基礎プログラミング+演習 ＃14 – 構造体と動的データ構造

久野 靖 (電気通信大学)

2018.2.3

今回は次の内容を取り上げます。

- C 言語の構造体機能
- 構造体とポインタによる動的データ構造

1 前回の演習問題解説

1.1 演習 1 — 文字列の基本的な演習

細かい課題が沢山あるのでいちいち#include や main は示さず、該当する関数だけ示すようにします。まず a は fgets を使ってみます。

```
#define BUFSIZE 100
char buf[BUFSIZE];
void ex13_1a(void) [
  printf("?> "); fgets(buf, BUFSIZE, stdin);
  printf("'%s'\n", buf);
}
```

実行のようすです。確かに改行が含まれています。

```
% ./a.out
?> how are you?
'how are you?
'
```

そこで b として長さを調べ、c として末尾が改行なら'\0' に変更。

```
int mystrlen(char *s, int lim) {
  int len = 0;
  while(len < lim && s[len] != '\0') ++len;
  return len;
}
void chopnl(char *s, int lim){
  int len = mystrlen(s, lim);
  if(len > 0 && s[len-1] == '\n') { s[len-1] = '\0'; }
}
void ex13_1bc(void) [
  printf("?> "); fgets(buf, BUFSIZE, stdin);
  chopnl(buf, BUFSIZE);
  printf("'%s'\n", buf);
}
```

こんどは次のように改行が削除できています。

```
% ./a.out
?> How are you?
'How are you?'
```

指定文字の削除です。変数 i を 1 文字ずつ進めながらその位置の文字を変数 j の位置にコピーして j を進めますが、削除する文字のときはコピーしません。'\0' に遭遇したら終わります。

```
void delchar(char *s, char del, int lim) {
  for(int i = 0, j = 0; i < lim; ++i) {
    if(s[i] != del) { s[j++] = s[i]; }
    if(s[i] == '\0') { break; }
  }
}
void ex13_1d(void) [
  printf("?> "); fgets(buf, BUFSIZE, stdin);
  chopnl(buf, BUFSIZE); delchar(buf, ' ', BUFSIZE);
  printf("'%s'\n", buf);
}
```

実行してみると、確かに空白が詰められます。

```
% ./a.out
?> How are you?
'Howareyou?'
```

次は e で文字列の反転です。下請けの交換関数を作り、文字列の前半それぞれについて、後半の対応する位置と交換します。

```
void cswap(char *s, int i, int j) {
  char c = s[i]; s[i] = s[j]; s[j] = c;
}
void reverse(char *s, int lim) {
  int len = mystrlen(s, lim);
  for(int i = 0; i < len/2; ++i) {
    cswap(s, i, len-i-1);
  }
}
void ex13_1e(void) [
  printf("?> "); fgets(buf, BUFSIZE, stdin);
  chopnl(buf, BUFSIZE); reverse(buf, BUFSIZE);
  printf("'%s'\n", buf);
}
```

実行してみます。確かに反転されています。

```
% ./a.out
?> How are you?
'?uoy era woH'
```

次はfで文字列コピー、gで文字列連結ですが、コピーは先のdelcharと非常に似ていて、ただ削除する文字がないのと、コピー元が別の文字列だということです。連結は最初にコピー先の長さを調べてそこからコピーするというだけの違いです。

```
void mystrncpy(char *s, char *t, int lim) {
  for(int i = 0, j = 0; i < lim; ++i, ++j) {
    s[j] = t[i];
    if(s[i] == '\0') { break; }
  }
}
void mystrncat(char *s, char *t, int lim) {
  int j = mystrlen(s, lim);
  for(int i = 0; j < lim; ++i, ++j) {
    s[j] = t[i];
    if(t[i] == '\0') { break; }
  }
}
void ex13_1fg(void) {
  mystrncpy(buf, "This is not ", BUFSIZE);
  mystrncat(buf, "a pen.", BUFSIZE);
  printf("'%s'\n", buf);
}
```

動かすと確かに連結できています。

```
% ./a.out
'This is not a pen.'
```

次はhの文字列比較ですが、だいたいヒントの通りですね。1か-1を返すところは3項演算子 (if-tthen-else式) を使っています。両方とも文字列の終わりになったか上限に到達したら抜け出て0を返します。

```
int mystrncmp(char *s, char *t, int lim) {
  for(int i = 0; i < lim; ++i) {
    if(s[i] != t[i]) { return (s[i] > t[i]) ? 1 : -1; }
    if(s[i] == '\0') { break; }
  }
  return 0;
}
void ex13_1h(void) {
  printf("%d\n", mystrncmp("abcde", "abcde", 6));
  printf("%d\n", mystrncmp("abcde", "abcdef", 6));
  printf("%d\n", mystrncmp("abcde", "abcd", 5));
  printf("%d\n", mystrncmp("abcd", "abce", 5));
  printf("%d\n", mystrncmp("abcd", "abad", 5));
}
```

テストケースを考えるほうがややこしそうですね。

```
% ./a.out
```

```
0
-1
1
-1
1
%
```

1.2 演習 2 — atoi と atof の実装

8 進と 16 進の両方に対応した myatoi を示します。switch 文は長くなるので個人的には if の方が好きですが、どちらでも書くことはできます。符号は前と同じ、その後「0x」ならその後ろの文字列を 16 進として解釈して累計していきます。数字が 0〜9 と a〜f に分かれているので条件が複雑です。そうでなくて 0 から始まる場合は 8 進で、こちらは単純です。それ以外は十進で前と同じですが、いちいち変数にいれないで直接扱い、for 文で反復を指定しています。

```
#include <stdio.h>
int myatoi(char *s) {
  int sign = 1, val = 0;
  if(*s == '-') { sign = -1; ++s; } else if(*s == '+') { ++s; }
  if(*s == '0' && s[1] == 'x') { // hex
    for(s += 2; *s>='0' && *s<='9' || *s>='a' && *s<='f'; ++s) {
      if(*s >= '0' && *s <= '9') { val = val*16 + (*s - '0'); }
      else { val = val*16 + 10 + (*s - 'a'); }
    }
  } else if(*s == '0') { // ocatl
    for(++s; *s>='0' && *s<='7'; ++s) { val = val*8 + (*s - '0'); }
  } else { // decimal
    for( ; *s>='0' && *s<='9'; ++s) { val = val*10 + (*s - '0'); }
  }
  return sign * val;
}
int main(int argc, char *argv[]) {
  for(int i = 1; i < argc; ++i) { printf("%d\n", myatoi(argv[i])); }
}
```

main ではコマンド引数 1 つずつを変換して表示するようにしました。実行のようすを見ましょう。

```
% ./a.out 10 012 -0x1f
10
10
-31
%
```

確かに大丈夫そうです。

atof の方は数字かどうかを判定する下請け関数を使うようにしてみました。また、指数部を取り出すのに myatoi を呼んでいます (なので指数が 8 進や 16 進で書けますがそれがいやなら前回の例題のままのものを使えばよいです)。

```
#include <stdio.h>
```

```
#include <stdbool.h>
(ここに myatoi をいれる)
bool digit(char c) { return c >= '0' && c <= '9'; }
double myatof(char *s) {
  int sign = 1, scale = 0;
  double val = 0.0;
  for( ; digit(*s); ++s) { val = val*10 + (*s - '0'); }
  if(*s == '.') {
    ++s;
    for( ; digit(*s); ++s) { val = val*10 + (*s - '0'); --scale; }
  }
  if(*s == 'e') { scale += myatoi(s+1); }
  for( ; scale > 0; --scale) { val *= 10; }
  for( ; scale < 0; ++scale) { val /= 10; }
  return sign * val;
}
int main(int argc, char *argv[]) {
  for(int i = 1; i < argc; ++i) { printf("%g\n", myatof(argv[i])); }
}
```

scale は全体に 10 の何乗を掛けるかを保持します (指数部とも言えます) 符号はこれまでと同じ、そのあと小数点の手前部分もこれまでと同じですが、小数点があった場合はその先は val はこれまでと同様に値を累積していきますが、同じだけ scale を減らしていきます。つまり 1.234 は 1234×10^{-3} なのでその -3 を数えるわけです。ここまで終わって次の文字が指数なら指数部ですが、その整数値の取り込みは上述のように myatoi を使い、現在の scale の値に足し込みます。そのあと、scale が正ならその値ぶん繰り返し 10 倍し、負ならばその値ぶん繰り返し 10 で割れば求める値になり、最後に符号を掛けて返します。実行のようすを示します。

```
% ./a.out 1.234 1.5e5 1e-3
1.234
150000
0.001
```

1.3 演習 4 — パターンマッチの拡張

この演習は全部は大変なので分かりやすいもののみ、変更の要点だけ示します。まずパターンを先頭に固定する^ですが、matchstr 中で先頭位置をずらして調べるのを^のときだけやめます。

```
char *matchstr(char *str, char *pat, int len, char **tail) {
  if(len == 0) { return NULL; }
//printf("matchstr: '%s' '%s'\n", str, pat);
  if(*pat == '^') {
    char *t = pmatch(str, pat+1, str + len);
    if(t != NULL) { *tail = t; return str; }
    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;
}
```

残りの機能は基本的に pmatch の中の if-else による分岐を増やして実現します (文字クラスはずっと大変なので扱っていません)。まず$はパターン側が$のとき、文字列側が最後の'\0' であれば成功でその場所を返し、それ以外は失敗とすればよいです。

```
  } else if(pat[0] == '$') {
    return (*str == '\0')? str : NULL;
```

次に*は+とよく似ていますが、0 回のマッチもありなところが違います。

```
  } else if(pat[1] == '*') {
    int i = 0;
    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;
```

つまり、i を「0 から」探し、縮める方も「0 まで」縮めて探します。最後に?ですが、これは「0 回か 1 回」と考えて最初にのばすところで while の代わりに 1 回だけ i を増やすかどうかテストし、あとは同じにします。

```
  } else if(pat[1] == '?') {
    int i = 0;
    if(pat[0] == str[i]) { ++i; }
    for( ; i >= 0; --i) {
      char *t = pmatch(str+i, pat+2, lim);
      if(t != NULL) { return t; }
    }
    return NULL;
```

動かしてみます。

```
% ./a.out 'abcabccaba' 'abc*a'
abcabccaba
^^^^
abcabccaba
   ^^^^^
abcabccaba
       ^^^
% ./a.out 'abcabccaba' '^abc*a'
abcabccaba
^^^^
% ./a.out 'abcabccaba' 'abc*a$'
abcabccaba
       ^^^
% ./a.out 'abcabccaba' 'abc?a'
```

```
abcabccaba
^^^^
abcabccaba
       ^^^
```

2 C 言語の構造体機能

2.1 構造体の概念と定義 exam

構造体 (structure) ないし**レコード** (record) とは、複数の**フィールド** (field) が集まったデータ構造です。複数のデータという点では配列に似ていますが、配列が「同種の」データの並びで、実行時に「何番目」という番号で要素を指定するのに対し、構造体では集まったデータそれぞれが違う型であってよく、そのためフィールドごとに別の名前をつけて扱います。

1 つのプログラムで様々な構造体型を使うこともあるので、どの構造体かを区別するために**タグ** (tag) と呼ばれる名前をつけます。タグの定義と構造体型を用いた変数宣言は分けた方が分かりやすいので、本資料ではそのようにします。その場合、タグの定義は次の形になります。

```
struct タグ名 { 変数定義… } ;
```

最後に「;」があるのに注意してください。次に変数宣言するときはこのタグを指定して次のようにします (つまり「struct タグ名」が型名になるわけです)。

```
struct タグ名 変数名 [= 初期値 ],..  ;
```

初期値を指定する場合、配列と同様、{...} で囲んだ値の並びをフィールドを定義した順番に指定します。気をつける必要があるのは、変数を宣言するときにその中身が分からないと困るので、プログラムの中でタグの定義を先に置かなければならないという点です [1] そして、構造体型の変数を定義した後は、その個々のフィールドは「変数. フィールド名」で読んだり書いたりできます。

では、色の RGB 値を扱う例題を見ていただきます。

```
// color1.c --- handle color struct.
#include <stdio.h>
struct color { unsigned char r, g, b; };
struct color mixcolor(struct color c, struct color d);
void showcolor(struct color c);

int main(void) {
  struct color white = { 255, 255, 255 };
  struct color c1 = { 10, 100, 120 };
  showcolor(c1);
  showcolor(mixcolor(white, c1));
}
struct color mixcolor(struct color c, struct color d) {
  struct color ret = { (c.r+d.r)/2, (c.g+d.g)/2, (c.b+d.b)/2 };
  return ret;
}
void showcolor(struct color c) {
  printf("%02x%02x%02x\n", c.r, c.g, c.b);
}
```

[1] ポインタ変数はアドレスのビット数が分かればよいので、ポインタ変数だけはタグの定義前でも宣言できます。

struct color という構造体は unsigned char つまり 0〜255 の整数を保持する r、g、b という 3 つのフィールドから成ります。関数 mixcolor はこの構造体を 2 つ受け取り 1 つ返します。関数 showcolor はこの構造体を 1 つ受け取り何も返しません。

main ですが、変数 whilte は RGB とも 255 の値の構造体になります。c1 はもうちょっと暗めの中間色です。そして次の 2 行で c1 および c1 と whilte を混合した色を 16 進表示しています。

mixcolor は、struct color 型の変数 ret を定義し、その RGB 値をパラメタとして受け取った 2 つの色の RGB 値それぞれの平均として初期化し、そして ret の値を返します。showcolor は簡単で、受け取った色の RGB 値をそれぞれ 16 進 2 桁ずつで表示します。

このように、C 言語では配列の名前はポインタを意味するため配列をそっくり値として渡したり代入したり返すことのはできないのですが、構造体についてはそっくり値として渡したり代入したり返すことができます。ただし大きいデータに対してやるとそのぶんだけ実行が遅くなるので注意も必要です。では実行例を見ましょう。

```
% gcc color1.c
% ./a.out
0a6478
84b1bb
```

演習 1 上の例題をそのまま打ち込んで実行しなさい。c1 の色は別のものにしてよいです。LMS 上に 16 進 6 桁を入力してその色を表示するページを用意してあるので、それを利用してどんな色か確認すること。OK なら次のような関数を作ってみなさい。

a. 渡された色と白の平均を取って返す関数 `struct color brighter(struct color c)`。
b. 渡された色と黒の平均を取って返す関数 `struct color darker(struct color c)`。
c. 2 つの色と 0.0〜1.0 の値を渡すとその 2 色を指定した比率で混ぜた色を返す関数 `struct color linearmix(struct color c, struct color d, double ratio)`。ratio が 0.5 のときは平均になるので mixcolor と同じになる。
d. 色相 (Hue)、彩度 (Saturation)、明度 (Brightness) を渡すと対応する色を返す関数 `struct color hsbcolor(double h, double s, double b)`。HSB から RGB への計算方法は調べてみてください。
e. パラメタは何も受け取らず、中で乱数を呼び出してランダムに色を生成して返す関数 `struct color randomcolor(void)`。stdlib.h で宣言されているライブラリ関数 `drand48(void)` は呼ばれるごとに $[0.0, 1.0)$ の一様乱数を返すのでよければ使ってください。[2]
f. その他、色を計算する何か面白い関数。

2.2 構造体の配列による表 exam

構造体を使う例として**表** (table) と**表の探索** (table lookup) を取り上げます。プログラミングの分野では表とは「**鍵** (key) となる値を指定して**値** (value) を読み書きでする」ようなものを言います。

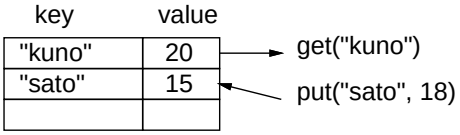

図 1: 表とその概念

[2] drand48 や時間計測を使うプログラムは sol ではコンパイル時に「gcc ファイル.c -std=gnu99 -lrt」というコマンドを使ってください。

たとえば図 1 は、鍵が文字列、値が整数であるような表を示しています。文字列の鍵"kuno"を指定しして取り出すと、対応する値「20」が取れます (表に指定した鍵が入っていない場合は「ない」ことを示す何らかの値が返されることにします)。また鍵"sato" を指定して値「18」を書き込むと、これまでの値の代わりにこの「18」が記録されます (表に指定した鍵が入っていない場合は新たにその鍵と値が追加されますが、表が満杯で追加に失敗することもあります)。

さて、この表はどのようにして実現したらいいでしょうか。すぐ思い付くのが、テーブルの 1 項目を構造体の値とし、それを並べた配列で表すものです。実際に見てみましょう。

```
// tbllinear1 --- table with linear array.
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "tbl.h"
#define MAXTBL 1000000
struct ent { char *key; int val; };
struct ent tbl[MAXTBL];
int tblsize = 0;

int tbl_get(char *k) {
  for(int i = 0; i < tblsize; ++i) {
    if(strcmp(tbl[i].key, k) == 0) { return tbl[i].val; }
  }
  return -1;
}
bool tbl_put(char *k, int v) {
  for(int i = 0; i < tblsize; ++i) {
    if(strcmp(tbl[i].key, k) == 0) { tbl[i].val = v; return true; }
  }
  if(tblsize+1 >= MAXTBL) { return false; }
  char *s = (char*)malloc(strlen(k)+1);
  if(s == NULL) { return false; }
  strcpy(s, k); tbl[tblsize].key = s; tbl[tblsize].val = v;
  ++tblsize; return true;
}
```

なんだんか見慣れない#include があるし main がないですが、とりあえずそれは保留して#define のところから読みます。1 つの項目は文字列の key、整数の val の 2 つのフィールドを持つ struct ent として、それが MAXTBL 個並んだ配列が表です。ただしどこまで入っているか覚えておく必要があるので、そのために変数 tblsize を使用し、初期値を 0 とします。

表を読むときは、値の入っている範囲について順に指定された鍵と等しいかどうか調べていき、等しい項目があれば対応する値を返します。最後まで一致するものがなければ、ここでは「ない」印として -1 を返すことにしました (安易ですが、表には 0 以上だけ入れるという予定で)。

書き込むときは同様に調べて行き、鍵が等しい項目があればそこの val に値を書き込み、「はい」を返します。問題は最後まで一致がなかった場合ですが、このときは新たな項目を追加し、鍵と値を覚えます (ただし tblesize を増やした時に確保した領域を超えるなら追加は失敗なので「いいえ」を返して終わります)。

覚える処理ですが、値は整数だからいいのですが、鍵が文字列なので注意が必要です。文字列は実際は文字へのポインタ型で、たとえば外部から読み込むための領域を指していたりしますよね。そう

すると、その場所だけ覚えていても次の入力のときに中身が書き変わってしまいます。

そこで、**malloc**(memory allocate) という新しい関数が出てきます。この関数は「指定サイズのあき領域を確保して領域の先頭を返す」関数です。これに文字列の長さ+1(末尾のナル文字のぶん) を渡して、戻された値を文字列へのポインタ型にキャストし、変数 s に入れます。もし malloc がメモリ不足で失敗した場合は NULL が返ってくるので、その場合は「いいえ」を返します。OK なら次に sctcpy でパラメタ k にある文字列をここへコピーします。[3] そして、表の key フィールドにはこのポインタ、val には渡された値を入れて、tblsize は増やして「はい」を返します。

この実現のように、列に並んだ値を端から順番に一致を見ていくやり方を**線形探索**と呼びます。表にデータが N 個入っていたとすると、平均して半分の場所で見つかるものとして、見つかる場合の比較数が $\frac{N}{2}$、見つからない場合は比較数が N ですから、見つかる比率がいくらであったとしても、線形探索では 1 項目の get/put にかかる時間計算量は $O(N)$ となります。

2.3 ファイルの分割とヘッダファイル

さて、これをテストする main はどこにあるのでしょう。唐突ですがここで、上で呼んだ表の機能 (要するに get と put で使えます) と、main やその下請け関数とを、別のファイルに分けることにします。両方で合わせなければならないのは、関数 get と put の呼び方だけです。そこで、**ヘッダファイル** tbl.h に次の 2 行 (プロトタイプ宣言) を入れます。

```
bool tbl_put(char *k, int v);
int tbl_get(char *k);
```

これを取り込んでいたのが上の#include "tbl.h"でした。このようにダブルクォートでファイル名を囲んだ場合は、現在位置にあるヘッダファイルを取り込めます (相対パス名や絶対パス名も指定可能)。

```
// tbltest1 --- test table functions
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include "tbl.h"
#define MAXBUF 100
void chopnl(char *s, int lim);

int main(void) {
  char buf[MAXBUF];
  int val;
  while(true) {
    printf("key (empty for quit)> ");
    if(fgets(buf, MAXBUF, stdin) == NULL) { return 1; }
    chopnl(buf, MAXBUF);
    if(strlen(buf) == 0) { return 0; }
    printf("val (-1 for query)> ");
    scanf("%d", &val);
    if(val != -1) { tbl_put(buf, val); }
    else          { printf("tbl[%s] == %d\n", buf, tbl_get(buf)); }
    if(fgets(buf, MAXBUF, stdin) == NULL) { return  1; }
  }
```

[3] 前回注意したにも関わらず長さ上限を指定していませんが、これを呼ぶ側でちゃんとナル文字で終わっている文字列であるように処理するということでご理解ください。

```
}
(chopnl をここに)
```

こちらも同じヘッダファイルを取り込むことで、同じプロトタイプ宣言によるチェックがなされます。さて main ですが、無限ループで繰り返しテストできるようにしています。ループの先頭でプロンプトを出して fgets で 1 行読みます。fgets はエラーや入力終わりのときは NULL を返すのでその場合はそこで終わりにします。先に見た chopnl で改行は削除し、長さ 0 だったら正常終了とします。そうでない場合はプロンプトを出して記録する値を scanf で読みますが、−1 のときは問い合わせて結果を表示、そうでない場合は値の登録を行います。最後にもう 1 回 fgets していますが、これは scanf で数字を読み込むとその後の改行文字はまだ読まれずに残っているので、それを読み捨てて次の入力が行の先頭からになるために必要なのです。

では動かしてみましょう。コンパイル時に 2 つのファイルを指定する点がこれまでとちょっと違います。

```
% gcc tbltest1.c tbllinear1.c -std=c99  ← 2 つ指定してコンパイル
% ./a.out
key (empty for quit)> kuno  ← kuno に 20 を登録
val (-1 for query)> 20
key (empty for quit)> kuno  ← kuno を検索
val (-1 for query)> -1
tbl[kuno] == 20
key (empty for quit)> sato  ← sato を検索
val (-1 for query)> -1      ←未登録
tbl[sato] == -1
key (empty for quit)> sato  ← sato に 15 を登録
val (-1 for query)> 15
key (empty for quit)> sato  ← sato を検索
val (-1 for query)> -1
tbl[sato] == 15
key (empty for quit)> sato  ← sato を変更
val (-1 for query)> 18
key (empty for quit)> sato
val (-1 for query)> -1
tbl[sato] == 18
key (empty for quit)>        ← [RET] で終わる
%
```

演習 2 上の例題をそのまま打ち込んで動かしなさい。動いたら次の変更をしてみなさい。

a. 登録できる値を整数 1 個から変更しなさい (整数 2 個とか文字列とか)。
b. 今は表は追加と書き換えしかできないが、削除機能をつけてみなさい。
c. 表の中身を全部まとめて表示する機能をつけてみなさい。
   (ヒント: この機能そのものは tbllinear1.c の中に置くのが自然で、main からそれを呼び出す。どういう場合にこの機能が呼ばれることにするかは好きに決めてかまいません。)
d. そのほか、面白いと思う機能をつけてみなさい。

2.4 C 言語における時間計測

先に述べたように、線形探索による表は 1 項目あたりの get/put の時間計算量が $O(N)$ です。これを計測によって確認してみましょう。そのために次のような計測プログラムを書きました。表の実現

部分のファイルは変更する必要がないことに注意。

```
// tblbench1 --- benchmark table performance
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include "tbl.h"

int main(int argc, char *argv[]) {
  if(argc != 2) { fprintf(stderr, "need count.\n"); return 1; }
  int count = atoi(argv[1]);
  struct timespec t1, t2;
  clock_gettime(CLOCK_REALTIME, &t1);
  for(int i = 0; i < count; ++i) {
    char buf[100];
    sprintf(buf, "s%d", (int)(drand48()*10000000));
    tbl_put(buf, i+1);
    int k = tbl_get(buf);
  }
  clock_gettime(CLOCK_REALTIME, &t2);
  int msec = 1000*(t2.tv_sec-t1.tv_sec) +
             (t2.tv_nsec-t1.tv_nsec)/1000000;
  printf("%d\n", msec);
}
```

このプログラムは指定した回数だけ乱数で生成した文字列データと数値データを表に put してすぐ同じものを get します。乱数は前に学んだ drand48() を使い、これに百万をかけて整数にしたものを先頭に「s」という文字をつけて文字配列 buf に生成します。**sprintf** というのは printf とそっくりで、ただし整形出力をファイルに出力するかわりに第 1 引数の文字配列に書き込みます。これを鍵として (書く場合は値はその整数値に 1 足したものとして)、get/put を呼びます。

そして、上記の繰り返し処理全体にかかる時間を測ります。そのために、ヘッダファイル<time.h>で定義されているライブラリ関数 clock_gettime を使います。

この関数は第 1 引数として定数 CLOCK_REALTIME(前記ヘッダファイルで定義) を渡すと、その呼んだ時点での 1970.1.1 の 0:00:00 からの経過時間を第 2 引数の構造体で受け取れます。構造体の定義 (これも前記ヘッダファイルにある) は次のようになります。

```
struct timespec {
        time_t  tv_sec;         /* seconds */
        long    tv_nsec;        /* and nanoseconds */
};
```

つまり、非常に細かい値も計測できる場合に備えて、秒数と秒に満たないナノ秒数とを別々に受け取ります。clock_gettime を上記のループの前と後で呼び、両方の時間を引き算して (ナノ秒では細かすぎるので) ミリ秒数に直し、それを表示しています。

では実際に計測してみましょう。[4]

[4] drand48 や時間計測を使うプログラムは sol ではコンパイル時に「gcc ファイル.c -std=gnu99 -lrt」というコマンドを使ってください。

```
% gcc tblbench1.c tbllinear1.c -std=gnu99 -lrt
% ./a.out 1000
9
% ./a.out 2000
36
% ./a.out 3000
85
%
```

このように、データ数が 2 倍、3 倍になると所要時間が 4 倍、9 倍になり、時間計算量が $O(N^2)$ であると分かります。それは当然で、データ数が 2 倍になると、1 回の所要時間が 2 倍になり、その反復回数も 2 倍になるから、掛け算して 4 倍なわけです。

ところでここに、同じ呼び出し方で使える表の別の実装があります。それを計測してみましょう。

```
% ./a.out 1000
0
% ./a.out 10000
5
% ./a.out 20000
10
% ./a.out 30000
15
%
```

このように 1000 では速すぎて計測できず、1 万から 2 倍、3 倍としたときに時間も 2 倍、3 倍となっています。つまり、要素 1 個の get/put にかかる時間は定数 $O(1)$ ということですね。その仕組みはすぐ後で説明します。

演習 3 自分でも線形探索の表を時間計測し、$O(N^2)$ の時間計算量であることを確認しなさい。

2.5 ハッシュ表と動的データ構造

1 回の get/put が $O(1)$ の表はどうやって作れるのでしょうか。1 つのヒントは、普通の配列のアクセス `a[i]` は読むときも書くときも 1 定の時間でよい、ということです。ですから、たとえば鍵が 0〜9999 の整数であれば、その大きさの配列を取れば $O(1)$ の表になります。

しかし今回は鍵が文字列ですから、この方法は使えません。また、実数値や配列が確保できなような大きい整数を鍵にする場合も同様です。そこで、「ある規則にしたがって、文字列 s から一定範囲の整数に変換する」関数を作ります。これを**ハッシュ関数** (hash function) と呼びます。たとえば、文字列`"kuno"`でこの関数を計算して場所を決めて格納したとすれば、後でまた`"kuno"` で検索したときも同じ関数で計算することで場所が分かり、すぐ取り出せるはずです。これが**ハッシュ表** (hash table) の基本的な考えです。

ただし、運が悪いと別の文字列でも関数の計算結果が同じ値になるかも知れません。これを**衝突** (collision) と言います。衝突の対処方法はいくつかありますが、ここでは衝突したときに同じ場所に複数の値が入れられるように単連結リストを使います (図 2)。

ではコードを見てみましょう。ハッシュ表の配列サイズが 9973 となっていますが、これは 1 万を超えない素数を選んだものです。ハッシュ表ではハッシュ関数の値がなるべく「バラバラに」ばらけて衝突が少ないことが重要なので、最後に表の範囲に入れるために表サイズで剰余を取りますが、そのとき素数の剰余の方がばらけやすいと考えているからです。

そして、構造体は単連結リストにするのでフィールドとして鍵、値に加えて次のセルを指す `next` があります。表そのものはセルを指すポインタの配列です。本来ならこの各要素に `NULL` を入れるべ

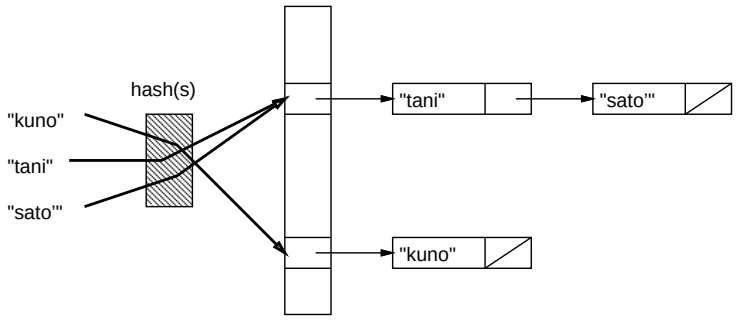

図 2: 単連結リストを使うハッシュ表

きですが、C 言語ではグローバル変数には数値なら 0、ポインタなら NULL が入っていることになっているので、初期化は省略しています。

次にハッシュ関数ですが、先頭に static とついています。これはこのファイル内だけで有効な関数という意味で、外部の関数と名前が衝突しないので短い名前を安心して使うことができます。あと、返す値も作業変数 v も符号なし整数です (マイナスの数が出て来ると扱いずらいため)。v の初期値は 1 で、文字列の 1 文字ごとにその文字コードを 11 倍して 1 を足したものを掛け算します。理由は説明しにくいのですが、素数倍がばらけやすいからとか、次々に掛けていくときたまたま 0 になって以後 0 のままになるのを防ぐため 1 足すとか、そういうことを考えています。最後まで掛け算したら上記のようにサイズで剰余を取って返します。

```
// tblchash --- table impl. with chained hash.
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "tbl.h"
#define MAXTBL 9973
struct ent { char *key; int val; struct ent *next; };
struct ent *tbl[MAXTBL];

static unsigned int hash(char *s) {
  unsigned int v = 1;
  while(*s) { v *= 11 * (*s++) + 1; }
  return v % MAXTBL;
}
static struct ent *lookup(struct ent *p, char *k) {
  for( ; p != NULL; p = p->next) {
    if(strcmp(p->key, k) == 0) { return p; }
  }
  return NULL;
}
int get1(struct ent *p, char *k) {
  struct ent *q = lookup(p, k);
  return (q == NULL) ? -1 : q->val;
}

static bool put1(struct ent **p, char *k, int v) {
  struct ent *q = lookup(*p, k);
```

```
  if(q != NULL) { q->val = v; return true; }
  q = (struct ent*)malloc(sizeof(struct ent));
  if(q == NULL) { return false; }
  int len = strlen(k);
  q->key = (char*)malloc(len+1);
  if(q->key == NULL) { return false; }
  strcpy(q->key, k);
  q->val = v; q->next = *p; *p = q; return true;
}
int tbl_get(char *k) { return get1(tbl[hash(k)], k); }
bool tbl_put(char *k, int v) { return put1(&tbl[hash(k)], k, v); }
```

外部から呼ぶ getr/put は一番下の 2 行です。これらは、配列中のハッシュ関数で計算した場所の値 (put では場所のアドレス) と鍵 (と put では値) を持って下請け関数を呼びます。

次は lookup を読みましょう。for 文ですが、初期化のところは使わないので空になっていて、ポインタ値 p が NULL でない間くりかえし、次のセルをたどって行きます。この「p->next」とは? これは**アロー演算子** (arrow operator) と呼び、意味は「(*p).next」とまったく同じですが、少し読みやすくなっているので積極的に使います。たどっていく中で key のフィールドが渡された k と等しい文字列なら、そのセルのポインタを返します。後まで見つからなかったら NULL を返します。

では次に下請け関数 get1 ですが、これは looup を呼んで結果が NULL なら -1、そうでなければ返されたセルの val フィールドを返すだけです。

最後に put1 ですが、これは第 1 引数がポインタのポインタになっています。なぜかというと、新たなセルを作るためにその場所に値を入れる必要が生じるかも知れないためです。lookup にも*p を渡し、返った結果が NULL でなければその値のセルがあったので、val フィールドに値を入れて「はい」で返ります。見つからなかったのなら、新たなセルが必要です。malloc でセルの領域を割り当て、変数 q に入れます。もし NULL なら失敗で返ります。OK なら今度は文字列の領域を割り当てて q->key に入れます。これも NULL なら失敗で返ります。OK なら文字列をコピーし、値を入れます。そして next にはこれまでの*p を入れ、最後に*p にこの q を入れれば、新たなセルが単連結リストの先頭に挿入されたことになります。

演習 4 このハッシュ表の実装を入力し、先の計測プログラムと一緒にして計測してみなさい。回数が多くなると $O(1)$ でなくなりますが、その理由を検討し、その問題を解消するような変更を行ってみなさい。

(ヒント: 多く登録しすぎるとそうなるので、登録数を調べて一定以上」登録しようとしたら満杯ということにするのが方法の 1 つです。)

演習 5 連結リストを使わないでハッシュ表を作る次の方法があります (図 3)。

- エントリは「値が入っている」「入っていない」を区分できるようにする (文字列が鍵なら NULL ポインタのとき入っていないということにすればよい)。
- ハッシュ関数を 2 つ用意する。
- 登録時は 1 つ目のハッシュ関数で選んだ位置 i に衝突があったら、2 つ目のハッシュ関数で値 d を計算し、$i+d$、$i+2d$、…を調べていって空いている位置にいれる。検索時は同様にして空いている位置に来たら登録されていないことが分かる。

なお、ハッシュ表の端まで来たら先頭に戻ることにします。また、ハッシュ表のサイズを素数にしておくことで、同じ $i+d$ に戻って来てしまうことがなくせます。表が満杯だと検索が止まらなくなるので、いくつ入れたか数えて管理することは必須です。この方法を**ランダムリハッ**

シュ(random rehash) と呼びます。この方法のハッシュ表を実現してみなさい。性能計測もすること。

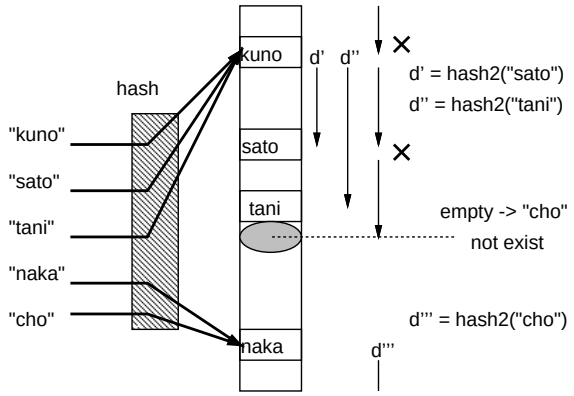

図 3: ランダムリハッシュ法

本日の課題 14A

「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 言語の構造体機能についてどのように思いましたか。

Q2. C 言語でファイルを複数に分ける方法が分かりましたか。

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 14B

「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. 表と検索とはどういうことか理解しましたか。

Q2. 線形探索の表とハッシュ表の計算量について納得しましたか。

Q3. 課題に対する感想と今後の要望をお書きください。