基礎プログラミング+演習 ＃13 – 文字列の操作+多次元配列

久野 靖 (電気通信大学)

2017.12.11

今回は次の内容を取り上げます。

- C 言語の文字・文字列の扱いと文字列の操作
- 文字列のパターンマッチの考え方と実装方法
- 2 次元配列とポインタの配列

1 前回演習問題の解説

1.1 演習 1 — 簡単な計算

これは関数本体だけ示します。説明は不要でしょう。

```
int fact(int n) {
  int result = 1;
  for(int i = 1; i <= n; ++i) { result *= i; }
  return result;
}
int power2(int n) {
  int result = 1;
  for(int i = 1; i <= n; ++i) { result *= 2; }
  return result;
}
int fib(int n) {
  int x1 = 1, x2 = 1;
  for(int i = 1; i <= n; ++i) { int x3 = x1+x2; x1 = x2; x2 = x3; }
  return x1;
}
int comb(int n, int r) {
  int result = 1;
  for(int i = 1; i <= r; ++i) { result = result * (n-r+i) / i; }
  return result;
}
```

1.2 演習 4 — 動的計画法による釣り銭問題

この問題は部屋割り問題とほぼ同様のやり型でできます。ただし、おつりを「多め」に渡したら損ですから、選択肢を考えるときに「残額がそのコインの額面以上」という条件をつけています。

```
// coins1 --- coin problem DP
#include <stdio.h>
#include <stdlib.h>
#define CMAX 1000
int coincnt[CMAX] = { 0, }, coinsel[CMAX] = { 0, };
void initialize(int *a, int *b, int num);

int main(int argc, char *argv[]) {
  initialize(coincnt, coinsel, CMAX);
  for(int i = 1; i < argc; ++i) {
    int n = atoi(argv[i]);
    printf("coin count %d => %d;", n, coincnt[n]);
    while(n > 0) { printf(" %d", coinsel[n]); n -= coinsel[n]; }
    printf("\n");
  }
  return 0;
}

void initialize(int *a, int *b, int n) {
  for(int i = 1; i < n; ++i) {
    int x = a[i-1] + 1, s = 1;
    if(i >= 5 && a[i-5]+1 < x) { x = a[i-5]+1; s = 5; }
    if(i >= 10 && a[i-10]+1 < x) { x = a[i-10]+1; s = 10; }
    if(i >= 25 && a[i-25]+1 < x) { x = a[i-25]+1; s = 25; }
    a[i] = x; b[i] = s;
  }
}
```

広域変数の配列とその大きさを initialize にパラメタで渡していますが、それは initialize の中では配列を 1 文字で指定したい (その方が読みやすいと思う) からです。枚数は coincnt から読むだけですが、トレースバックは「その金額のときの選択肢のコインの金額を引く」ことを金額が 0 になるまで繰り返します (だから 25 ¢ が 3 枚なら 3 回 25 と出ますがまあいいでしょう)。

```
% ./a.out 80 115 40
coin count 80 => 4; 5 25 25 25
coin count 115 => 6; 5 10 25 25 25 25
coin count 40 => 3; 5 10 25
%
```

1.3 演習 5 — 最大増加部分列

最大増加部分列はまず列の与え方を考える必要がありますが、ここではコマンド引数で与えることにします。それを整数に変換して配列 seq に格納します。そして len が動的計画法のための長さを入れる配列、pre がトレースバック用配列です。

```
// lis1 --- longest increasing sequence DP
#include <stdio.h>
#include <stdlib.h>
#define SMAX 1000
```

```
int seq[SMAX], len[SMAX], pre[SMAX];
int n, maxi, maxl = 0;

int main(int argc, char *argv[]) {
  if(argc < 3) { fprintf(stderr, "more than 2 nums.\n"); return 1; }
  for(int i = 1; i < argc; ++i) { seq[i-1] = atoi(argv[i]); }
  n = argc - 1;
  for(int i = 0; i < n; ++i) {
    int l = 1, p = -1;
    for(int j = 0; j < i; ++j) {
      if(seq[j] < seq[i] && len[j] >= l) { l = len[j]+1; p = j; }
    }
    len[i] = l; pre[i] = p;
    if(len[i] > maxl) { maxl = len[i]; maxi = i; }
  }
  while(maxi >= 0) {
    printf(" %d", seq[maxi]); maxi = pre[maxi];
  }
  printf("\n");
  return 0;
}
```

`l` と `p` は自分が属する列の最大長とその場合の「1 つ前の番号」を表します。どの数字もそれ単独で長さ 1 の列になるので、これを初期値とし、1 つ前は「ない」のでこれを -1 で表します。続いてループで自分の番号より手前の各要素を調べ、自分より小さい値で、なおかつ現在自分が属している最大列の長さと同じ以上であれば、その後ろに自分がつながることでこれまでより長くなるので、長さと手前の番号を更新します。調べ終わったら現在の `l` と `p` を自分の位置に格納します。さらに、全体の最大列が必要なのでそれは `maxl` に長さ、`maxi` に位置を覚えます。最後まで格納し終わったら、見つかった最大長の列についてトレースバックしながら出力します (なので表示は逆順になります)。

```
% ./a.out 1 5 7 2 6 3 4 9
 9 4 3 2 1
%
```

2 C 言語の基本型と文字列

2.1 C 言語の基本型 exam

基本型 (primitive type) とは「ひとまとまりでそれ以上分解できないような値の型」をいい、論理値 (はい/いいえの値)、文字、数値 (整数、実数) などがこれに相当します。なのですが、C 言語はそのデザインが「できるだけ数値型でやる」という設計になっています。具体的には次の通りです。

- 論理型 (Boolean) — 前にも出て来たように、C ではもともと論理型がなくて整数の 1 が「はい」、0 が「いいえ」を表しています。しかし他の言語と比較して、それでは書きづらいということで多くのプログラマが型名 `bool`、「はい」の値 `true`、「いいえ」の値 `false` を自己流で定義してきました。それもまたよくないということで、C99 からは「`#include <stdbool.h>`」でヘッダファイルを取り込むことで上の 3 つが使えます。[1]

[1] C89 まででは自分で「`#define bool int`」「`#define true 1`」「`#define false 0`」などと定義してください。

- **文字型** (character) — C 言語には文字を表す型 `char` がありますが、実際にはこの型は「8 ビット幅の整数型」であり、しかも符号つきなので「-128〜127」の範囲の整数が入れられます。そして、文字を表す定数は「`'a'`」、「`'!'`」のようにシングルクォートで囲み、中に 1 文字だけを入れられます。[2] そして整数型なので、`'a'` は 97、`'!'` は 33 とまったく同じです。これらはそれぞれの文字の ASCII コードの値なわけです。[3]

表 1: 文字定数と整数定数の記法の例

	文字定数			整数定数		
文字	文字・記号	16 進	8 進	16 進	8 進	十進
タブ (TAB)	`'\t'`	`'\0x09'`	`'\011'`	`0x09`	`011`	`9`
改行 (NL)	`'\n'`	`'\0x0a'`	`'\012'`	`0x0a`	`012`	`10`
復帰 (CR)	`'\r'`	`'\0x0d'`	`'\015'`	`0x0d`	`015`	`13`
ナル文字	`'\0'`	`'\0x0'`	`'\000'`	`0x0`	`00`	`0`
通常文字	`'a'`	`'\0x61'`	`'\141'`	`0x61`	`0141`	`97`

  文字定数には改行などの制御文字や任意の文字コードを指定する記法がありますが、整数定数と一緒に説明します (図 1)。整数では「`0x`」ではじまり 16 進法の数字を並べた 16 進定数、「`0`」ではじまり 8 進法の数字 (0〜7) を並べた 8 進定数が書けます。そして文字定数でも「`'...'`」の中に「`\`」に続けて同様に書くことができます (8 進は常に 0〜7 の数字 3 個)。さらに、改行、復帰、タブなどよく使う制御文字はこれらを 1 文字で表す記法があります。このように「`\`」で始まる列は通常の文字と違う特別な意味を持つことから**エスケープシーケンス** (escape sequence) と呼ばれます。なお「`\`」(エスケープ文字) そのものを書きたい場合は「`'\\'`」、シングルクォートを書きたい場合は「`'\''`」です。

- **整数型** (integeral type) — 整数型として `int` とビット数の多い `long` があることは既に説明しました。定数の表記方法に 8 進、16 進もあることは上述の通り。なお、`long` の定数を指定したい場合は「`1L`」のように最後にエルをつけます (小文字も使えますが 1 と紛らわしいので避けたい)。さらに文字型を含む整数型では、`unsigned int`、`unsigned long`、`unsigned char` のように冒頭に**符号なし** (unsigned) と指定することで、2 の補数の代わりに符号なし整数として扱う指定ができます (たとえば `unsined char` は 0〜255 が入れられます)。[4]
- **実数型** (real type) — 実数型として `double` とビット数の少ない `float` があることは既に説明ました。実数の定数は十進のみで、数字列の中に小数点または指数表記があれば実数、そうれ以外ば整数となります。「`10000.0`」、「`1e5`」「`1.0e+05`」はどれも 1 万を表す実数定数で型は `double`、`float` の定数を指定したい場合は最後に「`F`」をつけます (小文字も使えます)。

2.2 文字列の扱い exam

文字型の説明に続いてようやく、**文字列** (string) の説明ができます。C 言語では文字列は「文字の配列」です。そして、文字の配列の初期化に文字列を指定できます。簡単な例を見てみましょう。泥縄ですが、書式文字列において `char` の入出力は「`%c`」、文字列の入出力は「`%s`」で指定します。

```
// str1.c --- test string and array ops.
#include <stdio.h>
void showstr(char *s, int len);
```

[2] Ruby では`"..."`も`'...'` も文字列でしたが、C では前者は文字例、後者は文字で全く違っています。
[3] 日本語はどうなのという疑問が湧くでしょうけれど、C 言語は歴史的経緯により日本語の扱いが面倒なので、日本語を扱いたければ Ruby などを使うことを進めます。本資料では C 言語で扱う文字はすべて ASCII の範囲とします。
[4] このように様々な整数型があり、ビット幅もシステムによって異るので、正しい整数型を選ぶのは結構大変です。このため標準ヘッダファイルで `size_t`、`time_t` などの型が定義されていて、標準ライブラリではこれらの型を使っています。

```
int main(void) {
  char s[] = "abcde";
  printf("size of s = %d\n", sizeof(s));
  s[2] = 'X';
  printf("char: "); scanf("%c", s+3); showstr(s, sizeof(s));
  printf("str: "); scanf("%5s", s); showstr(s, sizeof(s));
  return 0;
}
void showstr(char *s, int len) {
  for(int i = 0; i < len; ++i) { printf("(%c %02x)", s[i], s[i]); }
  printf("; s = %s\n", s);
}
```

showstr は文字列配列 (実際には文字型へポインタ) と文字列の長さを渡して中身を表示させる下請け関数で、後で読みます。main ですが、まず変数 s は配列で、その初期値として文字列"abcde"を持たせます (配列のサイズは文字列の長さから決められます)。とりあえず sizeof でその配列の長さを表示させて確認します。

つぎに、その 2 文字目に「X」を入れ、3 文字目に scanf で 1 文字を読み込ませます。「s+3」というのは、s が配列なのでその先頭から 3 つ先の要素のアドレスを意味するのでしたね (そして同じものを&s[3] と書いても指定できるのでした)。そして showstr で中身を表示したあと、こんどは scanf で s に別の内容を読み込みます。ここで重要なのが、書式文字列で「"%5s"」のように長さの上限を指定していることです。これを指定しないと、長い文字列が入力された際に配列の領域を超えた書き込みが起きてトラブルになります。最後に読み込んだ内容を表示します。

最後に showstr の中身を見ましょう。長さが渡されるので、その長さぶんだけくりかえし、各文字を文字そのままと 16 進で表示し、最後に文字列全体を表示します。では実行してみましょう。

```
% gcc str1.c -std=c99
% ./a.out
size of s = 6
char: V          ←「V」を入力
(a 61)(b 62)(X 58)(V 56)(e 65)( 00); s = abXVe
str: this is     ←「this is」を入力
(t 74)(h 68)(i 69)(s 73)( 00)( 00); s = this
%
```

どうでしょうか。予想外のことがいつかあったかと思います。

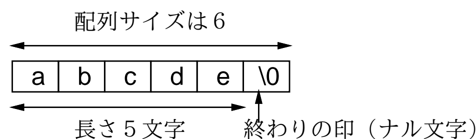

図 1: C の文字列とナル文字

まず、文字列は"abcde"の 5 文字に見えるのに、配列 s のサイズが 6 です。これは、C では文字列は最後に**ナル文字** (null character)「'\0'」がくっついているため 1 文字多くなるのです (図 1)。考えてみてください。文字列はいろいろな長さのものを扱いたいでしょうから、その長さを表す方法が必要ですね。常に長さを別の変数で管理するより、「最後にナル文字があってそこまでが文字列」とするのが単純でよい、というのが C 言語の設計方針なのでした。次に特定の場所に代入したり文字

を読み込むとそこが置き換わる、というのは想定通りですね。ナル文字を%c で出力すると何も出てこないというのはまあ分かります。次に「this is」を入力したけれど、入力できたのは「this」だけです。これは%s で入力すると「空白、改行、タブで区切られた 1 語を入力する」ことになっているためです。入力の最後にナル文字がつけられていることにも注意。

空白が入れられないのは不便ですね? 空白まで含めて行末まで入力したい場合は次のものを使ってください。これも読み込み過ぎが起きないように配列の長さを上限として渡してください。

```
fgets(文字列配列, 長さ上限, stdin);
```

あと 2 つほど補足です。"abcd"のような文字列リテラルの中にも文字リテラルと同じエスケープシーケンスが使えます。そして、文字列リテラルは配列と同じものなので「char *p = "abcd"」のように書くことができます。つまり文字列リテラルの型はその文字列が入った配列の先頭要素のポインタ値ですが、ただし文字列リテラルの中身を直接書き換えてはいけません (先の例では配列の初期値なので変更しても OK でした)。

演習 1 上の例題をそのまま打ち込んで動かしなさい。とくに文字列入力で文字列が 5 文字以上のときどうなるか観察すること。OK なら、次のものをやってみなさい。文字列配列のサイズは 20 くらいにすることをすすめます。

a. fgets を使って 1 行 (改行まで) 入力し、それを出力する。

b. 上記に加えて、文字列の長さを調べる関数 int mystrnlen(文字列, 長さ上限) を作る。

c. 上記を使って fgets で読み込んだ時に末尾につく改行文字を削除する関数 void chopnl(文字列, 長さ上限) を作る。末尾が改行でないときは削除しないこと。

d. 文字列中の指定した文字 (たとえば空白) をすべて削除して詰める関数 void deletechar(文字列, 長さ上限, 文字) を作る。

e. 文字列を左右ひっくり返す関数 void reverse(文字列, 長さ上限) を作る。

f. 文字列を別の文字配列にコピーする関数 void mystrncpy(文字配列, 文字列, 長さ上限) を作る。

g. 文字配列に中の文字列の後ろに別の文字列をくっつける関数 void mystrncat(文字配列, 文字列, 長さ上限) を作る。

h. 2 つの文字列を比較して等しければ 0、1 番目のものがコード順で前なら-1、2 番目がコード順で前なら 1 を返す関数 int mystrncmp(文字列, 文字列, 長さ上限) を作る。たとえば mystrncmp("abcd", "abcd", 100) → 0、mystrncmp("abcd", "efgh", 100) → -1、mystrncmp("abcd", "abca", 100) → 1 となる。

   ヒント: 先頭から両方の文字列を比べていき、最後まで (ナル文字まで) 同じなら等しい。そうでなければ、違いがあったところの対応する文字の大小関係で 1 か-1 を返せばよい。

2.3 文字列ライブラリ exam

上で演習問題にしたもののうち「my…」となっているものは、実はライブラリで実装されているものです (混同しないように my をつけました)。文字列ライブラリを使うためには、「#include <string.h>」を指定する必要がありますが、いちいち自分で定義せずに使えるので便利です。

- size_t strlen(s), size_t strnlen(s, L) — 領域 s にある文字列の長さを返す。
- char *strcpy(s, t), char *strncpy(s, t, L) — 領域 t から s に文字列をコピー。
- char *strcat(s, t), char *strncat(s, t, L) — t の文字列を領域 s の文字列末尾に連結。
- int strcmp(s, t), int strncmp(s, t, L) — 領域 s と t の文字列を比較し、結果として正の数、0、負の数のいずれかを返す。

文字 n がついているものは長さ上限を渡すもので、ついていないものは長さ上限を渡しません。長さ上限を渡さないと、用意されている領域より先まで書き込んでしまう恐れがあるので、常に長さ上限を渡すものを使うことを勧めます。ただし、`strlen` については領域に書き込まないので、渡す文字列を間違えない限りはあまり危険はありません。

なお、戻り値の型が見慣れないと思ったかも知れません。`size_t` は領域の長さなどの表現に使う標準ライブラリの型ですが、`int` に代入できると思って差し支えありません。また、コピー系のものは `char*`を返しますが、これは何も返さないよりはたまたま使える場合は利用できるように操作した文字列のアドレスを返しているもので、本資料では利用していません。

2.4 switch 文

文字列の話題ではないのですが、文字列と一緒に使うことが多い switch という制御構文をここで紹介しておきます。switch 文の一般形は次のようなものです。

```
  switch(式) {
case 値: case 値: 文…; break;
case 値: case 値: 文…; break;
default: 文…;
  }
```

まず「式」は整数型の式、「値」は整数の定数である必要があります (文字リテラルも整数の定数です)。そして、最初の式がどれかの「値」に一致したら、その場所にある文に実行が移ります (どれにもあてはまらなければ `default:`の次にある文に移ります。`default:`が書かれていなければ switch 文の中は実行せず次の文に進みます)。switch 文の中の文に実行が移ったあとは普通の実行ですが、`break;` はこの switch 文から外に出ることを意味します (もしうっかり `break;` を書き忘れると次にある `case` ラベルや `default:`の後の文に「合流」するので注意。はまりやすいです)。

では例題として、これまでお世話になってきた `atoi` のそっくりさんを作ってみます。

```
// switch1.c --- demonstaration of switch stat.
#include <stdio.h>
#include <stdbool.h>
int myatoi(char *s);

int main(int argc, char *argv[]) {
  if(argc != 2) { fprintf(stderr, "needs 1 argument.\n"); return 1; }
  printf("%d\n", myatoi(argv[1]));
  return 0;
}
int myatoi(char *s) {
  int sign = 1, val = 0;
  switch (*s) {
case '-': sign = -1;
case '+': ++s;
  }
  while(true) {
    char c = *s++;
    switch(c) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
      val = val * 10 + (c - '0'); break;
```

```
default:
        return sign * val;
      }
    }
  }
```

myatoi だけ読めばいいですね。まず符号の変数 sign に 1、値の変数 val に 0 を入れます。次に文字列の先頭の文字によって分岐し、'-' だった場合は sign に −1 を入れ直し、そして s を次の文字に進め…ますが、それは'+' の場合もやることなので、わざと break; を書かずに'+' の処理に合流しています。符号でない場合はこの部分の処理は何も起きません。次は無限ループで、その中で、s が指している箇所の文字を取り出して c に入れ、「その後で」s を次の文字に進めます。こうやって 1 文字ずつ処理していくわけです。そして、'0'～'9' の数字のどれかだった場合はこれまでの値を 10 倍して、c から'0' の値を引いたものを加えます。数字の文字コードは連続しているので、この引き算で'1' なら 1、'3' なら 3 等が得られるのです。そして、それ以外の文字 (文字列末尾の'\0' も含む) だったら、これまでに作って来た sign と val を乗じたものを返せばいいのです。ということは、途中に数字以外のものが出てきたらそこで終わりますが、atoi もそういうふうにできているのでした。さて、いかがでしたか。switch 文、便利でしょうか? しかし…次のも見てください。

```
int myatoi(char *s) {
  int sign = 1, val = 0;
  if(*s == '-') { sign = -1; ++s; } else if(*s == '+') { ++s; }
  while(true) {
    char c = *s++;
    if(c < '0' || c > '9') { return sign * val; }
    val = val * 10 + (c - '0');
  }
}
```

別に if 文でいいような気もしますね。まあ用途として合っていそうなことがあったときに、switch 文のことも思い出してあげてください。

演習 2 上の例題の好きな方を打ち込んで動かしなさい。動いたら次のいずかをやってみなさい。

a. 数字列の先頭が 0 ではじまったら 8 進として受け取るようにする。

b. 数字列の先頭が 0x ではじまったら 16 進として受け取るようにする。

c. atoi の代わりに自分流 atof(文字列を実数に変換) を (指数記法「e ±数字列」は不要)。

d. 指数記法も扱える自分流 atof を作る。

e. switch 文を使った何か面白いプログラムを作る。

2.5 部分文字列の検索

次のテーマは、ある (長い) 文字列 str 中に指定した (短い) 文字列 pat が含まれているか、含まれているとしたらどこに含まれているかを調べるという問題です。図 2 のように、「'abbabbab'」の中で「'bbab'」がどこにあるかを探すと、(先頭が 0 なので)1 文字目と 4 文字目にあることがわかる、という具合です。完成したプログラムを動かすと次のように先頭位置に印を表示します。

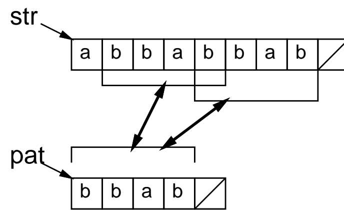

図 2: 部分文字列の検索

```
% ./a.out 'abbabbab' 'bbab'
abbabbab
 ^
abbabbab
    ^
%
```

では、いきなり全部示すのでなく、下請けの関数から順に読んでいきます。まず、文字列 `str` と `pat` の先頭 `len` 文字が一致していれば `true`、そうでなければ `false` を返す関数 `nmatch` を見てみます。なお、`bool`、`true`、`false` を使っていますので、`#include <stdbool.h>`が必要です。

```
bool nmatch(char *str, char *pat, int len) {
  while(len-- > 0) {
    if(*str++ != *pat++) { return false; }
  }
  return true;
}
```

「なんだこれは」と思うかも知れませんが、このようなのが C 言語流です。まず、`len` 文字比べるのですから、`len` が 0 より大きくなければ比べません。つまり `len > 0` を条件とする `while` を書いています。普通なら `while` の末尾で `len` を 1 減らすのですが (`len` はこの関数内ではローカル変数と同じなので他に使わなければ変更して行って構いません)、`len` を使っているのはここだけなので、条件を調べるため参照したあとすぐ減らしてしまいます。これが後置演算子 `len--`の働きでした。[5]

次に文字列 `str` と `pat` の先頭文字が等しいか調べますから、`if` の条件は`*str == *pat` でこれが「いいえ」なら `false` を返します。そうでなければ `str` も `pat` も次の文字の場所に増やしますが、それを別に書く代わりに後置演算子の`++`で行えます。指定文字ぶんだけ調べてループが終わったら `true` を返せばよいですね。

さて、では次はこれを利用して「どこか途中にある」場合にその位置を探す関数 `findstr` を作ります。どこにあるかは「何文字目」という整数を返せば良さそうですが、ここではその代わりに「その見つかった位置のポインタを返す」ことにします。というのは、ポインタ演算 `p + i` で「`p` の `i` 要素ぶん先のポインタ」が得られるのと逆に、「`q - p`」というポインタ同士の演算で「位置が何要素ぶん離れているか」が計算できるので同じことだからです。なお、見つからなかった場合は `NULL` という値を返します (実態は 0 ですが C 言語では `stdio.h` や `string.h` で定義されていて「ない」ことを表すのに使います)。それでは見てみましょう。

```
char *findstr(char *str, char *pat, int len) {
  if(len == 0) { return NULL; }
  for(int l = strlen(pat); l <= len; ++str, --len) {
    if(nmatch(str, pat, l)) { return str; }
```

[5]復習: `--i` は `i` を 1 減らし減らした後の `i` の値を返します。`i--`は `i` を 1 減らすが減らす前の `i` の値を返します。

```
  }
  return NULL;
}
```

len は str の長さで、これが 0 のときは見つかりようがないので NULL をすぐ返します。次はいきなり for 文ですが、初期設定として変数 l(小文字のエル — 1 と間違いやすいですが長さにはエルを使いたいので) に pat が何文字ぶんかを入れ、ループの条件としては l が len 以下の間繰り返します。この条件はヘンに思えるかも知れませんが、ある位置で試してあてはまらなければ、str を 1 進めて (つまり先頭の文字は除外して)、ということは長さは減らす必要があるので len は 1 減らす、というのがループ周回ごとの処理です。この 2 つを一緒に書くためにカンマ演算子, で並べています (for のこの場所にセミコロンは書けないので)。

こういうコンパクトに詰め込めるところが C 言語らしいところです。で、ループ本体では現在位置にあてはまるかを nmatch で調べ、OK なら str を返します。最後まであてはまらずにループを抜けたら NULL を返します。

では main を見てみましょう。コマンド引数で受け取った 2 つの文字列を str および pat として findstr を呼びます。

```
// findstr1.c --- search substring occurence in longer string.
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
void putrepl(char c, int count);
char *findstr(char *str, char *pat, int len);
bool nmatch(char *str, char *pat, int len);

int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  char *str = argv[1];
  int len = strlen(str);
  while(true) {
    str = findstr(str, argv[2], len - (str - argv[1]));
    if(str == NULL) { return 0; }
    printf("%s\n", argv[1]);
    putrepl(' ', str - argv[1]); printf("^\n"); ++str;
  }
}
void putrepl(char c, int count) {
  while(count-- > 0) { putchar(c); }
}
(ここに findstr)
(ここに nmatch)
```

長さのかわりにへんなものを渡していますが何でしょうか? それに while ループ (それも条件が true なので無限ループ) ですが…まず最初は、str と argv[1] は同じなので (str - argv[1]) は 0 で、長さ len がそのまま渡ります。そして、findstr であてはまり位置が返されます。これが NULL ならあてはまりは無いので 0 を返して終わりです。そうでない場合は、もともとの文字列を表示し、次に「str が argv[1] より何文字進んでいるか」計算してそのぶんだけ空白を出力し (putrepl はすぐ読めますね — putchar は 1 文字を出力する関数です)、そのあと目印の記号と改行を出力します。ループ本体の最後で str を 1 文字ぶん先に進めるので、次は今の場所より先にあるあてまり位置を探すこ

とになります。というわけで、findstr に渡す長さは len から先に進んだぶんだけ差し引く必要があるのでした。実行結果は先に示した通り。

演習 3 このプログラムをそのまま動かしなさい。OK なら、目印を先頭位置に 1 文字表示するのでなく、pat のあてはまる範囲に連続して表示するように直してみなさい。

2.6 正規表現のマッチ

上では「指定した文字列ぴったり」があてはまる位置を調べましたが、Unix の**正規表現** (regular expression) のように「パターン」が指定できるとずっと便利です。正規表現には多くの機能があり全部は大変なので、とりあえず「+」つまり「直前の文字を 1 回以上繰り返す」機能だけ作ってみます。実行例を先に見ましょう。

```
% ./a.out 'aababbaaabaaabbbaa' 'abb+aa+'
aababbaaabaaabbbaa
   ^^^^^^
aababbaaabaaabbbaa
            ^^^^^^
%
```

パターンは abb+aa+つまり「a が 1 個、b が 2 個以上、a が 2 個以上」で、実際そのようなところだけに印がついています。

ではコードを見ます。main は先のものと似ていますが、大きく違うのは matchstr(先の findstr の代わり) は「どこからどこまでがあてはまったか」返す必要があるという点です。2 つ値を返す際、Ruby では配列が使えますが、C では配列はアドレスしか返せず、このような用途に不向きです。そこで先と同じに先頭位置は返値で返し「どこまで」の方は scanf のように変数のアドレスを渡し、その変数に格納してもらうことで受け取ります。受け取る変数の型は文字へのポインタ「char*」ですから、その変数へのポインタは「char**」つまりポインタへのポインタ、になります。

```
// matchstr1.c --- match pattern occurence in a string.
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
void putrepl(char c, int count);
char *matchstr(char *str, char *pat, int len, char **tail);
char *pmatch(char *str, char *pat, char *lim);

int main(int argc, char *argv[]) {
  if(argc != 3) { fprintf(stderr, "needs 2 args.\n"); return 1; }
  char *str = argv[1];
  int len = strlen(str);
  while(true) {
    char *tail;
    str = matchstr(str, argv[2], len - (str - argv[1]), &tail);
    if(str == NULL) { return 0; }
    printf("%s\n", argv[1]);
    putrepl(' ', str-argv[1]); putrepl('^', tail-str);
    putchar('\n'); ++str;
  }
```

```
}
void putrepl(char c, int count) {
  while(count-- > 0) { putchar(c); }
}
```

「どこまで」が受け取れれば、その範囲全体に印をつけるのも簡単です (putrepl は先の例と同じ)。

次に matchstr ですが、長さ 0 なら NULL を返すのは同じ。次に、探される文字列の末尾位置 lim を計算し、str がそれより手前にある間 str を 1 つずつ進めながら調べる for ループに入ります。その中では pmatch を読んで現在の str のその位置にあてはまるかどうか調べます。pmatch はあてはまるならその終わりの位置、あてはまらなければ NULL を返すようにしたので、NULL でなければ終わりの位置を渡されて来たポインタの場所に格納して先頭位置を返します。for ループを終わってもあてはまりが無ければ自分も NULL を返します。

```
char *matchstr(char *str, char *pat, int len, char **tail) {
  if(len == 0) { return NULL; }
  for(char *lim = str + len; str < lim; ++str) {
    char *t = pmatch(str, pat, lim);
    if(t != NULL) { *tail = t; return str; }
  }
  return NULL;
}
```

pmatch では終端位置 lim が渡されてくるので、現在位置 str がそれより先ならあてはまらないので NULL を返します。そうでなくてパターンの最後まで来たら、あてはまり終わったということなので成功として現在位置を返します。その次の printf はデバッグ用ですが動きが分からないときはコメントアウトして動かしてみてください。

```
char *pmatch(char *str, char *pat, char *lim) {
  if(str > lim) { return NULL; }
  if(*pat == '\0') { return str; }
//printf("pmatch: '%s' '%s'\n", str, pat);
  if(pat[0] == str[0] && pat[1] == '+') {
    int i = 1;
    while(pat[0] == str[i]) { ++i; }
    for( ; i > 0; --i) {
      char *t = pmatch(str+i, pat+2, lim);
      if(t != NULL) { return t; }
    }
    return NULL;
  } else {
    if(*pat != *str) { return NULL; }
    return pmatch(str+1, pat+1, lim);
  }
}
```

さて次ですが、現在位置があてはまって次が「+」のときは繰り返しのパターンです (最初の条件は*pat == *str でもいいのですが、この後*(pat + 1) なども必要になるので代わりに短く書ける配列添字記法に揃えています)。

その処理の内容ですが、まず変数 i を用意し、str[i] が pat[0] と等しい間 i を増やすことで、pat[0] の文字の並びが「最大で」いくつあるかを i に求めます。ただしこの i は「最大の」繰り返

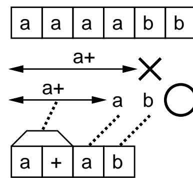

図 3: パターンのあてはまりの探索

し数であり、実際にあてはまるのはそれより少ない回数かも知れません。図 3 を見てください。文字列が'aaaabb'、パターンが'a+ab' のとき、先頭からマッチさせるとしてパターン'a+' を最大の'aaaa' とマッチさせると、文字列の次は'b' ですからマッチしません。1 つ減らして'aaa' にすれば、残りが'ab' になるのでうまくマッチします。ですから次に、for ループで i を 1 つずつ減らしながらこれで OK かどうかを調べます。[6]

なお、この for ループでは初期化は空です。このように for 文では初期化も条件も更新も空っぽでも大丈夫です。そこでは何もしないし、条件は成り立っているものと見なされます。つまり「for(; ;) {...}」は無限ループつまり「while(true) {...}」と同等です。

話を戻して、i を変化させながら、その反復数の状態で文字列の残り、パターンの残り (+の次の文字)、文字列の終端 (これはずっと同じ) を渡して自分自身を再帰的に呼びます。そこから返された値が NULL でなければ OK なので、その値をそのまま返します (これがあてはまりの終端)。ループを全部試し終わっても成功しなければ NULL を返します。

以上が「+」パターンの処理で、残りは普通の場合です。次の文字どうしが一致しなければ NULL を返し、一致していれば文字列もパターンも 1 文字進めた状態で pmatch を再帰呼び出しします。

この pmatch は 1 つ処理したら自分を再帰的に呼び出しますが、理由がお分かりでしょうか。それは、たとえば「+」パターンが複数あった場合、最初のもので長さを変化させながら、さらにそれぞれについて 2 番目でも長さを変化させ調べる必要があるからです。そのためには普通は 2 重ループが必要ですが、再帰を使えば何重のループでも実行時に作り出せます。

演習 4 上のコードを打ち込み、パターンが正しく処理されていることを確認しなさい。OK なら、次のことをやってみなさい。

a. 「+」(1 回以上の繰り返し) に加えて「*」(0 回以上の繰り返し) も記述できるようにしてみなさい。
b. 「?」(直前の文字があってもなくてもよい) を実現してみなさい。
c. ^(先頭に固定) と$(末尾に固定) を実現してみなさい。
d. 文字クラス [...](... の文字のいずれかならあてはまる) を実現してみなさい。[^...](... のいずれでもなければ) も実現できるとなおよいです。
e. ここまでに出て来た特殊文字の機能をなくすエスケープ記号「\」を実現しなさい (この文字に続いて特殊文字があった場合通常の文字として扱う)。
f. その他、パターンマッチにおいてあると面白いと思う好きな機能を選び実現しなさい。

3 ポインタ配列と多次元配列

3.1 ポインタ配列

ここまで C 言語の配列として 1 次元のものばかり扱って来ましたが、2 次元以上の配列ももちろん使いたいです。しかしその前に、ポインタの配列を見てみましょう。一番お世話になっているのが

[6]なぜ長い方から順に調べるかということですが、それはパターンを書いたらそれに対するなるべく長いあてはまりを優先するのが人間にとって自然だからです。

argv です。このパラメタはコマンド行で指定した文字列の並びを受け取ります。つまり文字列の配列ですが、文字列というのは C 言語では文字の配列でしたね。そして配列の値というのは先頭要素へのポインタなわけで、ですから argv はポインタの配列なのです。文字でなく整数や実数のポインタの配列も当然あり得ます (図 4)。[7]

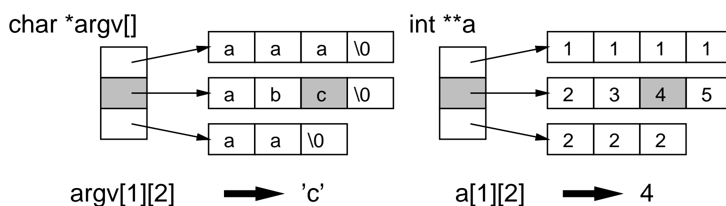

図 4: ポインタの配列の図解

ところで、char *argv[] って何でしょう? C では配列の名前はその先頭要素のポインタ値と同じであり、配列を関数に渡すときはパラメタの型はポインタというふうに説明してきました。しかし配列を渡すことを想定しているのなら、パラメタも配列と書きたいですよね? そこで C 言語では、パラメタに型を書く時、パラメタ名の直後に [] を書くことができ、これを先頭の*と同じに扱います。ですから実際には char **argv でもよかったのです。[8]

3.2 多次元配列

さて、C では 2 次元以上の配列も普通に要素数を指定することで作り出せます。int c[10][10]; であれば 100 要素、int d[10][10][10]; であれば 1000 要素の領域が確保されて使えます。参照も c[i][j] とか d[j][k][l] とか普通です。

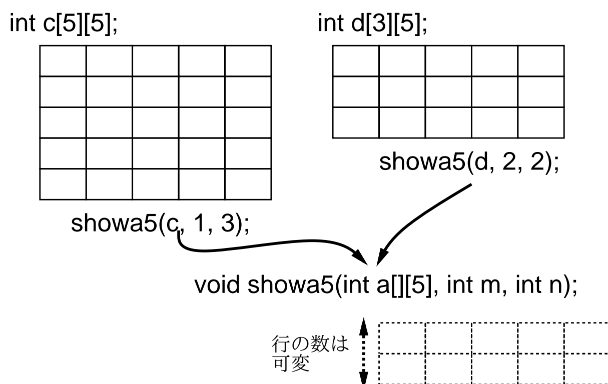

図 5: 2 次元配列の図解

ただし、2 次元以上の配列をパラメタとして渡すときはちょっと注意が必要です。1 次元目についてはこれまでと同じ考えでポインタとして渡され、実際の個数はいくつでもよいのですが、2 次元目以降の要素数は定数を書かないといけません。そうしないと呼ばれる側で大きさが分からないからです。このため、たとえば int c[5][5] や int d[3][5] を受け取るパラメタは int a[][5] のように書く必要があります (図 5)。例を見てみます。

```
// array2dim1.c --- demonstorate of 2dim array.
#include <stdio.h>
void showa5(int a[][5], int n, int m);
```

[7]Ruby では「配列の配列」等はすべてポインタの配列でした。配列オブジェクトはすべて参照値として扱うので。
[8]2 個以上 [] を連続させることはできません。その説明はすぐ後で。

```
int main(int argc, char argv) {
  int c[5][5] = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6},
    {3, 4, 5, 6, 7}, {4, 5, 6, 7, 8}, {5, 6, 7, 8, 9} };
  int d[][5] = {{5,4,3,2,1},{6,5,4,3,2},{7,6,5,4,3}};
  showa5(c, 1, 3); showa5(d, 2, 2); return 0;
}
void showa5(int a[][5], int m, int n) {
  for(int i = m; i <= n; ++i) {
    for(int j = 0; j < 5; ++j) { printf("%3d", a[i][j]); }
    putchar('\n');
  }
}
```

実行例は次の通り。

```
% ./a.out
  2  3  4  5  6
  3  4  5  6  7
  4  5  6  7  8
  7  6  5  4  3
%
```

では、2 次元目以上の要素数が様々なものを受け取る関数はどうすればいいの? それは C89 までででは「できません」でしたが、C99 ではできます。具体的には、C99 ではパラメタの配列の要素数に一緒にパラメタとして受け取る整数型のパラメタ名を書けます (ただし配列より前に書かれている必要あり)。つまり、次のようにできるのです。

```
void showx(int w, int a[][w], ...) { ... }
```

ただしこの機能を使うと簡単には C89 に変換できなくなるので、注意して使ってください。[9]

演習 5 2 次元以上の配列またはポインタ配列を使った自分の面白いと思うプログラムを作りなさい。

本日の課題 **13A**

「演習 1」または「演習 2」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. C 言語の文字列機能についてどのように思いましたか。

Q2. C 言語のポインタ配列、2 次元配列について納得しましたか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 **13B**

「演習 1」〜「演習 5」の (小) 課題から 1 つ以上を選択してプログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. 文字列の基本的な操作ができるようになりましたか。

Q2. パターンマッチがどのように実装されるか理解しましたか。

Q3. 課題に対する感想と今後の要望をお書きください。

[9] また、この機能は C11 規格ではオプションとなっています。