これを乱数の種と

[4] stdlib.h では、rand によって生成される乱数の最大値が RAND_MAX というマクロとして定義されています。

[5] NULL は stdio.h、stdlib.h、time.h などで定義されているマクロで、何も指さない (どの場所でもない) ことを表すポインタ型の値 0 に定義されています

[6] time.h の中で、 time_t という名前の型が unsigned int 型や unsigned long int 型として定義されています。

して srand に渡しています。プログラム中で関数 time を使う場合は「#include <time.h>」が必要です。

実行する度に乱数を 100 個出力するプログラム

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <time.h>
 4
 5 int main ()
 6 {
 7     int i;
 8
 9     srand(time(NULL));
10     for (i = 0; i < 100; i ++)
11         printf ("%d\n", rand()%10);
12     return EXIT_SUCCESS;
13 }
```

付録 C. 前回の演習問題 prog3-4.c のプログラム例

prog3-4.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 typedef int Boolean;
 5 #define TRUE            (1)
 6 #define FALSE           (0)
 7
 8 #define MAX_DATA        (1000)
 9
10 void showdata (int data[], int num)
11 {
12     int i;
13
14     if (num > 0) {
15         printf("%3d", data[0]);
16         for (i = 1; i < num; i++)
17             printf(",%3d", data[i]);
18     }
19     printf("\n");
20 }
21
22 void showchanged (Boolean changed[], int num)
23 {
24     int i;
25
26     if (num > 0) {
27         for (i = 0; i < num; i++) {
28             if (changed[i])
29                 printf("^^^^");
30             else
31                 printf("    ");
32         }
33     }
34     printf("\n");
35 }
36
37 void bubblesort(int data[], int num)
```

```
38 {
39     Boolean changed[MAX_DATA];
40     int start, i;
41     int tmp;
42
43     for (start = 0; start < num-1; start++) {
44         showdata(data, num);
45         for (i = 0; i < num; i++)
46             changed[i] = FALSE;
47         for (i = num-1; start < i; i--) {
48             if (data[i-1] > data[i]) {
49                 tmp = data[i];
50                 data[i] = data[i-1];
51                 data[i-1] = tmp;
52                 changed[i-1] = changed[i] = TRUE;
53             }
54         }
55         showchanged(changed, num);
56     }
57 }
58
59 int main ()
60 {
61     int data[MAX_DATA];
62     int num;
63
64     for (num = 0; num < MAX_DATA; num++) {
65         printf("%2d番目のデータ => ", num+1);
66         if (scanf ("%d", &data[num]) != 1)
67             break;
68     }
69
70     bubblesort(data, num);
71     showdata(data, num);
72     return EXIT_SUCCESS;
73 }
```