プログラミングおよび実習 I・第 5 回

2003 年 5 月 7 日

1 C における文字列の取り扱い

1.1 文字コードの配列としての文字列

プログラム中で文字列を扱いたい場合、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するのが簡単です。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 [1]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。

1.2 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[2]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')
```

[1]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

[2]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。

```
        n++;
    return n;
}

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) を表す定数式でもあるためです。このように 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。

1.3 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したが、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**[3]と呼びます。たとえば、C のプログラム中で "Test" と書くと、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[4]の先頭要素のアドレス[5]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");
```

[3] **文字列定数**とよぶこともあります

[4] 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)

[5] 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。

1.4 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[6]。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[7]。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[8]。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' };
```

1.5 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
3 int main()
4 {
5     char a[] = "Hello!";
6     char *p = "Hello!";
7
8     printf ("Hello!\n");
```

[6] 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

[7] 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

[8] 「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 ("%s\n", a);
10     printf ("%s\n", p);
11     printf ("%s\n", "Hello!");
12     return 0;
13 }
```

このプログラムの 9 行目と 10 行目を見ると、5 行目と 6 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。5 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、6 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[9]。

1.6 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[10]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[11]。

2 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)
```

[9] そうした場合の動作は予想がつかないものとなります

[10] 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。

[11] 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[12]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 [13]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。

3 関数 getLine をオセロゲームに応用する

次のプログラム othello12.c は、前回の演習問題で作成した othello11.c (5 – 9 ページの付録を参照) の関数 inputMove の定義を変更して、関数 getLine で入力を行うようなプログラムとしたものです。inputMove の関数定義以外の部分は省略されています。inputMove の定義で変更のあった行には * 印がつけてあります。

新しい inputMove では、まず、 getLine でキーボードからの 1 行分の文字列を配列 buf に格納しています (プログラム 80 行目)。この時、 C の sizeof 演算子[14]を使って、配列 buf の大きさを計算し、これを getLine の第 2 引数としています[15]。getLine によって、キーボードから入力された 1 行分の文字列が buf に格納されますが、そこから列番号と行番号の 2 文字を標準入出力ライブラリ関数 sscanf を使って取り出しています。sscanf は、scanf と同様の機能を持った関数です。sscanf はキーボードからの入力を調べる代りに、第 1 引数で指定された文字列の内容を調べてくれます。sscanf や scanf は、正常にデータを読み込むことのできた変換仕様 (%c や %d など) の数を int 型の返り値として返しますので[16]、othello12.c ではこの返り値を調べて (空白類を無視して) ちょうど 2 つの文字が読み込めたかどうかをチェックしています。

---

[12]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

[13]ヘッダファイル stdio.h で -1 にマクロ定義されています

[14]sizeof 演算子を変数や配列、式、( ) で囲まれた型指定に対して適用すると、その型のデータを記憶するために必要なメモリの大きさ (何 byte 必要か) を符号なし整数の値として返してくれます

[15]関数 getLine の第 2 引数は配列 (第 1 引数) の要素数ですから、本来ならここは sizeof buf/sizeof(char) あるいは sizeof buf/sizeof buf[0] とすべきですが、sizeof char (sizeof buf[0]) は 1 (byte) であることが分かっていますので、単に sizeof buf としています

[16]文字が読み込めなかった場合は EOF を返します

othello12.c の inputMove

```
 70 /* ユーザーが指定した位置に石を置く */
 71 void inputMove(int b[][BOARD_SIZE], int turn)
 72 {
*73     char buf[100], ch1, ch2, ch3;
 74     int x, y;
 75
 76     /* 位置を入力する */
 77     while (1) {
 78         printf("位置 = ");
 79         /* キーボードからの1行分の入力を buf に格納する */
*80         if (getLine(buf, sizeof buf) < 1)
*81             continue;
 82         /* buf から空白類を無視して、最大3文字までを抜き出す。
 83            2文字の入力でないなら再入力を促す */
*84         if (sscanf(buf, " %c %c %c", &ch1, &ch2, &ch3) != 2)
*85             continue;
 86         /* とりあえず c5 のような形式の入力と仮定してみる */
 87         x = ch1 - 'a';
 88         y = ch2 - '1';
 89         if (0 <= x && x < BOARD_SIZE && 0 <= y && y < BOARD_SIZE
 90                 && b[x][y] == NONE) {
 91             break;
 92         }
 93         /* 逆に 5c のような形式の入力と仮定してみる */
 94         x = ch2 - 'a';
 95         y = ch1 - '1';
 96         if (0 <= x && x < BOARD_SIZE && 0 <= y && y < BOARD_SIZE
 97                 && b[x][y] == NONE) {
 98             break;
 99         }
100         printf("そこには置けません\n");
101     }
102     /* 石を置く */
103     b[x][y] = turn;
104 }
```

このプログラムの 89 行目や 96 行目から始まる if 文を、

```
if (b[x][y] == NONE
        && 0 <= x && x < BOARD_SIZE && 0 <= y && y < BOARD_SIZE) {
    break;
}
```

のようにしてはいけません[17]。こうすると、変数 x や y の値が配列 b の添字の範囲に収まっていること確認する前に b[x][y] を参照してしまうことになります。配列要素に代入を行ったり、その参照を行う場合には、添字の範囲が妥当かどうかを常に意識するようにしてください。

このプログラムをコンパイルして実行すると次のようになります。列番号や行番号だけで Enter キーが押された時など、不完全な入力がされた場合には、もう一度プロンプトを表示して、始めから入力し直すようにしていることに注目してください。

othello12 の実行例

```
s1609h017% ./othello12
 a b c d e f g h
```

[17]付録 (5 – 9 ページ) のプログラム othello11.c の 64 行目や 71 行目についても同じです

```
1□□□□□□□□
2□□□□□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = 2
位置 = 2d
 a b c d e f g h
1□□□□□□□□
2□□□●□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = g
位置 = g55
位置 = g5
 a b c d e f g h
1□□□□□□□□
2□□□●□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□○□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = 9e
そこには置けません
位置 = 8e
 a b c d e f g h
1□□□□□□□□
2□□□●□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□○□
6□□□□□□□□
7□□□□□□□□
8□□□□●□□□
位置 = ^C
s1609h017%
```

4 演習問題

4.1 例題プログラムを試す

講義で解説した `othello12.c` というプログラムを自分で作り[18]、コンパイル、実行してみなさい。5 – 4 ページの関数 `getLine` の定義を `othello12.c` に書き加えるのを忘れないようにしましょう。完成したら、「`mprog1 othello12.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。次の演習問題も同様です。

[18] 前回作成した `othello11.c` をコピーして変更すると簡単かも知れません

4.2 q が入力された場合はプログラムを終了する

まず、othello12.c を othello13.c にコピーしなさい。その後 othello13.c を変更して、次の実行例のように、q という 1 文字が入力されたらプログラムを終了するようにしなさい。ただし、次のような方針にしたがってこれを実現しなさい。

1. 関数 inputMove の返り値の型を int 型に変更し、キーボードから石の位置が入力され (その位置に石を置い) た時は 0 を、q という 1 文字が入力された時は 1 を返り値として返すように関数定義を変更する。
2. 関数 main では、inputMove の返り値を調べて、 0 でない場合は、盤面の表示と石の位置の入力を行っている繰り返しを break 文で脱出する。

othello13 の 実行例

```
s1609h017% ./othello13
 a b c d e f g h
1□□□□□□□□
2□□□□□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = 2d
 a b c d e f g h
1□□□□□□□□
2□□□●□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□□□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = q5
そこには置けません
位置 = g5
 a b c d e f g h
1□□□□□□□□
2□□□●□□□□
3□□□□□□□□
4□□□○●□□□
5□□□●○□○□
6□□□□□□□□
7□□□□□□□□
8□□□□□□□□
位置 = q
s1609h017%
```

次回の予定

次回は othello13.c をさらに発展させて、挟まれた石が反転するようにしたいと思います。

プログラミングおよび実習 I・第 5 回・終り

付録: 前回の演習問題 othello11.c のプログラム例

othello11.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 #define OPPONENT(t)     (-(t))      /* t の相手 */
 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 */
15     for (x = 0; x < BOARD_SIZE; x++)
16         for (y = 0; y < BOARD_SIZE; y++)
17             b[x][y] = NONE;
18     /* 中央に4つの石を置く */
19     b[BOARD_SIZE/2-1][BOARD_SIZE/2-1] = WHITE;
20     b[BOARD_SIZE/2-1][BOARD_SIZE/2] = BLACK;
21     b[BOARD_SIZE/2][BOARD_SIZE/2-1] = BLACK;
22     b[BOARD_SIZE/2][BOARD_SIZE/2] = WHITE;
23 }
24
25 /* 盤面を表示する */
26 void showBoard(int b[][BOARD_SIZE])
27 {
28     int x, y;
29
30     /* 列番号を表示する */
31     for (x = 0; x < BOARD_SIZE; x++)
32         printf(" %c", x + 'a');
33     printf("\n");
34     /* 各行を表示する */
35     for (y = 0; y < BOARD_SIZE; y++) {
36         printf("%c", y + '1');
37         for (x = 0; x < BOARD_SIZE; x++) {
38             if (b[x][y] == BLACK)
39                 printf("●");
40             else if (b[x][y] == WHITE)
41                 printf("○");
42             else
43                 printf("□");
44         }
45         printf("\n");
46     }
47 }
48
49 /* ユーザーが指定した位置に石を置く */
50 void inputMove(int b[][BOARD_SIZE], int turn)
51 {
52     char ch1, ch2;
53     int x, y;
54
55     /* 位置を入力する */
56     while (1) {
```

```
57          ch1 = ch2 = ' ';
58          printf("位置 = ");
59          /* キーボードからの空白類を無視して2文字を入力する */
60          scanf(" %c %c", &ch1, &ch2);
61          /* とりあえず c5 のような形式の入力と仮定してみる */
62          x = ch1 - 'a';
63          y = ch2 - '1';
64          if (0 <= x && x < BOARD_SIZE && 0 <= y && y < BOARD_SIZE
65                  && b[x][y] == NONE) {
66              break;
67          }
68          /* 逆に 5c のような形式の入力と仮定してみる */
69          x = ch2 - 'a';
70          y = ch1 - '1';
71          if (0 <= x && x < BOARD_SIZE && 0 <= y && y < BOARD_SIZE
72                  && b[x][y] == NONE) {
73              break;
74          }
75          printf("そこには置けません\n");
76      }
77      /* 石を置く */
78      b[x][y] = turn;
79  }
80
81  int main()
82  {
83      /* 盤面の状態 (BLACK/WHITE/NONE) */
84      int board[BOARD_SIZE][BOARD_SIZE];
85      /* つぎの手番 */
86      int turn = BLACK;
87
88      reset(board);
89      while (1) {
90          showBoard(board);
91          inputMove(board, turn);
92          turn = OPPONENT(turn);
93      }
94
95      /* プログラムを終了する */
96      return 0;
97  }
```