# I117（14）文字列処理 (4)

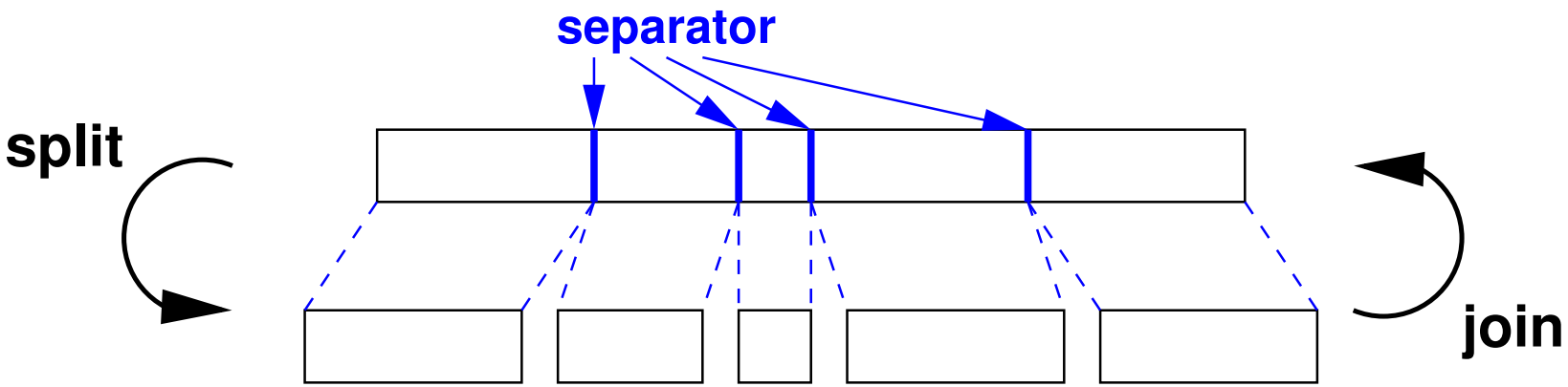
知念

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

# split & join

多数のデータを扱うため、データ操作関数を作る

- split: 文字列を区切り文字で分割
- join: 多数の文字列を区切り文字で連結




典型的なテキスト操作

# split & join *(cont.)*

例

```
"ab,cd,ef" split => "ab" "cd" "ef"
"ab,cd,ef" <=  join "ab" "cd" "ef"
```

処理単位で考えると、文字列と文字列群の操作
文字列群をレコードとして管理する型 rec_t を設ける

```
typedef struct {
    int len;
    int used;
    char **fields;
} rec_t;
```

# split & join *(cont.)*

レコードを扱う関数群をまとめて libspji.c とする

関数群

| | |
|---|---|
| 初期化 | int rec_init(rec_t*) |
| 内容除去 | int rec_clear(rec_t*) |
| 内容表示 | int rec_print(rec_t*) |
| メモリ領域拡張 | int rec_expand(rec_t*, int) |
| フィールド代入 | int rec_setfield(rec_t*, int, char*) |
| 分割 | int rec_split(rec_t*, char*, int) |
| 連結 | int rec_join(rec_t*, char*, int, int) |

前半はこれまでの講義から自明、それ以外を説明

```
int rec_setfield(rec_t*rc, int pos, char*cont) {
  while(pos>=rc->len) { rec_expand(rc, rc->len); }
  if(rc->fields[pos]) { free(rc->fields[pos]); }
  rc->fields[pos] = strdup(cont);
  if(!rc->fields[pos]) {
    fprintf(stderr, "rec_setfield: no memory\n");
    return -1;
  }
  if(pos+1>rc->used) { rc->used = pos+1;  }
  return 0;
}
```

指定場所に保存、保存場所がなければ領域拡張

```
int rec_split(rec_t*rc,char*src,int sep) {
  int   c=0, fno=0;
  char  tmp[BUFSIZ], *p=src, *q=tmp;
  while(*p && c<BUFSIZ) {
    if(*p==(char)sep) {
      *q = '\0';  rec_setfield(rc, fno, tmp);
      q = tmp;  fno++;  p++;  }
    else {
      *q++ = *p++;  c++;  }
  }
  *q = '\0';  rec_setfield(rc, fno, tmp);
  q = tmp;  fno++;
  return fno;  }
```

```
int rec_join(rec_t*rc,char*dst,int dlen,int sep){
    int i, c=0;  char *p, *q=dst;
    for(i=0;i<rc->used;i++) {
        if(rc->fields[i]) {
            p = rc->fields[i];
            while(*p && c<dlen) {
                *q++ = *p++;  c++;   }
        }
        if(i<rc->used-1 && c<dlen) {
            *q++ = (char)sep;  c++;  }
    }
    *q = '\0';
    return rc->used;  }
```

split 利用例

```
#include "libspji.c"
int main(){
    rec_t record ;
    char *msg="ab,cd,ef";
    rec_init(&record);
    rec_split(&record, msg, ',');
    printf("\"%s\" => ", msg);
    rec_print(&record);
}
```

```
% ./a.out
"ab,cd,ef" => {"ab","cd","ef",}
```

split & join 利用例

```
#include "libspji.c"
int main() {
  rec_t record;
  char *msg="ab,cd,ef", buf[BUFSIZ];
  rec_init(&record);
  rec_split(&record, msg, ',');
  rec_join(&record, buf, BUFSIZ, '|');
  printf("\"%s\" <= ", buf);
  rec_print(&record);  }
```

```
% ./a.out
"ab|cd|ef" <= {"ab","cd","ef",}
```

# テキストファイルへ適用

テキストファイルに応用すると便利

```
ab,cd,ef
```

行毎に split を適用すると各フィールドが取り出せる

```
while(1行取り出す) {
    split
    レコード表示
}
```

```
rec_t record;  int nf;
char src[BUFSIZ], dst[BUFSIZ];

rec_init(&record);
while(fgets(src, BUFSIZ, stdin)) {
    chomp(src);
    rec_clear(&record);
    nf = rec_split(&record, src, ',');
    printf("in  %d %s\n", nf, src);
    rec_print(&record);
    nf = rec_join(&record, dst, BUFSIZ, '|');
    printf("out %d %s\n", nf, dst);
}
```

# テキストファイルへ適用 *(cont.)*

- 各行をレコードとしてフィールドに分割(split)
- 全フィールドを別の区切り文字で文字列に直して表示(join)

実行結果

```
% cat x2
ab,cd,ef
```

⇒

```
% ./a.out < x2
in  3 ab,cd,ef
{"ab","cd","ef",}
out 3 ab|cd|ef
```

区切り文字変更 コンマ→縦棒

# テキストファイルへ適用 *(cont.)*

少し難しい例

- 空フィールド

```
% cat x3
aaa,,ccc,
xxx,yyy,zzz
```

⇒

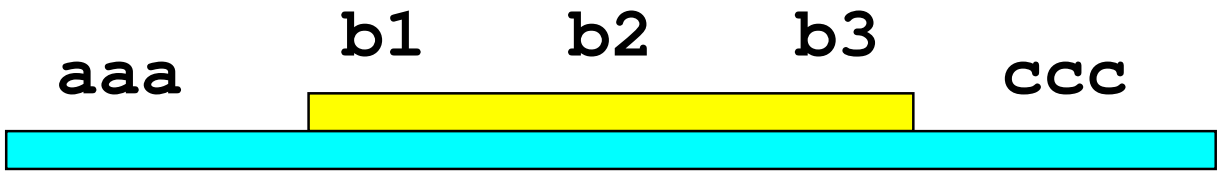
```
% ./a.out < x3
in  4 aaa,,ccc,
{"aaa","","ccc","",}
out 4 aaa||ccc|
in  3 xxx,yyy,zzz
{"xxx","yyy","zzz",}
out 3 xxx|yyy|zzz
```

# ネスト（入れ子）

レコードの中にレコードを含める
内側のレコードでは別区切り文字を使う

```
aaa,b1:b2:b3,ccc
```

複雑なデータが格納できる、模式的に図にすると




内部を含めて区切り文字を変更する

```
rec_t rec2;  int nf2;  char dst2[BUFSIZ];
rec_init(&rec);  rec_init(&rec2);
while(fgets(src, BUFSIZ, stdin)) { chomp(src);
    rec_clear(&rec);  rec_clear(&rec2);
    nf = rec_split(&rec, src, ',');
    printf("in  %d %s\n", nf, src);
    nf2 = rec_split(&rec2, rec.fields[1], ':');
    nf2 = rec_join(&rec2, dst2, BUFSIZ, '#');
    free(rec.fields[1]);
    rec.fields[1] = strdup(dst2);
    nf = rec_join(&rec, dst, BUFSIZ, '|');
    printf("out2 %d %s\n", nf, dst);
}
```

# ネスト（入れ子） *(cont.)*

```
% ./a.out < x4
in  3 aaa,b1:b2:b3,ccc
out2 3 aaa|b1#b2#b3|ccc
```

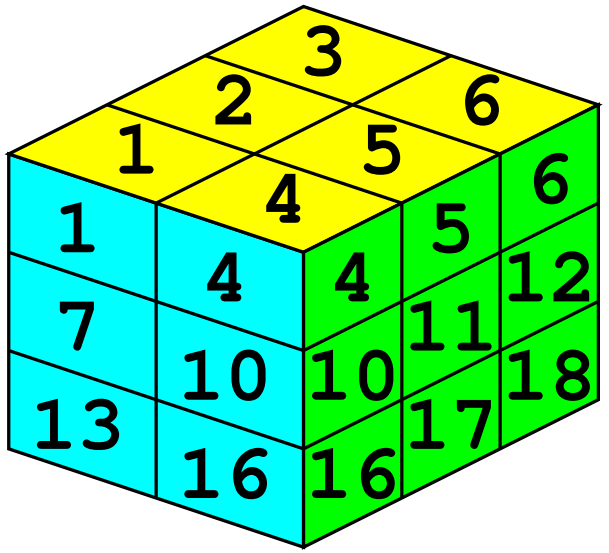
- 区切り文字を変更
  - レベル1: コンマ→縦棒
  - レベル2: コロン→シャープ

# ネスト（入れ子） *(cont.)*

多次元データが格納できる

- 直観的には 3次元データが分かりやすい
  以下の例は 2 x 3 x 3 配列

```
1:2:3,4:5:6
7:8:9,10:11:12
13:14:15,16:17:18
```




- 4次元以上も可能だが人間が理解しずらい

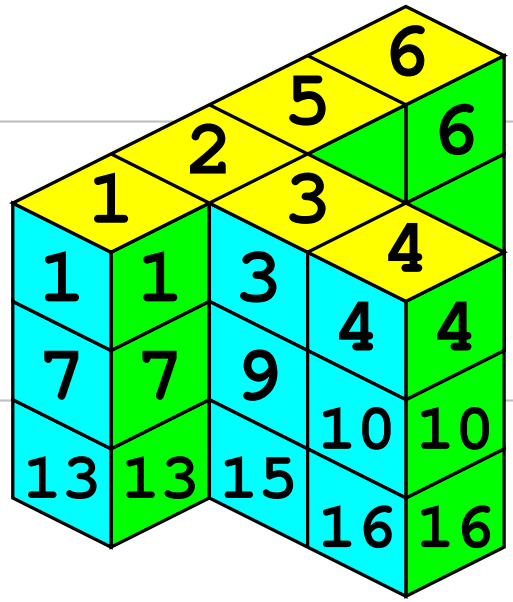
# ネスト（入れ子） *(cont.)*

## 4次元データ、３ｘ２ｘ２ｘ２ 配列

```
1-2:3-4,5-6:7-8,9-10:11-12
13-14:15-16,17-18:19-20,21-22:23-24
```

## 三次元で一部が飛び出した形状

```
1,2:3:4,5,6
7,8:9:10,11,12
13,14:15:16,17,18
```

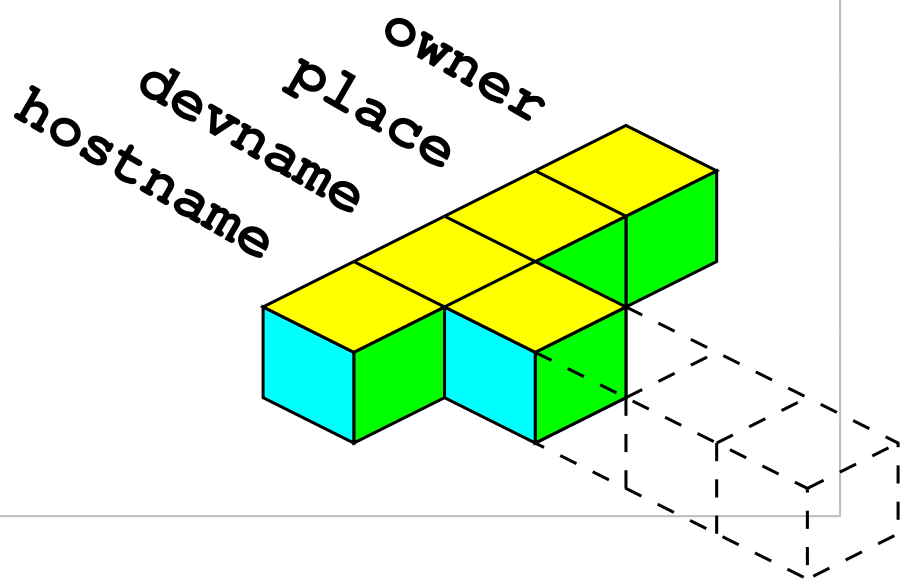


# ホスト情報を蓄える

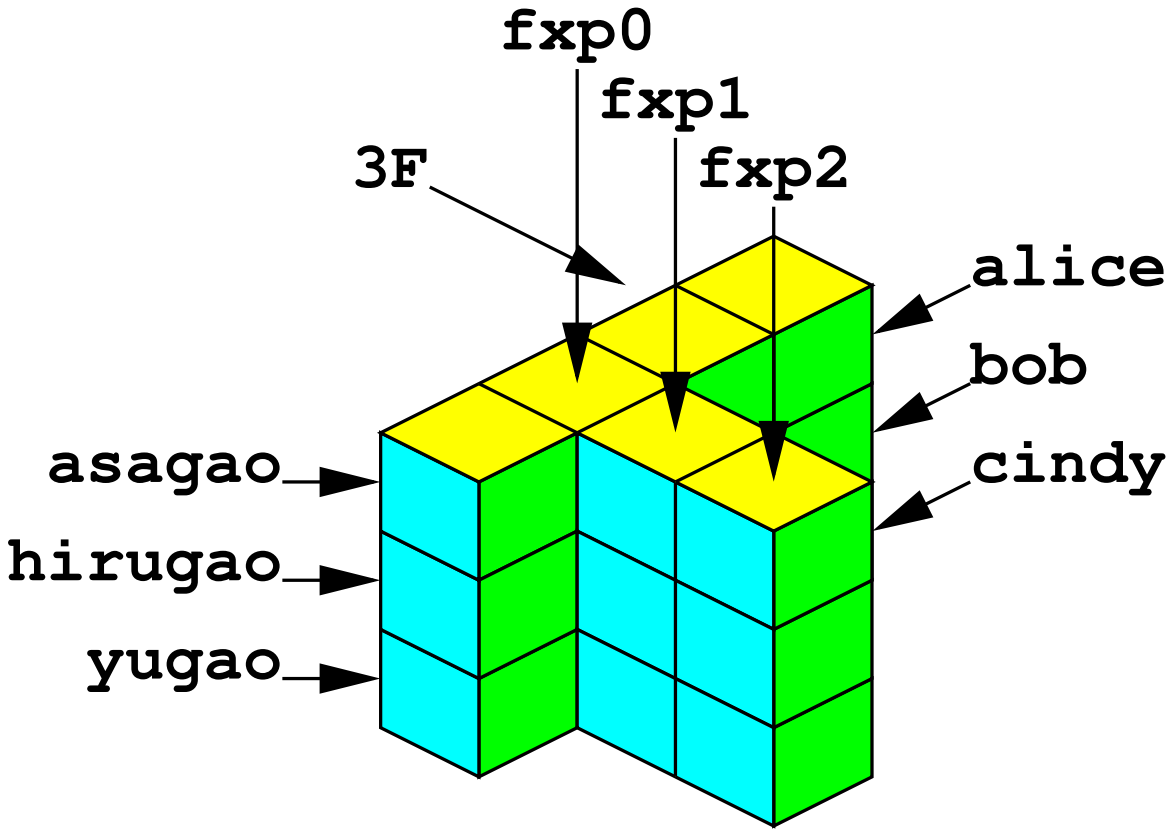
## 複雑なデータの例

- 複数のネットワークデバイスを持つ

```
typedef struct {
    char *hostname;
    int ndev;
    char **devname;
    char *place;
    char *owner;
} host_t;
```



```
% cat hd01
asagao,fxp0:fxp1:fxp2,3F,alice
hirugao,fxp0:fxp1:fxp2,2F,bob
yugao,fxp0:fxp1:fxp2,5F,cindy
```



# 構造体の配列へ格納

```
#define HOSTMAX     (10)
int hostnum=0;
host_t host[HOSTMAX];
char line[BUFSIZ];
int i,j;
...
rec_init(&rec);  rec_init(&rec2);
while(fgets(line, BUFSIZ, stdin)) {
  chomp(line);
  rec_clear(&rec);  rec_clear(&rec2);
  nf = rec_split(&rec, line, ’,’);
  nf2 = rec_split(&rec2, rec.fields[1], ’:’);
```

```
  host[hostnum].hostname = strdup(rec.fields[0]);
  host[hostnum].ndev = nf2;
  host[hostnum].devname =
        (char**)malloc(sizeof(char*)*nf2);
  for(i=0;i<nf2;i++) {
    host[hostnum].devname[i] =
        strdup(rec2.fields[i]);
  }
  host[hostnum].place = strdup(rec.fields[2]);
  host[hostnum].owner = strdup(rec.fields[3]);
  hostnum++;
}
```

ネットワークデバイスの数は可変長

# 構造体内容を表示

```
for(j=0;j<hostnum;j++) {
    printf("%d: %s < ", j, host[j].hostname);
    for(i=0;i<host[j].ndev;i++) {
        printf("%s ", host[j].devname[i]);
    }
    printf("> %s %s\n",
        host[j].place, host[j].owner);
}
```

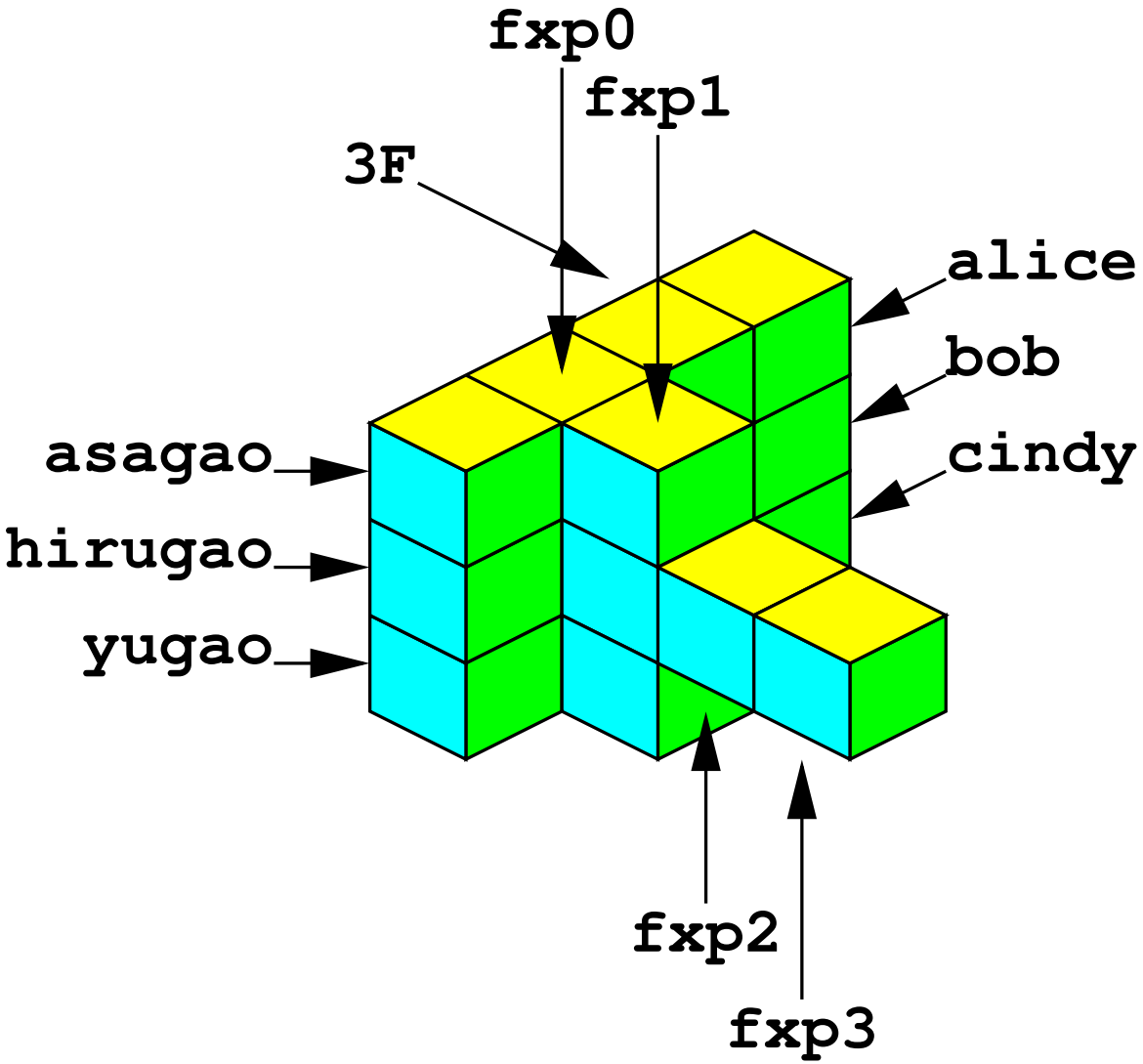
```
% ./a.out < hd01
asagao < fxp0 fxp1 fxp2 > 3F alice
hirugao < fxp0 fxp1 fxp2 > 2F bob
yugao < fxp0 fxp1 fxp2 > 5F cindy
```

# ホスト情報を蓄える *(cont.)*

## ネットワークデバイス数を変えた例

```
% cat hd02
asagao,fxp0:fxp1,3F,alice
hirugao,fxp0:fxp1:fxp2:fxp3,2F,bob
yugao,em0:fxp1,5F,cindy

% ./a.out < hd02
asagao < fxp0 fxp1 > 3F alice
hirugao < fxp0 fxp1 fxp2 fxp3 > 2F bob
yugao < em0 fxp1 > 5F cindy
```


fxp0
fxp1
3F
alice
bob
cindy
asagao
hirugao
yugao
fxp2
fxp3

# ホスト情報を蓄える *(cont.)*

## さらに、デバイスに IP アドレスをつける

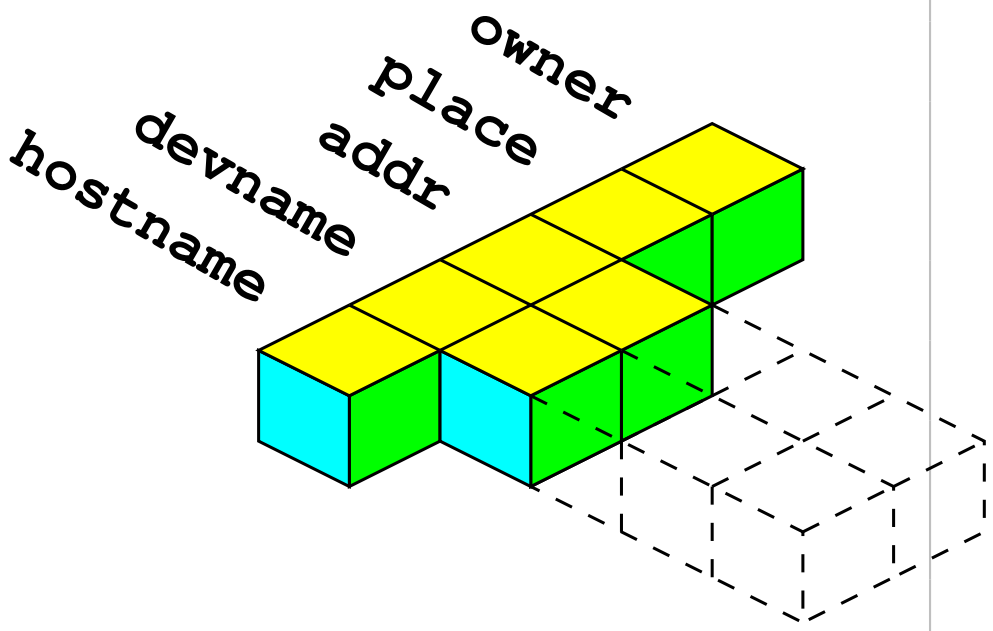
```
% cat hd03a
asagao,fxp0|10.0.0.1:fxp1|10.0.1.1,3F,alice
hirugao,fxp0|10.0.0.2,2F,bob
yugao,em0|192.168.0.1:fxp1|10.0.1.3,5F,cindy

% ./a.out < hd03a
asagao < fxp0|10.0.0.1 fxp1|10.0.1.1 > 3F alice
hirugao < fxp0|10.0.0.2 > 2F bob
yugao < em0|192.168.0.1 fxp1|10.0.1.3 > 5F cindy
```

# ホスト情報を蓄える *(cont.)*

## IPアドレスを別に格納するなら

```
typedef struct {
    char *hostname;
    int ndev;
    char **devname;
    char **addr;
    char *place;
    char *owner;
} host_t;
...
rect_t rec3;  int nf3;
```



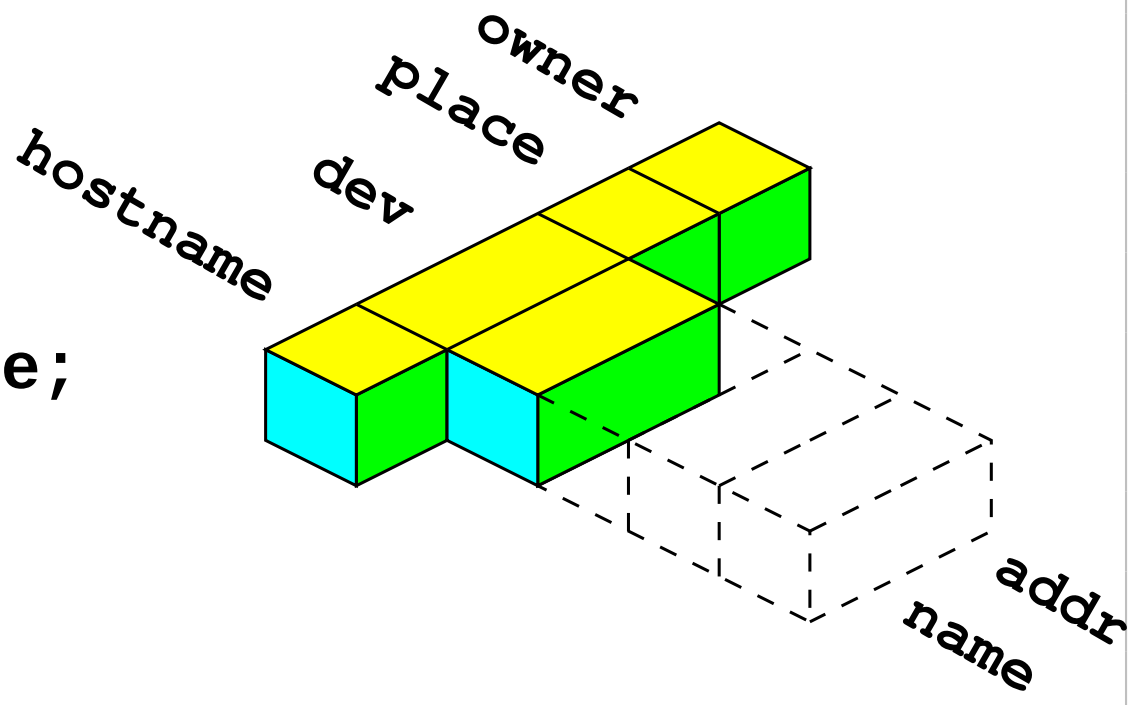
```
rec_clear(&rec3);
...
host[hostnum].ndev = nf2;
host[hostnum].devname =
    (char**)malloc(sizeof(char*)*nf2);
host[hostnum].addr =
    (char**)malloc(sizeof(char*)*nf2);
for(i=0;i<nf2;i++) {
  nf3 = rec_split(&rec3, rec2.fields[i], '|');
  host[hostnum].devname[i] =
        strdup(rec3.fields[0]);
  host[hostnum].addr[i] =
        strdup(rec3.fields[1]);
}
```

# 実行結果

```
% ./a.out < hd03a
asagao
        fxp0 10.0.0.1
        fxp1 10.0.1.1
  3F alice
hirugao
        fxp0 10.0.0.2
  2F bob
yugao
        em0 192.168.0.1
        fxp1 10.0.1.3
  5F cindy
```

# デバイス専用構造体を用いる

```
typedef struct {
    char *name;
    char *addr;
} dev_t;
typedef struct {
    char   *hostname;
    int    ndev;
    dev_t *dev;
    char  *place;
    char  *owner;
} host_t;
```



```
host[hostnum].ndev = nf2;
host[hostnum].dev =
        (dev_t*)malloc(sizeof(dev_t*)*nf2);
for(i=0;i<nf2;i++) {
    nf3 = rec_split(&rec3, rec2.fields[i], '|');
    host[hostnum].dev[i].name =
        strdup(rec3.fields[0]);
    host[hostnum].dev[i].addr =
        strdup(rec3.fields[1]);
}
```

結果は同じ

# 補足

libspji.c は格納に関してはこれまでの辞書と同じ

- データの配列、確保長、使用長がメンバ
- 多くの場合データは文字ポインタ（文字列）
- 領域が足りなくなると拡張

違い

- (libspji.c) split や join そして print
- (これまで) 検索や重複省略

どういう処理が一般的か分かって来たことだろう

# 演習

1) libspji.c を実際に作成せよ★
2) レコード構造体の拡張
   a) split 向け区切り文字
      - 区切り文字を保存するメンバを設けよ★
      - メンバに保存されている区切り文字を用いる split 関数を作れ★
      - 現行 split 関数の名前を変更せよ★
   b) print 向け区切り文字
      - 区切り文字を保存するメンバを設けよ★

# 演習 *(cont.)*

  - ◇ split 向けとは別のメンバとする
- メンバに保存されている区切り文字を用いる print の関数を作れ★
- 区切り文字を指定する pint 関数を作れ★

整理すると、最終的に関数が4つあることになる

```
int rec_split(rec_t*, char*);
int rec_split_sep(rec_t*, char*, int);
int rec_print(rec_t*);
int rec_print_sep(rec_t*, int);
```

# 演習 *(cont.)*

3) 3次元データのサンプル

a) 区切り文字を使って記録したテキストファイルを作る★

b) 内容を読み込み表示するプログラムを作れ★

- プログラムは以下ように実行できるようにすること

```
% ./yourprogram < yourdata
```

# 演習 *(cont.)*

4) split を使ってファイルを解析し、出納処理するプログラムを作れ★

- データファイルの内容は以下のとおり

```
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
```

# 演習 *(cont.)*

- 漢字コードは EUC とする
- 各人物の初期値（はじめの残高）は 0 とする

5) 前述の名前とIPアドレスをもつデバイスを複数持つホストの情報を蓄えるプログラムを実際に作れ ★

- デバイス名とIPアドレスは別のメンバにしても、まとめて新たな型を設けても構わない