I117（13）CSV （その3）

知念

北陸先端科学技術大学院大学 情報科学研究科
School of Information Science,
Japan Advanced Institute of Science and Technology

CSV データの活用

csvparse.c を関数群をまとめたファイルとして扱う

- csvparse.c を libcsv.c という名前に変える
- コールバック（callback）を呼び出す形式にする
  - readfield_callback
- main() を取り除く

辞書プログラムを改造

- コールバックを登録
- 特定フィールドを格納して各種処理を行う

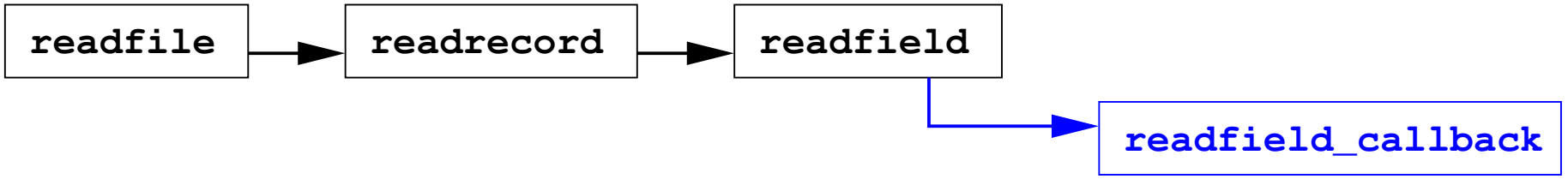
コールバック

特定条件で呼び出す処理をコールバック（callback）と呼ぶ
今回は readfield() の最後で関数を呼び出す

```
...
if(readfield_callback) {
  int ck;
  ck = readfield_callback(rno, fno, value);
  if(ck<0) {
    fprintf(stderr, "callback error\n");  }
}
...
```

コールバックを導入する理由

- 関数群は構造に沿って何段も潜って行く
- main() からフィールド解析を呼ぶのは困難
- 逆に、フィールド解析に達した際に、
  あらかじめ登録した関数を呼び出す形態にする



初期値は NULL、デフォルトでは作動しない

```
int (*readfield_callback)(int,int,char*)=NULL;
```

活用プログラム — 単なる格納

- 定義通り、引数が int, int, char* の関数を作る
- readfield_callback に代入する

```
int savefield(int rno, int fno, char *cont) {
    if(fno!=1) { return 0; }
    if(dictuse>=dictlen) {  expand();  }
    dict[dictuse].value = strdup(cont);
    dict[dictuse].count = 1;  dictuse++;
    return 0;  }
...
    readfield_callback = savefield;
```

今回は第2フィールド（fno==1）のみ格納

活用プログラム — 単なる格納 *(cont.)*

あとは readfile() を呼び出すだけ

```
readfile();
```

結果

```
% ./pcrlf <smp1
aaa,bbb,ccc CRLF
zzz,yyy,xxx CRLF
% ./a.out < smp1
 0 1 bbb
 1 1 yyy
```

活用プログラム — 重複除く

例によって、bsearch と qsort を使う

```
dict_t *ppos, ref;
ref.value = cont;
ppos = bsearch(&ref, dict, dictuse,
            sizeof(dict[0]), dictcmp);
if(!ppos) { /* not found */
  if(dictuse>=dictlen) {  expand();  }
  dict[dictuse].value = strdup(cont);
  dict[dictuse].count = 1;  dictuse++;
  qsort(dict, dictuse, sizeof(dict[0]), dictcmp); }
else {  ppos->count++;  }
```

活用プログラム — 重複除く *(cont.)*

実行結果（その1）

```
% cat smp1 smp4 | ./pcrlf
aaa,bbb,ccc CRLF
zzz,yyy,xxx CRLF
aaa,bbb,ccc
% cat smp1 smp4 | ./a.out
no LF (record separator)
 0 1 yyy
 1 2 bbb
```

活用プログラム — 重複除く *(cont.)*

実行結果（その2）

```
% ./pcrlf < vv.csv
"野口英世",1000 LF
"樋口一葉",5000 LF
"福沢諭吉",10000 LF
"夏目漱石",1000 LF
"新渡戸稲造",5000 LF
"聖徳太子",10000
```

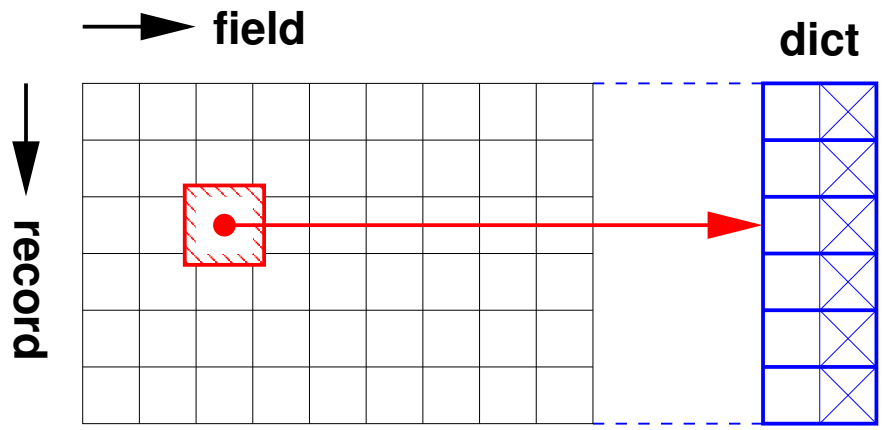
⇒

```
% nkf -e vv.csv | ./a.out
no LF (record separator)
 0 2 1000
 1 2 10000
 2 2 5000
```

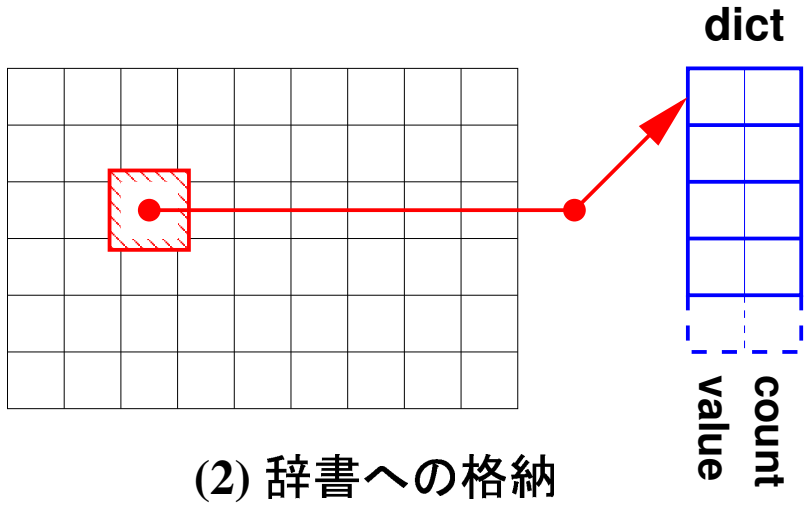
数字部分を抜きだして頻度計上

※ 文字列比較なので 10000 が 5000 の先に出る

活用プログラム — 重複除く *(cont.)*



(1) 辞書への機械的格納



(2) 辞書への格納

- 1つ目はレコードと同数の辞書容量が必要
- 2つ目は重複を省くので辞書容量は減る
  - ◇ 減る程度は内容による

レコード単位の処理

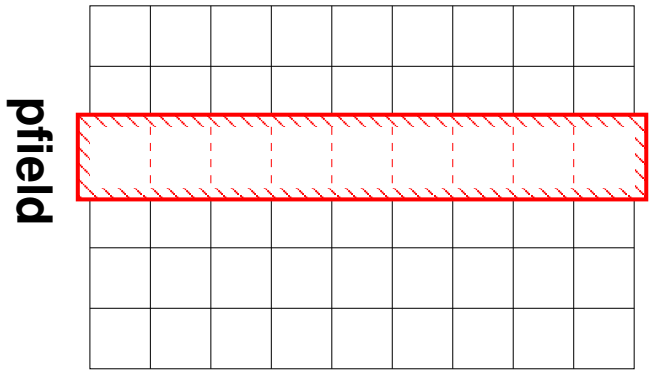
全てのフィールド内容を保存する
レコードをまたがず、レコード毎に消去

```
#ifndef MAXFIELD
#define MAXFIELD    (10)
#endif

char *pfield[MAXFIELD];
int  numfield=-1;
```

レコード（フィールド集合）を配列に格納
ひとまず、10個用意（可変長にする方法は既に示した）

レコード単位の処理 *(cont.)*



(3) レコードを配列に格納

任意のフィールドに同時にアクセス可能

- 先の例は固定の1フィールドしか扱えない

配列分のメモリを消費

レコード単位の処理 *(cont.)*

フィールド解析時の処理を変更

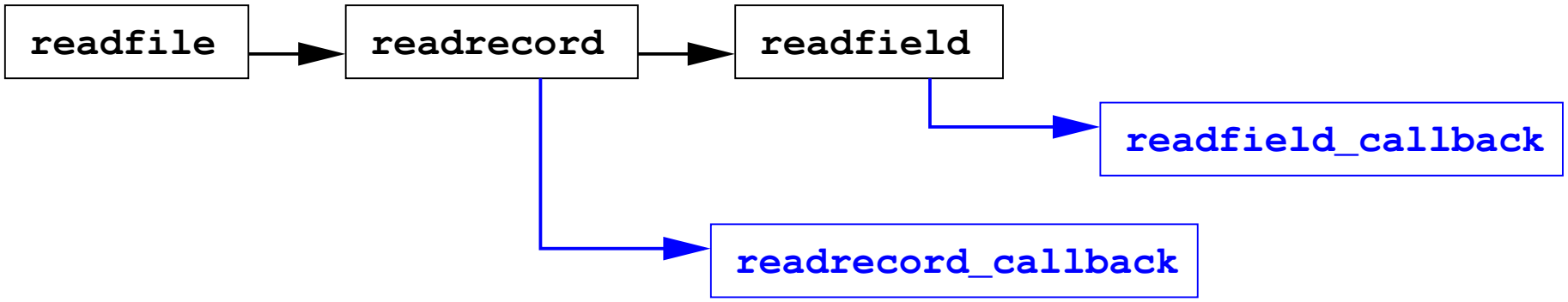
```
int savefield(int rno, int fno, char *cont) {
  if(fno<MAXFIELD) {
    if(pfield[fno]) {  free(pfield[fno]); }
    pfield[fno] = strdup(cont);
    if(fno+1>numfield) { numfield = fno+1; }
  }
  else {
      fprintf(stderr, "over-run field range\n");
  }
  return 0; }
```

レコード解析時のコールバックを設ける

```
int (*readrecord_callback)(int)=NULL;
```

readrecord () 末尾でコールバックを呼ぶ

```
if(readrecord_callback) {
  ck = readrecord_callback(rno);
  if(ck<0) {
    fprintf(stderr, "callback error\n");
  } }
```



レコード単位の処理 *(cont.)*

具体的コールバック関数 — 単純表示

```
int saverecord(int rno){
  int i;
  printf("record %d\n", rno);
  for(i=0;i<numfield;i++) {
      printf("  %2d: ’%s’\n",
          i, pfield[i] ? pfield[i] : "*NULL*" );
  }
  clearfield();  return 0;
}
```

clearfield() は pfields 内容を消去する関数

レコード単位の処理 *(cont.)*

レコードが切り替わる際に各フィールド内容を消去

```
int clearfield() {
  int i;
  for(i=0;i<MAXFIELD;i++) {
    if(pfield[i]) {
      free(pfield[i]);  pfield[i] = NULL;
    }
  }
  numfield=-1;
}
```

レコード単位の処理 *(cont.)*

実行結果

```
record 0
   0: ’野口英世’
   1: ’1000’
record 1
   0: ’樋口一葉’
   1: ’5000’
record 2
   0: ’福沢諭吉’
   1: ’10000’
...(略)
```

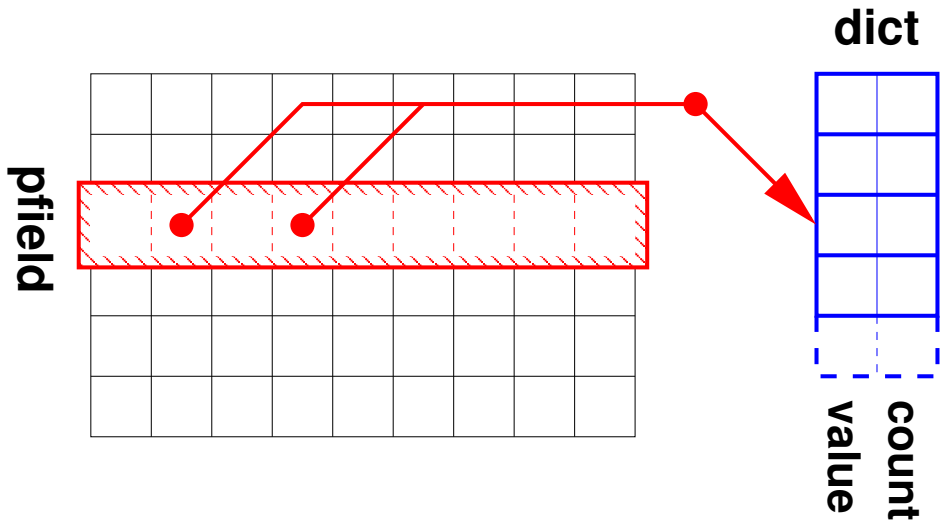
レコード単位の処理 *(cont.)*

簡単な出納処理コールバック
各人物の数字を加減する

```
野口英世,40
樋口一葉,50
福沢諭吉,10
野口英世,100
福沢諭吉,-70
樋口一葉,150
福沢諭吉,100
```

例によって dict_t の辞書を流用

レコード単位の処理 *(cont.)*



(4) フィールド内容を辞書へ格納

注目レコードと辞書のメモリを消費

```
int saverecord(int rno) {
    dict_t *ppos, ref;
    int adiff = atoi(pfield[1]);
    ref.value = pfield[0];
    ppos = bsearch(&ref, dict, dictuse,
                sizeof(dict[0]), dictcmp);
    if(!ppos) { /* not found */
        if(dictuse>=dictlen) { expand(); }
        dict[dictuse].value = strdup(pfield[0]);
        dict[dictuse].count += adiff;  dictuse++;
        qsort(dict, dictuse,
                sizeof(dict[0]), dictcmp); }
    else { ppos->count += adiff; }
    clearfield(); return 0;  }
```

レコード単位の処理 *(cont.)*

実行結果

```
0 40 福沢諭吉
1 140 野口英世
2 200 樋口一葉
```

- 各人物の出納が計算されている
- 負の値も扱えている

全レコード格納

シーケンシャルアクセスはここまでの格納方法で十分

- 全体を一回読むだけ
- 一部を格納して演算処理

ランダムアクセスは全レコード格納が必要

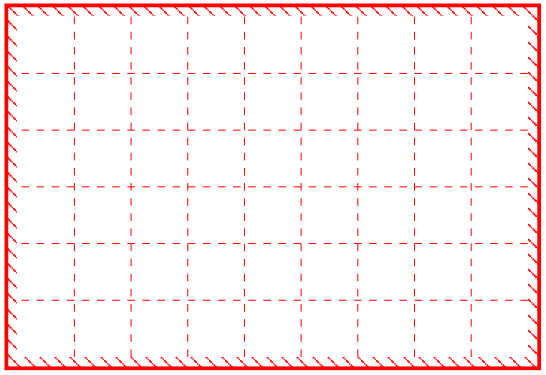
- 行列計算など

ファイル上はフィールドとレコード両方が可変長

- 巨大な二次元配列をつくるのか
- 他に手はないのか

全レコード格納 *(cont.)*

巨大2次元配列



(5) 全レコードを2次元配列に格納

- 無駄が多い
- よほどのことがない限り採用しない

```
#define MAXRECORD   (10000)
#define MAXFIELD    (100)
char *table[MAXRECORD][MAXFIELD];
...
int savefield(int rno, int fno, char *cont) {
    if((rno<0 && rno>=MAXRECORD) ||
        (fno<0 && fno>=MAXFIELD))
        return -1;
    if(table[rno][fno])
        free(table[rno][fno]);
    table[rno][fno] = strdup(cont);
    return 0;  }
```

全レコード格納 *(cont.)*

レコードやフィールドが不足時の拡張は大きな手間
m*n 行列を u*v 行列に

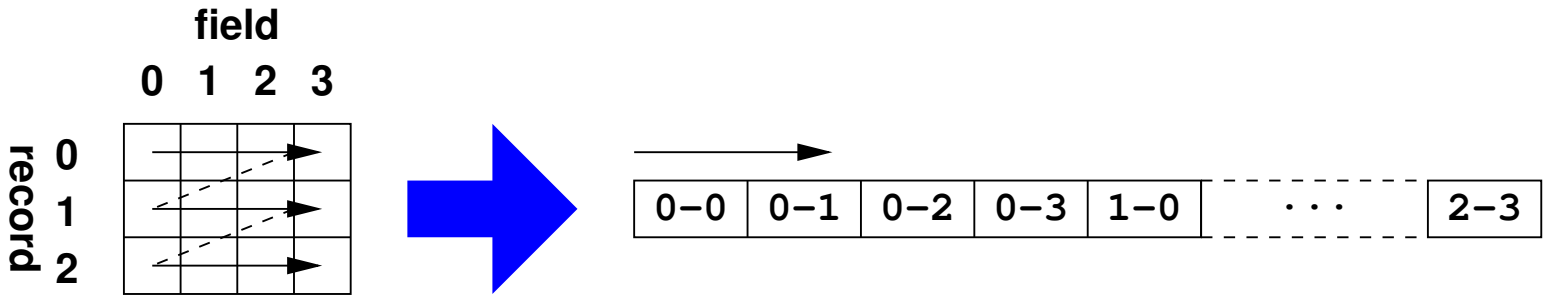
- u*v 行列をつくる
- m*n 行列の内容を u*v にコピー
- m*n を開放

2次元なので、これまでの辞書の拡張とは段違い

辞書で全レコード格納

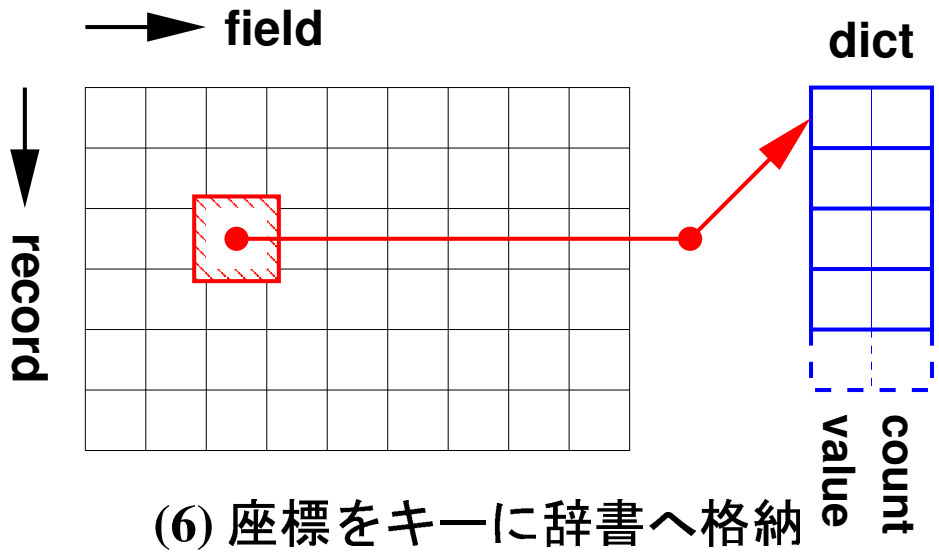
レコード番号とフィールド番号を組み合わせた文字列をキーにして辞書に蓄えれば...

- 2 次元が 1 次元に
- 可変長に対応



※ awk 等が多次元配列に同様の処置を施している

辞書で全レコード格納 *(cont.)*



座標からのキー文字列生成は簡単

```
    sprintf(name, "%d-%d", rno, fno);
```

readfield_callback はこうなる

```
int savefield(int rno, int fno, char *cont) {
    char   name[BUFSIZ];
    dict_t *ppos, ref;
    sprintf(name, "%d-%d", rno, fno);
    ref.value = name;
    ppos = bsearch(&ref, dict, dictuse,
                sizeof(dict[0]), dictcmp);
    if(!ppos) { /* not found */
        if(dictuse>=dictlen) {  expand();  }
        dict[dictuse].value = strdup(name);
...
```

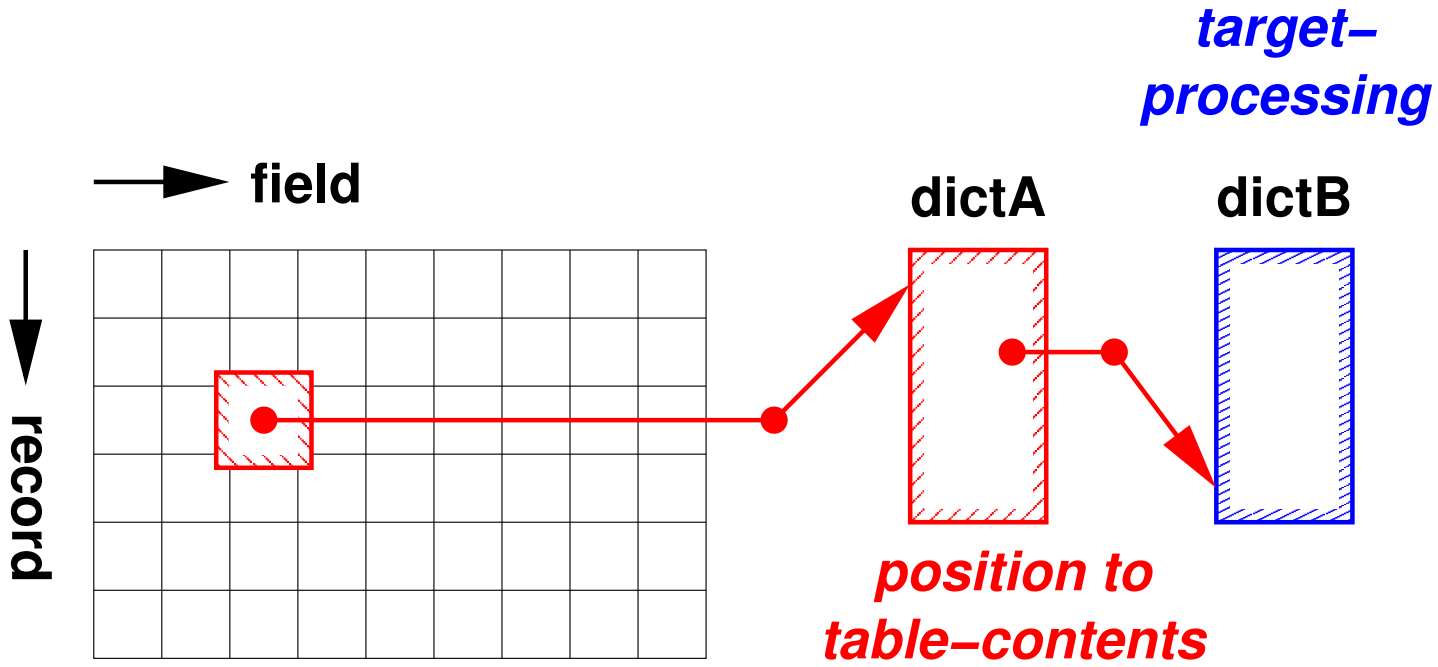
辞書で全レコード格納 *(cont.)*

```
% nkf -e bamount.csv | ./a.out | pr -2 -t -w 38
#expand 10 -> 20
no LF (record separator)
 0 1 0-0                  7 1 3-1
 1 1 0-1                  8 1 4-0
 2 1 1-0                  9 1 4-1
 3 1 1-1                 10 1 5-0
 4 1 2-0                 11 1 5-1
 5 1 2-1                 12 1 6-0
 6 1 3-0                 13 1 6-1
```

※ メモリが節約できているわけではなく、無闇に消費しないということ

辞書で全レコード格納、具体的処理

出納計算など目的の処理につかう辞書と区別が必要



(7) 座標をキーに格納した辞書から目的の処理を施す

辞書で全レコード格納、具体的処理 *(cont.)*

データ構造や関数群を整理する

- キーと内容が一緒ではない場面が多い
  - key と value をメンバとする
- value の型を整数向けと文字列向けに分ける
  - idict と sdict と呼ぶ
- len や use を内包する型を作る
  - idict_a と sdict_a とする
  - 辞書の内容の型は idict_c と sdict_c に

データ型

```
typedef struct {
    char *key;
    int   value;
} idict_c ;

typedef struct {
    idict_c *slot;
    int      len;
    int      use;
} idict_a;
```

```
typedef struct {
    char *key;
    char *value;
} sdict_c ;

typedef struct {
    sdict_c *slot;
    int      len;
    int      use;
} sdict_a;
```

関数

```
int idict_keycmp(const void *, const void *);
int idict_valuecmp(const void *, const void *);
int idict_init(idict_a*);
idict_a *idict_new();
int idict_expand(idict_a*);
int idict_print(idict_a*);
idict_c *idict_findpos(idict_a*, char *xkey);
int idict_add(idict_a*, char *xkey, int xval);
```

同様に sdict 分も用意、以下例外

```
int sdict_add(idict_a*, char *xkey, char* xval);
```

CSV 読み込み — sdict （位置→データ）への格納

```
idict_a *idict;  sdict_a *sdict;
int maxrno=-1, maxfno=-1;
int
savefield_sdict(int rno, int fno, char *cont) {
    char   name[BUFSIZ];  sdict_c *ppos;
    if(rno>maxrno) { maxrno = rno; }
    if(fno>maxfno) { maxfno = fno; }
    sprintf(name, "%d-%d", rno, fno);
    ppos = sdict_findpos(sdict, name);
    if(!ppos) { sdict_add(sdict, name, cont); }
    return 0;  }
```

sdict （位置→データ）からの取り出し

```
int tableref(char *dst, int r, int f) {
    char name[BUFSIZ];  sdict_c *ppos;
    sprintf(name, "%d-%d", r, f);
    ppos = sdict_findpos(sdict, name);
    if(!ppos) {
        fprintf(stderr,
          "#not found data %d-%d\n", r, f);
        return -1;
    }
    strcpy(dst, ppos->value);
    return 0;  }
```

テーブルを走査して出納処理、idict 操作

```
int bamount_proc() {
    int ck, r, am;  idict_c *ipos;
    char name[BUFSIZ], amstr[BUFSIZ];
    for(r=0;r<=maxrno;r++) {
        ck = tableref(name, r, 0);
        if(ck<0) {  continue;  }
        ck = tableref(amstr, r, 1);
        if(ck<0) {  continue;  }
        am = atoi(amstr);
        ipos = idict_findpos(idict, name);
        if(ipos) {  ipos->value += am;  }
        else {      idict_add(idict, name, am);  }
    } }
```

idict,sdic 初期化の後 CSV を読み込んで、出納処理

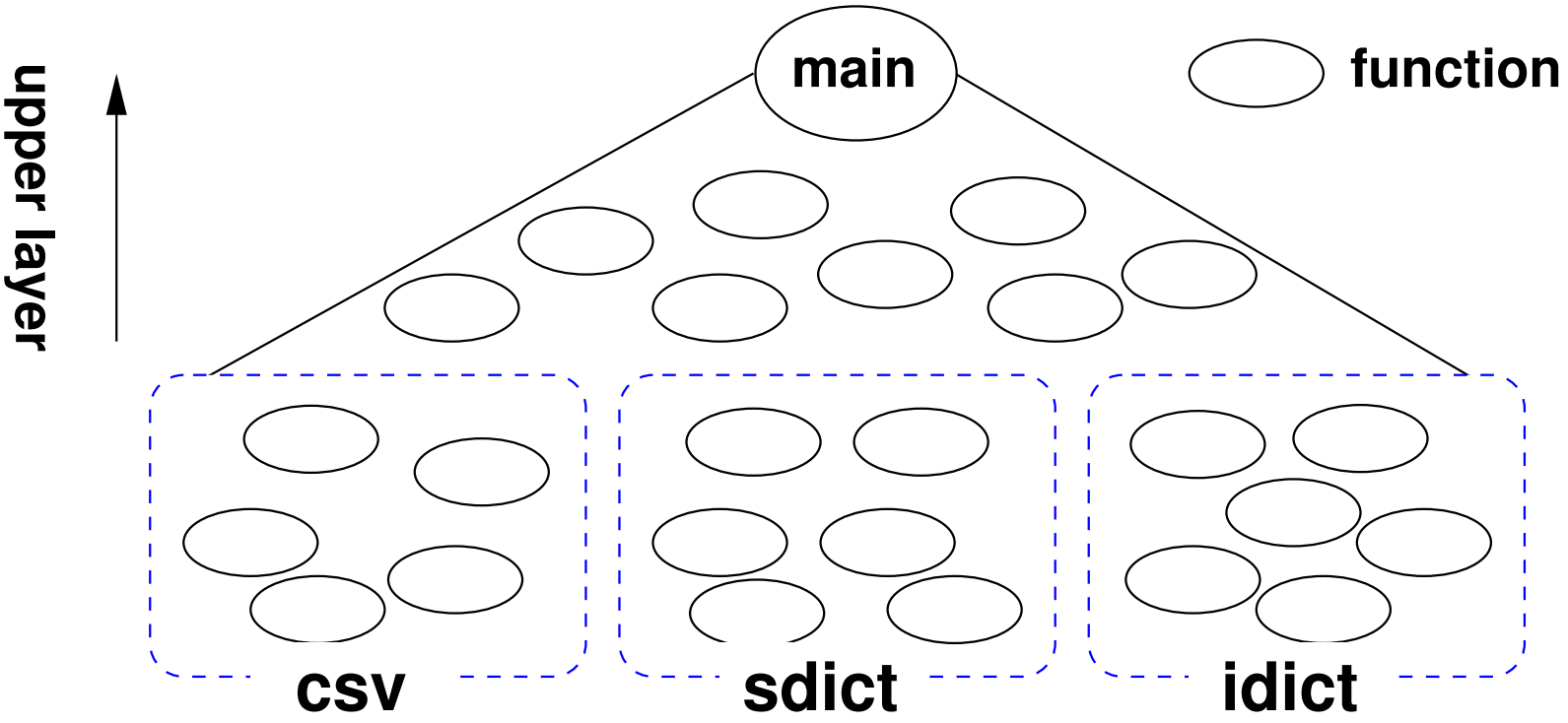
```
int main() {
    int   ck;

    idict = idict_new();  idict_init(idict);
    sdict = sdict_new();  sdict_init(sdict);

    readfield_callback = savefield_sdict;
    ck = readfile();

    bamount_proc();
    idict_print(idict);
}
```

関数階層構造



csv,sdict,idict 間に依存関係はない

辞書で全レコード格納、具体的処理 *(cont.)*

実行結果

```
% nkf -e bamount.csv | ./a.out
#sdict expand 10 -> 20
no LF (record separator)
 0 樋口一葉 200
 1 福沢諭吉 40
 2 野口英世 140
```

まとめ

CSV パース結果を活用

シーケンシャルアクセス向け

- カレントフィールドを格納
- カレントレコードを格納
- 単純格納
- 重複省き格納
- 出納計算

まとめ *(cont.)*

ランダムアクセス向け、全レコード格納

- 単純巨大2次元配列へ格納
- 座標をキーに辞書へ格納
  - ◇ 2次元の可変長を回避
- 単純格納
- 重複省き格納
- 出納計算
  - ◇ 辞書を2段使用

補足

libcsv.c をより本格的するなら

- 各種変数名変更
  現行の変数名は安直、他と衝突する可能性がある
- 関数名変更
  他と衝突する可能性がある
  csv_ を付けるなど工夫する

位置と値を辞書に格納するのは疎な配列で非常に有効

- 密な配列ではさほど効果が無い
- 今回は大きさが不明なので採用

補足 *(cont.)*

「辞書」

- 辞書と「連想リスト」、「連想配列」はほぼ同じ
- 「連想記憶」と呼ぶこともある
- 「ハッシュ」は格納方式やそれに登場する位置計算方式の名前
  - オープンハッシュ
  - クローズハッシュ
  - オープンチェーン

演習

1) 3つ以上フィールドを使う処理の例を考え、プログラムを作れ★
2) テーブル上を何度も往復する処理の例を考え、プログラムを作れ★
3) 最後の2段辞書を使った出納プログラムを実際に作れ★★
4) 先のプログラムでテーブル上の位置をキーに変換する際の区切り文字を、ハイフンではなく、通常入力されない文字に変更せよ★

演習 *(cont.)*

5) sdict, idict に似せて実数(real)を蓄える辞書 rdict の型定義や関数群を作り、そのサンプルプログラムを作れ★
   - 実数を蓄える型は double とせよ
6) n次元空間上のデータを辞書に格納するプログラムを作れ（$n \geq 3$）★

演習 *(cont.)*

7) 矩形（長方形）を辞書で扱うプログラム
   a) 左下と右上の頂点の座標（4つの数）を格納する辞書を作れ★★
   b) ある座標を与えるとその座標を含んだ矩形を取り出す関数を作れ★★
   c) ある座標を与えるとその座標に直近の矩形を取り出す関数を作れ★★
      - 距離はユークリッド距離とする