プログラミングおよび実習II・第 9 回

2003 年 11 月 19 日

1 実力チェック

次の関数 selectsort は、int 型の配列 d の先頭から num 個の要素を選択ソート法で昇順に並び替えるものです。

```
void selectsort(int d[], int num)
{
    int start, pos, i;
    int min, tmp;

    for (start = 0; start < num-1; start++) {
        min = d[start];
        pos = start;
        for (i = start+1; i < num; i++) {
            if (d[i] < min) {
                min = d[i];
                pos = i;
            }
        }
        if (pos != start) {
            tmp = d[start];
            d[start] = d[pos];
            d[pos] = tmp;
        }
    }
}
```

この関数の定義を書き換え、Record 型へのポインタ型の配列 d と、要素の個数 num (int 型) を引数として受け取ると、配列 d の要素を、それが指す構造体の score の部分 (メンバ) が昇順になるように (同じ選択ソート法のアルゴリズムで) 並び替える関数にしなさい。ただし、Record 型の定義は「付録 B」(9 ページ参照) の prog8-1.c と同じであるとします。解答用紙のプログラム名は「prog9-1.c」とし、関数 selectsort の (変更後の) 定義のみを書きなさい。

2 一連の実験を自動化する

前回作成した prog8-2 のコマンドライン引数 (要素数) をいろいろと変えて、整列に要する時間がどうなるか調べてみます。

```
s1542h017% ./prog8-2 10
10個の要素の整列に要した時間 (15.550 - 5.770) / 8356128 = 0.0000011704 秒
s1542h017% ./prog8-2 100
100個の要素の整列に要した時間 (11.910 - 1.810) / 601463 = 0.0000167924 秒
s1542h017% ./prog8-2 1000
1000個の要素の整列に要した時間 (11.240 - 1.280) / 47456 = 0.0002098786 秒
s1542h017% ./prog8-2 10000
10000個の要素の整列に要した時間 (11.370 - 1.400) / 2458 = 0.0040561432 秒
s1542h017% ./prog8-2 100000
100000個の要素の整列に要した時間 (11.530 - 1.570) / 119 = 0.0836974790 秒
```

```
s1542h017%
```

この例のように、人間が 1 回の実験ごとにプログラムを起動してその終了を待ち、出力された結果を書き留めて行くこともできますが、毎回プログラムの終了を待ってから、次の引数で起動するのもわずらわしいですし、出力結果を書き留める際に読み間違いや書き間違いが起る可能性もあります。そこで、このような一連の実験を自動化してみましょう。

バッチファイル　Unix (Linux) や Windows などの OS には、実行したい一連のコマンド (プログラム) とコマンドライン引数をファイルに記述しておくと、それを次々と実行してくれるような仕組が用意されています。このようなファイルを一般に**バッチファイル**と呼びます。たとえば Unix (Linux) の場合、gedit や emacs、vi などのエディタを使って、適当なファイル (たとえば batch8-2) に

batch8-2 の内容

```
./prog8-2 10
./prog8-2 100
./prog8-2 1000
./prog8-2 10000
./prog8-2 100000
```

の 5 行を書いておき、chmod コマンドで

batch8-2 の保護モードを変更する

```
s1542h017% chmod a+x batch8-2
```

のように、このファイルをだれでも実行可能な状態にします。こうしておくことで、この batch8-2 というバッチファイルを (他のコマンドやプログラムを起動するときと同じ方法で) 起動することができるようになります。batch8-2 が起動されると、その中に書かれているコマンド (プログラム) が書かれている順に実行されて行きます。次は batch8-2 の実行例です。

```
s1542h017% ./batch8-2
10個の要素の整列に要した時間 (15.550 - 5.770) / 8356128 = 0.0000011704 秒
100個の要素の整列に要した時間 (11.910 - 1.810) / 601463 = 0.0000167924 秒
1000個の要素の整列に要した時間 (11.240 - 1.280) / 47456 = 0.0002098786 秒
10000個の要素の整列に要した時間 (11.370 - 1.400) / 2458 = 0.0040561432 秒
100000個の要素の整列に要した時間 (11.530 - 1.570) / 119 = 0.0836974790 秒
s1542h017%
```

一旦、この batch8-2 を起動すれば、あとはすべての実験が終るのを待っているだけです。

出力のリダイレクション　シェル[1]のリダイレクション機能を使って、標準出力をファイルにリダイレクトすると、実験結果を書き留める必要もなくなります。たとえば

```
s1542h017% ./batch8-2 >batch8-2.log
```

[1]「シェル」とは、Unix (Linux) において、キーボードから入力されたコマンド名やコマンドライン引数の文字列などを読み取って、指定されたプログラムを指定された引数で起動するという仕事をするプログラムの総称です。「コマンドラインインタープリタ」ということもあります。

のように、「>ファイル名」を付けてコマンド (プログラムやバッチファイル) を起動すると[2]、本来なら標準出力 (端末エミュレータのウィンドウ) に表示されていた実験結果が、そこに表示される代りに、指定されたファイルに書き込まれていきます。そのファイルが存在しない場合は、自動的にそのファイルが作成されますし、すでにそのファイルが存在している場合は、古い内容をすべて消去した後に、起動したコマンド (プログラムやバッチファイル) の出力が書き込まれます。ファイルの元の内容の末尾からコマンドの出力を書き足したい場合は

```
s1542h017% ./batch8-2 >>batch8-2.log
```

のように「>」の代りに「>>」を使います[3]。

パイプと tee コマンド　コマンド (プログラムやバッチファイル) の出力は、シェルのパイプ機能を使って、他のプログラムの標準入力に接続することもできます。たとえば、tee というコマンドは、標準入力から読み取った内容を標準出力に書き出しながら、同じ内容をコマンドライン引数で指定されたファイルに書き込んでいくものですが、一連の実験を行うバッチファイルの出力をパイプ機能を使って tee へ送ることで、実験結果を端末エミュレータのウィンドウで確認しながらファイルに保存していくこともできます。つぎの実行例では、この方法を使って batch8-2 が出力した実験結果を batch8-2.log というファイルに保存しています。

```
s1542h017% ./batch8-2 | tee batch8-2.log
10個の要素の整列に要した時間 (15.550 - 5.770) / 8356128 = 0.0000011704 秒
100個の要素の整列に要した時間 (11.910 - 1.810) / 601463 = 0.0000167924 秒
1000個の要素の整列に要した時間 (11.240 - 1.280) / 47456 = 0.0002098786 秒
10000個の要素の整列に要した時間 (11.370 - 1.400) / 2458 = 0.0040561432 秒
100000個の要素の整列に要した時間 (11.530 - 1.570) / 119 = 0.0836974790 秒
s1542h017%
```

端末エミュレータのウィンドウに表示された内容 (5 行) が、batch8-2.log というファイルに書き込まれているはずです。

3　演習問題

以下の 3 つの演習問題に取り組みなさい。prog9-1.c と prog9-2.c については、自分が作成したプログラムを mprog2 コマンドで提出してください。prog9-1.c については教員か TA のチェックも必要となります。batch8-2 と batch9-1、batch9-2.c については提出の必要はありません。

[2] > (や >>、<) の両側の空白はあってもなくても構いません。

[3] シェルのリダイレクション機能を使うと、起動されたコマンド (プログラム) の標準入力 (通常はキーボード) を、指定したファイルから読み取るように変更することもできます。この場合「コマンド名 <ファイル名」のように「<」を使って標準入力となるファイルを指定します。

3.1 prog9-1.c

まず、前回の演習問題で作成したプログラム prog8-2.c を prog9-1.c にコピーしなさい。つぎに、このプログラムの関数 quicksort の定義の部分を「1. 実力チェック」で作成した関数 selectsort で置き換え、また、quicksort を呼び出していた部分を、代りに selectsort を呼び出すように変更して、選択ソート法での整列時間を測定するプログラムに変更しなさい。

3.2 prog9-2.c

同様に、まず、前回の prog8-2.c を prog9-2.c にコピーしなさい。つぎに、関数 quicksort の定義や呼び出しを第 4 回の「1.1 実力チェック」の方針で効率化したバブルソートを行う関数 bubblesort[4]を定義して置き換え、(効率化した) バブルソート法での整列時間を測定するプログラムに変更しなさい。ただし、prog9-2.c の関数 bubblesort は、Record型へのポインタ型の配列を整列することに注意すること。

3.3 batch8-2、 batch9-1、 batch9-2

前回のプログラム prog8-2 を、 10、 100、 1000、 10000、 100000 の 5 通りのコマンドライン引数で起動するバッチファイル batch8-2 を作りなさい。また、 10、 100、 1000、 10000 の 4 通りの引数で prog9-1 を起動するバッチファイル batch9-1 と、prog9-2 を起動するバッチファイル batch9-2 をそれぞれ作りなさい。バッチファイルが完成したら、それぞれを実行して得られた結果を、シェルのリダイレクションなどの機能を使って batch8-2.log、batch8-3.log、batch8-4.log に保存しなさい。この課題は mprog2 コマンドで提出する必要はありません。

プログラミングおよび実習Ⅱ・第 9 回・終り

[4]第 5 回の 1 ページに int 型の配列を昇順に整列させるプログラムがありますので、これを参照してください。

付録 A. C における文字列の取り扱い

文字コードの配列としての文字列

C プログラムでは、通常、char 型の整数 (文字コード) の 1 次元配列を使って文字列を表現します。たとえば「Test」という 4 文字からなる文字列の場合、char 型の配列を用意し、その先頭から「T」、「e」、「s」、「t」の 4 文字の文字コードを順に格納します。ただし、扱う文字列の長さが常に 4 文字であるとは限りませんので、その文字列が何 byte (何文字) の文字列であるのか関する情報が必要となります。この文字列の長さに関する情報は int 型の変数などに (配列とは別に) 記憶しておくこともできますが、通常 C 言語では (そうする代りに) 文字コードの配列の中に、文字列の終端を表す特殊な文字コード 0 [5]を置いて、そこで文字列が終っていることを表します。

「Test」という文字列が、char 型の配列 s に格納されている様子は、ちょうど次のようなプログラムが実行されたときの状況と同じになります。

```
char s[10];

s[0] = 'T';
s[1] = 'e';
s[2] = 's';
s[3] = 't';
s[4] = '\0';
```

文字列の終端を表す 0 が置かれますので、 n 文字 (n byte) からなる文字列を格納するためには、配列の長さ (要素数) は少なくとも $n+1$ でなければならないことに注意が必要です。また、 0 が格納されている要素より後の配列要素は、その文字列の表現には関わっていないことにも注意してください。

アドレスとしての文字列

C 言語では、文字列は配列中に格納された文字コードの列 (終端を表す 0 を含む) として表現されますが、配列はそれ全体を代入したり、引数として全体の値を渡したりすることができないため、通常その文字列を格納している配列の先頭要素のアドレスとして文字列を扱います。次のプログラムは、仮引数 str の指す文字列の長さ (終端の 0 は数えない) を返す関数 stringLength と、仮引数 src の指す文字列を、dst の指す配列[6]に (終端の 0 を含めて) コピーする関数 copyString を定義したものです。関数 copyString は、第 1 仮引数 dst に渡される配列が十分な長さを持っていることを前提としたプログラムとなっています。

```
int stringLength(char str[])
{
    int n = 0;

    while (str[n] != '\0')
        n++;
    return n;
}
```

[5]この 0 は整数値の 0 であって、「0」という数字の文字コードではありません。 0 は ASCII コードの制御文字 NUL の文字コードですが、これを文字列の終端を表すために用いますので、 NUL 文字を含むような文字列をこの方法で表現することはできません

[6]正確には、dst はこの配列の先頭要素を指します。仮引数 dst に渡ってくる実引数の値は「char 型の要素からなる配列の先頭要素のアドレス」です。

```
void copyString(char dst[], char src[])
{
    int i = 0;

    while ((dst[i] = src[i]) != '\0')
        i++;
}
```

これらの関数の仮引数は、すべて char 型要素からなる配列型ですが、C では、関数定義の仮引数がある型の要素からなる配列型であった場合、その型へのポインタ型としてその仮引数が宣言されたものと扱われます。つまり、この 2 つの関数定義での仮引数は、それぞれ

```
int stringLength(char *str)
void copyString(char *dst, char *src)
```

のように宣言しても同じ意味になります。

C 言語の**配列添字演算子** [ ] は、(配列名を含めて) ある型へのポインタ型の値を持つ一般の式 p に対して (その後に書いて) 適用することができ、p[i] は、p が指すデータの i 個先に格納されているデータを表します。p がある型へのポインタ型の値を、i が整数型 (int 型など) の値を持っている場合、式 p+i は、p が指すデータの i 個先に格納されているデータを表しますので、p[i] と *(p+i) はまったく同じ意味となります。

a が配列名であるとき、その添字 i の要素を a[i] で表すことができるのは、配列名 a はその先頭要素のアドレス (配列要素の型へのポインタ型の値) を表す定数式と見なされるためです。C では、ある型を要素とする配列型は、ある型へのポインタ型と見なして扱うことができます。

文字列リテラル

C のプログラムの中では、ある文字の文字コードを、その文字を「'」(1 重引用符) で囲むことで表すことができますが、同様に、ある文字列を表現する文字コードの列とその終端を表す 0 を、その文字列を「"」(2 重引用符) で囲むことで表現できます。プログラム中にこのように記された文字列データを**文字列リテラル**[7]と呼びます。たとえば、C のプログラム中で "Test" と書くと、それは、その要素が先頭から 'T'、'e'、's'、't'、0 となっている長さ (要素数) 5 の char 型配列[8]の先頭要素のアドレス[9]を表し、

```
int i;

for (i = 0; i < 4; i++)
    printf ("%c", "Test"[i]);
printf ("\n");
```

というプログラムは、結局

```
printf ("Test\n");
```

[7] **文字列定数**とよぶこともあります

[8] ただし、この配列の要素の値を書き換えることは許されていません (書き換えた場合にどうなるかは予想できません)

[9] 文字列リテラルのデータ型は char 型へのポインタ型になります

と同じ出力を行います。文字列リテラル "Test" が、あたかも初期化された長さ 5 の (読み出し専用の) char 型配列のように振る舞っていることに注意してください。

文字定数の場合と同様に、\t や \n のように書くことで、制御文字などを文字列リテラル中に含めることができます。「"」や「\」(バックスラッシュ) 自身を文字列リテラル中に含めたい場合は、それぞれ \" や \\ と書きます。

文字列の初期化

C では、文字列を 1 次元配列に格納された文字コードの列 (と終端を表す 0) として扱うことを説明しましたが、一般の 1 次元配列について、次のような形式でその宣言と各要素の初期化を同時に行うことができます。

要素の型 配列名[要素数] = { 式$_0$, 式$_1$, 式$_2$, ... 式$_n$ };

このように配列を宣言すると、この配列のためのメモリが確保されると同時に、添字 i の要素が 式$_i$ でそれぞれ初期化されます。n は「要素数」未満でなければなりません。n が 「要素数−1」に満たない場合は、この配列の添字 n+1 以降の要素は 0 で初期化されます[10]。また、初期化を伴う配列の宣言の場合「要素数」は省略することもできます。「要素数」を省略すると、n+1 (初期値が明示されている要素の個数) を要素数とする配列が確保されます。

文字列コードの列を格納する char 型の配列の場合、文字列リテラルを使って、

```
char message[] = "Hello!";
```

のように配列の各要素を初期化することもできます[11]。このとき、文字列の終端を表す 0 を含めて配列要素が初期化されます[12]。この例の場合、長さ (要素数)7 の配列 message が確保され、その先頭要素から順に 'H'、'e'、'l'、'l'、'o'、'!'、 0 が格納されます。つまり、次のような宣言と等価です。

```
char message[7] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

printf による文字列の出力

文字列を関数 printf で出力する場合は、printf の出力書式文字列の中に %s という**変換仕様**を置き、その位置に埋め込みたい文字列を、対応する実引数として printf に渡します。

たとえば、次のプログラムの 4 つの printf の呼び出しは、まったく同じ文字列を出力します。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char a[] = "Hello!";
7     char *p = "Hello!";
8
```

[10] 要素の型に応じて、整数型や浮動小数点型、ポインタ型 (アドレス) の 0 で初期化されます

[11] 文字列リテラルを {} で囲って「char message[] = { "Hello!" };」のように書いても同じ意味になります

[12] 「要素数」が省略された場合には、終端の 0 を格納するための要素も含めて配列の長さが決定されます

```
 9     printf ("Hello!\n");
10     printf ("%s\n", a);
11     printf ("%s\n", p);
12     printf ("%s\n", "Hello!");
13     return EXIT_SUCCESS;
14 }
```

このプログラムの 10 行目と 11 行目を見ると、6 行目と 7 行目の 2 つの宣言に違いがないように見えますが、実はそうではないことに注意が必要です。6 行目では、長さ (要素数)7 の配列が確保され、その各要素が初期化されます。a は配列名であり、配列型の定数ですから、その要素の値を変更することはできても、a に別の配列を指させることはできません。一方、7 行目では、char 型データのアドレスを格納する変数 p が用意され、これとは別に確保された長さ 7 の (読み出し専用の) char 型配列の先頭要素のアドレスが、この変数 p に初期値として代入されます。p は char 型へのポインタ型のデータを格納する変数ですから、その値は代入によっていつでも変更することができます。ただし、その初期値は文字列リテラルによって作られた読み出し専用の文字列を指していますので、この文字列の内容 (配列の各要素の値) を変更することはできません[13]。

scanf による文字列の入力

関数 scanf の入力書式文字列の中に %s という変換仕様を書くと、空白類 (スペースやタブ、改行文字など) を区切りとした 1 つの文字列を入力し、char 型の配列に格納することができます。変換仕様 %s に対応する実引数は char 型へのポインタ型 (char 型のデータの記憶領域のアドレス) でなければなりません。

ただし、入力される文字列の長さは一般に予想できませんので、その文字列を格納する配列もどれだけの長さのものを準備しておけばよいか分かりません。もし、入力された文字列が、プログラムが用意した配列の長さを越えてしまう[14]と、scanf は配列のために確保された領域を越えてメモリの内容を書き換えてしまいますので、実行中のプログラムの動作は予想できないものとなってしまいます。このため scanf の変換仕様 %s を使った文字列入力はあまりお勧めできません[15]。

getchar による文字の読み込み

次の関数 getLine は、キーボードから入力された 1 行分の文字列を、長さが size の char 型配列 buf に格納し 0 で終端します。ただし、最後の改行文字は格納されません。また、入力された行の文字数 (byte 数) が size−1 を越えた場合は、越えた分の文字列は読み飛ばされます (配列には格納されません)。関数 getLine は buf に格納した文字数 (byte 数) を返り値として返します。

```
#include <stdio.h>

int getLine(char buf[], int size)
```

[13] そうした場合の動作は予想がつかないものとなります

[14] これを Buffer Overflow (あるいは Buffer Overrun) と呼びます。悪意を持ったユーザーが Buffer Overflow を利用してプログラムの誤動作を引き起すことで、情報システムの乗っ取りが可能になってしまうことがよくあります。

[15] 関数 scanf では、変換仕様 %s の % と s の間に正の整数を挟み、配列に格納される最大の文字数 (byte 数) を指定することができます。この指定を行うことで Buffer Overflow を防止することは可能です。

```
{
    int ch;
    int n = 0;

    /* 文字が読み込めなかったり、改行文字の場合は繰り返しをやめる */
    while ((ch = getchar()) != EOF && ch != '\n') {
        /* 配列 buf にまだ余裕があれば、読み込んだ文字を格納する */
        if (n < size - 1)
            buf[n++] = ch;
    }
    /* buf を NUL 文字で終端する */
    buf[n] = '\0';
    return n;
}
```

この関数 getLine は、標準入出力ライブラリの getchar という関数[16]を呼び出して、標準入力 (キーボード) から 1 文字を読み込んでいます。getchar は読み込んだ文字の文字コードを int 型の返り値として返します。読み込みができなかった時には、定数 EOF [17]を返します。getchar の返り値を格納している変数 ch が char 型でなく int 型になっているのは、正常に読み込まれた 1byte の文字コードの値と、読み込みができなかったことを表す定数 EOF (-1) を区別するためには char 型では大きさが不十分だからです。

付録 B. 前回の演習問題のプログラム例

prog8-1.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 NAME_LEN        (100)
 9 #define MAX_DATA        (1000)
10
11 typedef struct {
12     char name[NAME_LEN];
13     int score;
14 } Record;
15
16 void showdata(Record *d[], int num)
17 {
18     int i;
19
20     for (i = 0; i < num; i++)
21         printf("%3d %s\n", d[i]->score, d[i]->name);
22 }
23
24 void quicksort(Record *d[], int h, int t)
25 {
26     Record *tmp;
27     char *r;
28     int i, j;
29
```

[16]本当の関数ではなく、関数形式のマクロとして stdio.h で定義されている場合もあります

[17]ヘッダファイル stdio.h で -1 にマクロ定義されています

```
30      if (h >= t)
31          return;
32
33      r = d[(h+t)/2]->name;
34      i = h;
35      j = t;
36      while (1) {
37          while (strcmp(d[i]->name, r) < 0)
38              i++;
39          while (strcmp(r, d[j]->name) < 0)
40              j--;
41          if (j <= i)
42              break;
43          tmp = d[i];
44          d[i++] = d[j];
45          d[j--] = tmp;
46      }
47      quicksort(d, h, i-1);
48      quicksort(d, j+1, t);
49  }
50
51  int main()
52  {
53      Record data[MAX_DATA], *dp[MAX_DATA];
54      int num;
55
56      for (num = 0; num < MAX_DATA; num++) {
57          printf("%2d番目の点数と名前 => ", num+1);
58          if (scanf("%d %99[^\n]", &data[num].score, data[num].name) != 2)
59              break;
60          dp[num] = &data[num];
61      }
62      quicksort(dp, 0, num-1);
63      showdata(dp, num);
64      return EXIT_SUCCESS;
65  }
```

prog8-2.c の関数 checktime と main の定義

```
 1  #define MIN_SEC         (10.0)
 2  #define INIT_SEC        (3.0)
 3
 4  typedef struct {
 5      int repeat;         /* 整列を繰り返した回数 */
 6      double t1;          /* データの初期化に要した時間(秒) */
 7      double t2;          /* データの初期化と整列に要した時間(秒) */
 8  } ExpResult;
 9
10  ExpResult checktime(int num, double sec)
11  {
12      static Record data[MAX_DATA], *dp[MAX_DATA];
13      ExpResult r;
14      int i, j;
15
16      /* 名前の部分を初期化 */
17      for (i = 0; i < num; i++)
18          strcpy(data[i].name, "Test");
19
20      /* 先にデータ生成と整列、clock_time に必要な時間を求める */
21      clock_reset();
22      clock_on();
```

```
23      r.repeat = 0;
24      do {
25          for (i = 0; i < num; i++)
26              data[i].score = rand();
27          for (i = 0; i < num; i++)
28              dp[i] = &data[i];
29          quicksort(dp, 0, num-1);
30          r.repeat++;
31      } while (clock_time() < sec);
32      r.t2 = clock_off();
33
34      /* データ生成と clock_time の呼び出しだけに必要な時間を求める */
35      clock_reset();
36      clock_on();
37      for (j = 0; j < r.repeat; j++) {
38          for (i = 0; i < num; i++)
39              data[i].score = rand();
40          clock_time();
41      }
42      r.t1 = clock_off();
43      return r;
44  }
45
46  int main(int argc, char *argv[])
47  {
48      int num;
49      ExpResult r;
50
51      if (argc != 2) {
52          printf("要素数が指定されていません。\n");
53          exit(EXIT_FAILURE);
54      }
55
56      if (sscanf(argv[1], "%d", &num) != 1
57              || num <= 0 || num > MAX_DATA) {
58          printf("要素数の指定が正しくありません。\n");
59          exit(EXIT_FAILURE);
60      }
61
62      /* 試験的に INIT_SEC 秒間計測 */
63      r = checktime(num, INIT_SEC);
64      /* 純粋な整列時間の累計が MIN_SEC 以上になるように計測 */
65      r = checktime(num, MIN_SEC*r.t2/(r.t2-r.t1));
66      /* 結果を出力する */
67      printf("%d個の要素の整列に要した時間 "
68             "(%.3f - %.3f) / %d = %.10f 秒\n",
69             num, r.t2, r.t1, r.repeat, (r.t2 - r.t1) / r.repeat);
70      return EXIT_SUCCESS;
71  }
```