9 配列、ポインタ、文字列

9–1 復習 一次元配列

配列宣言の例：

宣言	···	説明
`int a[100];`	···	大きさが100の`int`型一次元配列。 `a[0]`, `a[1]`, ..., `a[99]` が確保される。
`int f[5] = {0, 1, 2, 3, 4};`	···	`f[0]=0`, `f[1]=1`, ... と初期設定される。
`int a[100] = {0};`	···	初期値指定が足りない分は `0` と見なされ、結局全て `0` に初期設定される。
`double a[] = {2.0, 3e-2, -5.};`	···	初期値指定が完全な場合は、配列の大きさ指定は省略できる。

宣言 ··· 説明
`char s[] = "abc";` ··· char型配列で文字列を表す際の略記法。 char s[4]={'a', 'b', 'c', '\0'}; と宣言するのと同等。

注意：

- 配列の添字は 0 から始まる。
- 配列の添字が有効範囲にあることをチェックするのはプログラマ。
- 添字式を囲む四角括弧 [ ] は実は演算子。
  結合の優先順位は、関数引数を囲む（ ）と共に最も高い。
- 配列の大きさは定数式で指定するのが無難。
  ⇨ 右下は誤り、左下は ANSI C99 より前の規格で誤り。

```
int f(int size)
{
  int a[size];
  ......
```

```
  ......
scanf("%d", &size);
double x[size];
  ......
```

この制約はコンパイル時に確保する領域の大きさが確定している必要があるためである。

配列の大きさがコンパイル時に決まらない場合は、

（方法1）多少の無駄があっても十分な大きさの配列を用意する。

または、

（方法2）malloc()関数等を用いて各々次の様な書き方をする。

```
int f(int size)
{
  int *a;
  a = (int *) malloc(sizeof(int)*size);
  ......
```

```
int *x;
......
scanf("%d", &size);
x = (int *) malloc(sizeof(int)*size);
......
```

9–2 復習 ポインタ

番地演算子 & と間接演算子 *（7.3節）：

&v ··· 変数 v へのポインタ（≈番地）。

*p ··· ポインタ p の指す記憶領域、
すなわち、p番地の記憶領域。

9–3 配列とポインタの関係, ポインタ算術

配列名

- 一次元配列の場合、配列名は計算機内部では先頭要素を指す定数ポインタとして扱われている。

（. . . . . [ ] は演算子。）

ポインタ算術

ポインタに関する算術は特殊である。例えば、

- ポインタ p に関して ++p という演算を施すと、このポインタはメモリ空間上の次の番地ではなく配列の次の要素を指すようになる。

- ポインタ p と int型変数 i の間の加算 p+i は実際には
  p + i× sizeof(pの指すデータ型)番地
  を表す。

  ⇨ 配列要素 a[i] の代わりに*(a+i) という式を
  用いることが出来る。

  また、逆に、配列要素を指すポインタ p があったとすると、
  p という名前の配列が宣言されてなくても、*(p +i) の意味
  で p[i] と書くことができる。

- p と q が同じ配列を指しているポインタである時、p-q は実際にはp
  の指している所からqの指している所までの要素の個数、すなわち

$$\frac{\text{qの指す番地からpの指す番地までのバイト数(符号あり)}}{\text{sizeof(pの指すデータ型)}}$$

  という整数を表す。

例9．1（ポインタ算術）

```
[motoki@x205a]$ nl pointer-arithmetic-Kelley.c
 1  #include <stdio.h>

 2  int main(void)
 3  {
 4    double  a[2], *p, *q;

 5    p = a;
 6    q = p+1;                              /*ポインタ算術*/
 7    printf("%d\n", q-p);                  /*ポインタ算術*/
 8    printf("%d\n", (int)q - (int)p);      /*通常の算術*/
 9    return 0;
10  }
[motoki@x205a]$ gcc pointer-arithmetic-Kelley.c
[motoki@x205a]$ ./a.out
1
8
```

```
[motoki@x205a]$
```

配列とポインタの関係について：

- 配列名は、その配列の先頭要素を指す定数ポインタとして振舞う。
  従って、

  ① `s[i]` と`*(s+i)` は等価である。
  （`s`が`char`型以外の時も、これは成立する。）

  ② 配列要素を指すポインタ `p` があったとすると、`*(p+i)` の意味で `p[i]`
  と書くことも出来る。

例９．２（配列とポインタの関係）　先の例7.4 の中ので挙げた関数 sum の定義

```
14  double sum(double a[], int size)
15  {
16    int    i;
17    double sum=0.0;

18    for (i=0; i < size; ++i)
19      sum += a[i];
20    return sum;
21  }
```

は、次のように書くことも出来る。

```
14  double sum(double a[], int size)
15  {
16    int    i;
17    double sum=0.0;

18    for (i=0; i < size; ++i)
19      sum += *(a+i);
20    return sum;
21  }
```

これは、さらに次のように書くことも出来る。

```
14  double sum(double a[], int size)
15  {
16    int    i, *p;
17    double sum=0.0;

18    for (p=a; p < &a[size]; ++p)
19      sum += *p;
20    return sum;
21  }
```

例題９．３（Quicksort;未整列区間を配列要素へのポインタで表して引数結合を行う
例題 **7.3** で挙げた **Quicksort** のプログラムにおいては、未整列の区間の両端の位置 (関数の引数 from と to) を表すのに配列の添字を用いていた。配列の添字の代わりに配列要素を指すポインタを関数引数として用いるように、例題 **7.3** のプログラムを書き換えてみよ。

（考え方）　配列名をquicksort()関数に引き渡す必要は無くなる。従って、main()内からのquicksort()関数の呼び出しは

```
quicksort(a, &a[0], &a[SIZE-1]);
```

ではなく

```
quicksort(&a[0], &a[SIZE-1]);
```

これに対応して、quicksort()の関数プロトタイプは...

```
void quicksort(int *from, int *to);
```

quicksort()の中からpartition()を呼び出す際に引数として引き渡すデータも配列添字ではなく配列要素へのポインタとし、partition()の関数値もポインタとするのが妥当である。従って、
partition()の関数プロトタイプは...

```
int *partition(int *from, int *to);
```

（プログラミング）　修正例を次に示す。

```
[motoki@x205a]$  nl pointer-quicksort-3.c
     1 #include <stdio.h>
     2 #include <stdlib.h>   /* 乱数発生のライブラリ関数を使う..

     3 #define  SIZE   100
     4 #define  WIDTH  10
     5 #define  TRUE    1

     6 void set_an_array_random(int a[], int size);
     7 void pretty_print(int a[], int size);
     8 void quicksort(int *from, int *to);
     9 int  *partition(int *from, int *to);

    10 int main(void)
    11 {
    12   int      a[SIZE], seed;
```

```
13    printf("Input a random seed (0 - %d):  ", RAND_MAX);
14    scanf("%d", &seed);
15    srand(seed);

16    set_an_array_random(a, SIZE);
17    printf("\nbefore sorting:\n");
18    pretty_print(a, SIZE);

19    quicksort(&a[0], &a[SIZE-1]);
20    printf("\nafter sorting:\n");
21    pretty_print(a, SIZE);
22    return 0;
23 }

24 /*************************************************
25 /* 引数で与えられた配列の各要素をランダムに設定... */
```

```
26 /*-------------------------------------
27 /* (仮引数) x     : int型配列
28 /*             size : int型配列 x の大きさ
29 /* (関数値)  :  なし
30 /* (機能) :  配列要素 x[0]～x[size-1] に 0～999 の間の乱
31 /*              設定する。
32 /*************************************************
33 void set_an_array_random(int x[], int size)
34 {
35   int i;

36   for (i=0; i<size; ++i)
37     x[i] = rand() % 1000;
38 }

39 /*************************************************
40 /* 引数で与えられた配列の要素を順番に全て出力... */
```

```
41 /*------------------------------------
42 /* (仮引数) x     : int型配列
43 /*            size : int型配列 x の大きさ
44 /* (関数値)  :  なし
45 /* (機能) :  配列要素 x[0]～x[size-1] の値を順番に全て出.
46 /*            する。但し、各々の値は横幅7カラムのフィール...
47 /*            に出力することにし、また、1行にWIDTH個の要...
48 /*            を出力する。
49 /************************************************
50 void pretty_print(int x[], int size)
51 {
52   int i, count=1;

53   for (i=0; i<size; ++i, ++count) {
54     printf("%7d", x[i]);
55     if (count >= WIDTH) {
56       printf("\n");
```

```
57        count = 0;
58      }
59    }
60    if (count > 1)
61      printf("\n");
62  }

63  /*************************************************
64  /* 引数で与えられた配列要素を小さい順に並べ替え... */
65  /*-----------------------------------
66  /* (仮引数) from : int型配列要素を指すポインタ
67  /*           to   : int型配列要素を指すポインタ
68  /* (関数値)  :  なし
69  /* (機能) :  quicksortアルゴリズムを使って、配列要素
70  /*                *from,*(from+1),*(from+2), ..., *to
71  /*            を値の小さい順に並べ替える。
72  /*************************************************
```

```
73 void quicksort(int *from, int *to)
74 {
75   int *pivot_pointer;

76   if (from < to) {
77     pivot_pointer = partition(from, to);     /* 分割操作
78     quicksort(from, pivot_pointer - 1);
79     quicksort(pivot_pointer + 1, to);
80   }
81 }

82 /*************************************************
83 /* 引数で与えられた配列の部分列にquicksortの分割操作... */
84 /*                                  (quicksortの関数) */
85 /*-----------------------------------
86 /* (仮引数) from : int型配列要素を指すポインタ
87 /*          to   : int型配列要素を指すポインタ (> from)
```

```
 88 /* (関数値) : 分割操作によって得られた枢軸要素を指すポイ...
 89 /*              (以下の「(機能)」の項で出て来る pivot_pointe
 90 /* (機能) : *from,*(from+1), ..., *to を並べ替えて
 91 /*            max*from,...,*(pivot_point-1) <= *pivot_poi
 92 /*            *pivot_point < min*(pivot_point+1),...,*to
 93 /*            となるようにする。
 94 /*************************************************************
 95 int *partition(int *from, int *to)
 96 {
 97   int pivot;

 98   pivot = *from;          /* 最初の要素を枢軸要素に選ぶ。 */
 99   while (TRUE) {          /* 工夫の余地あり。              */
100     for ( ; from<to && *to>pivot; --to)
101       ;
102     if (from == to) {
103       *from = pivot;
```

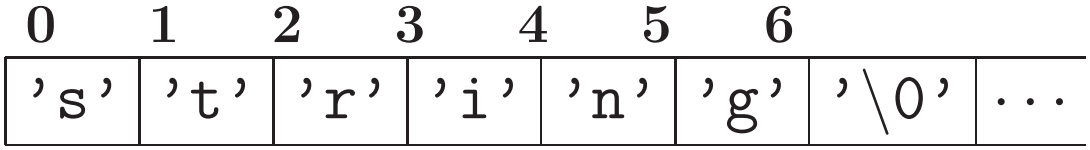
```
104       return from;
105     }
106     *from = *to;

107     for ( ; from<to && *from<=pivot; ++from)
108       ;
109     if (from == to) {
110       *to = pivot;
111       return to;
112     }
113     *to = *from;
114   }
115 }
```

注意：
この例は、配列要素へのポインタを引数結合することによって配列の受渡しをすることも出来ることを示しているだけで、そうすべきだと言っている訳ではない。　実際、そうすることによって、プログラムが分かりにくくなることが多い。

9–4 文字列の扱い, 不揃い配列

文字列についてのまとめ(1.4.7節):

- char型の配列を使う。
- 文字列の終わりの印として文字列の最後にヌル文字'\0'を置く。
  ⇨ (配列の大きさ) ≥ (文字列の長さ) + 1



- 文字列を2重引用符で囲めば文字列定数
- char型配列で文字列を表す場合は、初期設定を次の様に行うことが出来る。

```
char s[]="string";
```

(これは次の設定と同等。)

```
char s[]={'s','t','r','i','n','g','\0'};
```

補足事項：

- 文字列定数の値はその文字列が確保されている領域へのポインタになっている。

  ⇨ `char *p="abc";` という宣言も出来る。
  `"abc"[1]` や `*("abc"+2)` は文法的に正しい式になる。

- 2つの宣言の違い:

  `char *p="abc";` ··· pという名前のポインタのために記憶領域が確保される。
  ⇨ 計算している内にpが"abc"という文字列領域を指さなくなることもある。

  `char a[]="abc";` ··· aは定数ポインタ。

- ナル文字 '\0' を出力しないよう気を付けること。
  [印字可能文字ではなく機能文字であるため。]

- 文字列操作のライブラリ関数が多数用意されている。
  ⇨ この講義ノート4.7節等を御覧下さい。

例題９．４（文字列操作のライブラリ関数）　長さが **10** 以下の英単語 W と１行が **80** 文字以下の英文章を読み込み、英文章中に現れる単語 W を全て大文字に変換して得られる文章を出力するＣプログラムを作成せよ。

（考え方）

最初に英単語 W を読み込むことにすれば、あとは

① 英文章の次の１行の読み込み，
② 読み込んだ１行の中に現れる単語Wを全て大文字に変換，
③ 変換後の１行の出力

という作業を繰り返すだけである。

（考え方）

最初に英単語 W を読み込むことにすれば、あとは

① 英文章の次の1行の読み込み，
② 読み込んだ1行の中に現れる単語Wを全て大文字に変換，
③ 変換後の1行の出力

という作業を繰り返すだけである。

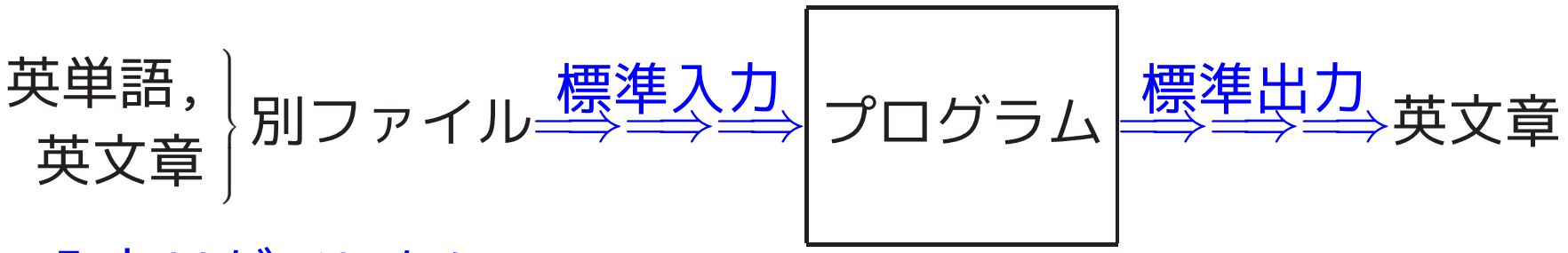
ここで、

- 英文章の次の1行を読み込むためには 、
  fgets() というライブラリ関数を利用できる。
  (⟹ 4.7節を参照)
- 文字列の中から小さな文字列パターンを探索するためには 、
  strstr() という文字列操作のライブラリ関数を利用できる。
- 英単語 W の長さを測るためには 、
  strlen() という文字列操作のライブラリ関数を利用できる。
- 英字を大文字に変換するためには 、
  toupper() という文字種類変換のライブラリ関数を利用できる。

(プログラミング)

英単語 w を保持するために　　char型配列 word[] を、
英単語 w を大文字化したものを保持するために
　　char型配列 WORD[] を、
1行分の文字列を保持するために char型配列 line[] を、
読み込んだ1行分の文字列を前から順に走査するために
　　ポインタ変数 remaining_seq を
用意した。

そして、次の様なデータ入力を想定して、プログラムを構成した。



(入力リダイレクション)

```
[motoki@x205a]$  nl toupper-some-words-in-sentences.c   Enter
     1 /* 長さが10以下の英単語 w と1行が80文字以下の英文章を */
     2 /* 読み込み、英文章中に現れる単語wを全て大文字に変換  */
     3 /* して得られる文章を出力するCプログラム             */
     4 /*                 (入力リダイレクションにより       */
     5 /*                  ファイルから入力することを想定する。) */

     6 #include <stdio.h>
     7 #include <string.h>
     8 #include <ctype.h>

     9 int main(void)
    10 {
    11   char word[11], WORD[11], line[82], *remaining_seq;
    12   int  word_length, i;       /* 英文章の1行は最長で     */
    13                              /* 80文字+改行コード+'\0' */

    14   scanf("%10s", word);     /* 大文字にする英単語を入力 */
```

```
15    word_length=strlen(word);
16    for (i=0; i<word_length; i++)
17      WORD[i]=toupper(word[i]);
18    WORD[word_length]='\0';

19    printf("単語 %s を大文字に換えて得られる文章:\n", word);
20    while (fgets(line, 82, stdin)!=NULL) {  /*次の1行を..
21      remaining_seq = line;
22      while ((remaining_seq=strstr(remaining_seq, word))
                                                !=NULL) {
23        for (i=0; i<word_length; i++)
24          remaining_seq[i]=WORD[i];
25        remaining_seq += word_length;
26      }
27      printf("   %s", line);
28    }
29    return 0;
30  }
```

```
[motoki@x205a]$
```

```
[motoki@x205a]$  gcc toupper-some-words-in-sentences.c  Enter
[motoki@x205a]$  cat toupper-some-words-in-sentences.data  Enter
language
------
Why C?
------
C is a small language.  And small is beautiful in programming.
C has fewer keywords than Pascal, where they are known as
reserved words, yet it is arguably the more powerful language.
C gets its power by carefully including the right control
structures and data types and allowing their uses to be nearly
unrestricted where meaningfully used.  The language is readily
learned as a consequence of its functional minimality.  C is
the native language of UNIX, and UNIX is a major interactive
operating system on workstation, servers, and mainframes.
Also, C is the standard development language for personal
computers.  Much of MS-DOS and OS/2 is written in C.  Many
windowing packages, database programs, graphics libraries, and
other large-application packages are written in C.
(A.Kelly&I.Pohl, "A Book on C" 4th ed., Addison-Wesley, 1998.)
[motoki@x205a]$  ./a.out <toupper-some-words-in-sentences.data  Enter
```

単語 language を大文字に換えて得られる文章：

```
   ------
   Why C?
   ------
   C is a small LANGUAGE.  And small is beautiful in programming.
   C has fewer keywords than Pascal, where they are known as
   reserved words, yet it is arguably the more powerful LANGUAGE.
   C gets its power by carefully including the right control
   structures and data types and allowing their uses to be nearly
   unrestricted where meaningfully used.  The LANGUAGE is readily
   learned as a consequence of its functional minimality.  C is
   the native LANGUAGE of UNIX, and UNIX is a major interactive
   operating system on workstation, servers, and mainframes.
   Also, C is the standard development LANGUAGE for personal
   computers.  Much of MS-DOS and OS/2 is written in C.  Many
   windowing packages, database programs, graphics libraries, and
   other large-application packages are written in C.
   (A.Kelly&I.Pohl, "A Book on C" 4th ed., Addison-Wesley, 1998.)
[motoki@x205a]$
---
```

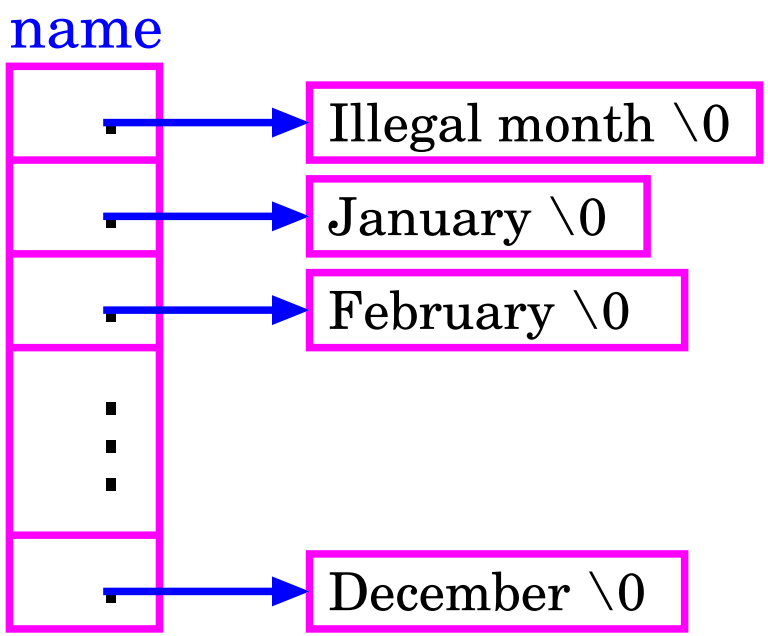
不揃い配列：

- 長さの様々な文字列を2次元のchar型配列に入れておくのはメモリに無駄が出来る。例えば、例8.2で作ったプログラムの中(main)に

```
                        /* name[k]              */
char name[][15]={   /* =k番目の月の名前 */
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
      };
```

という宣言があるが、この部分は次のように書き直すとメモリが節約できる。(不揃い配列という。)

```
                        /* name[k]                */
char *name[]={          /* =k番目の月の名前 */
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
      };
```



9–5 多次元配列

多次元配列の宣言（3.8節）：

- 配列名が a、大きさが $k_1 \times k_2 \times \cdots \times k_n$ の配列の宣言／領域確保は次の様に行う。

  データ型　a[k_1][k_2] ··· [k_n]

- 宣言時の配列の初期化は例えば次の様に行う。

```
int num[12]={31, 28, 31, 30, 31, 30,
             31, 31, 30, 31, 30, 31};
char tab[2][3]={{1,2,3}, {4,5,6}};
```

1番目の添字	0			1		
2番目の添字	0	1	2	0	1	2
tab	1	2	3	4	5	6
	tab[0]			tab[1]		

多次元配列の表し方：　C言語においては、

1次元配列を1つの要素とする配列を2次元配列、

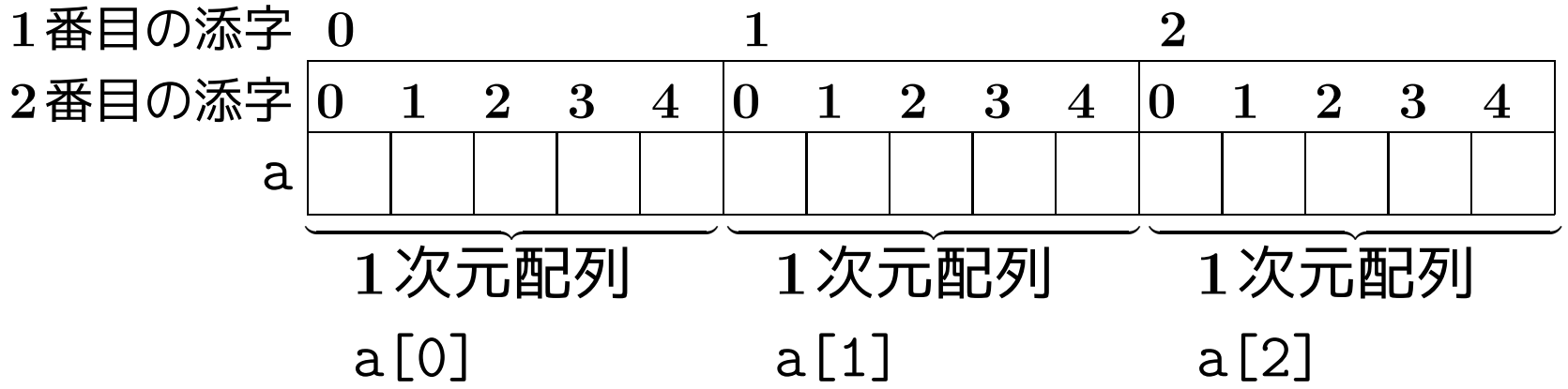
2次元配列を1つの要素とする配列を3次元配列、.....

と考える。

⇨ 全ての要素はメモリ上に連続的に配置される。

例9．5（2次元配列の様子）　`int a[3][5];`　という宣言は、演算子[ ]の結合性(左から右)から　`int (a[3])[5];`　と同等である。

この宣言は、配列aはa[0]～a[2]から成り、配列要素a[0]～a[2]は各々int型の1次元配列であると言っている訳だから、次のようなメモリが確保される。



配列名の値

- 1次元配列の場合、配列名は計算機内部では先頭要素を指す定数ポインタとして扱われる。
- C言語においては、配列の機構としては1次元のものしか用意されていない。[多次元配列は1次元配列の組合せにすぎない。]

⇨ 配列Aの名前は計算機内部では
A[0]への定数ポインタ &A[0] と同等に扱われる。

例9．6（配列名の値）　宣言　`int a[3][5];`　によって確保される2次元配列の場合、

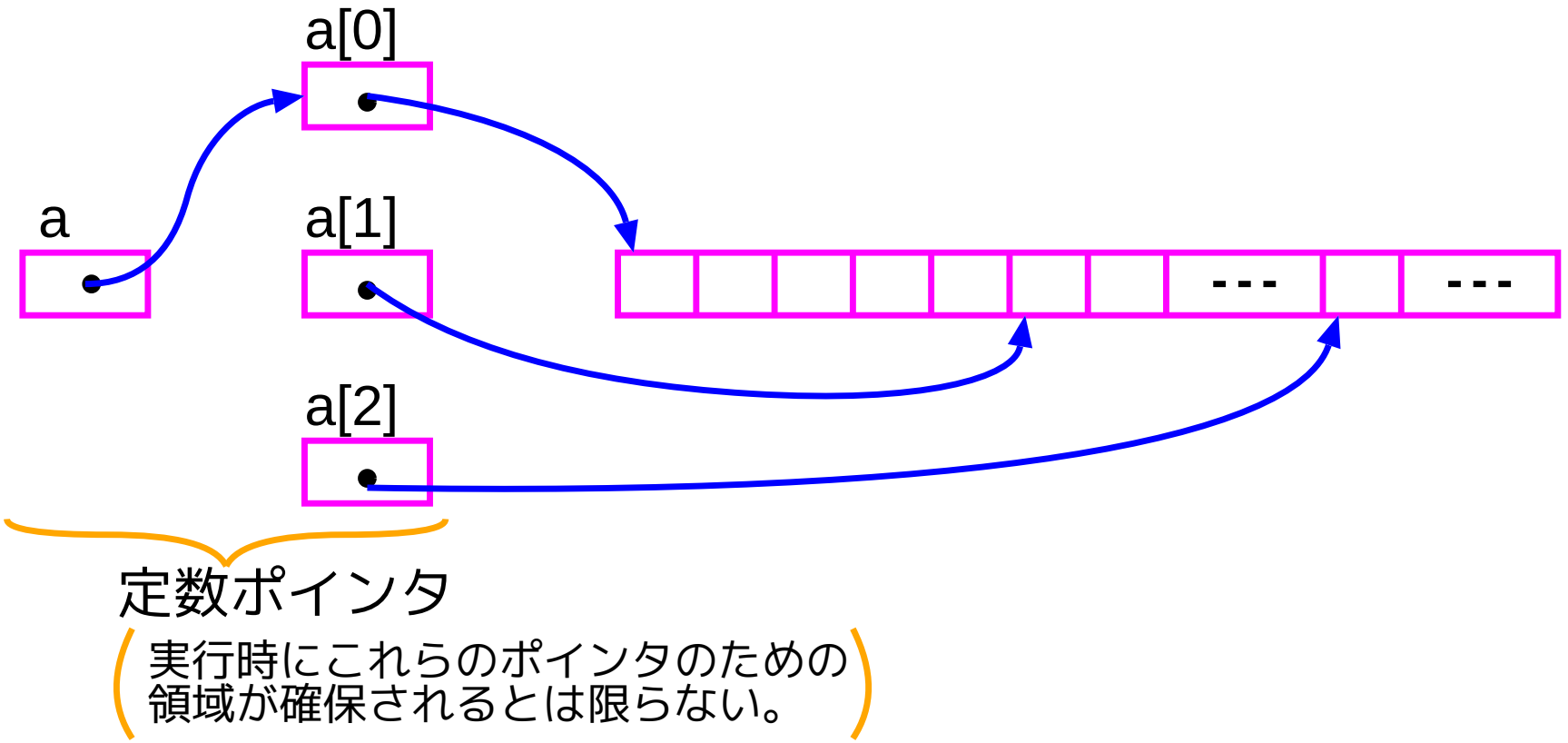
配列名`a`は　　`a[0]`への定数ポインタ `&a[0]` と同等、

配列名`a[0]`は `a[0][0]`への定数ポインタ `&a[0][0]` と同等、

配列名`a[1]`は `a[1][0]`への定数ポインタ `&a[1][0]` と同等、

配列名`a[2]`は `a[2][0]`への定数ポインタ `&a[2][0]` と同等、

ということになる。



配列要素へのアクセスの仕方：

多次元配列の場合、配列要素へのアクセスの仕方は沢山ある。

例9．7（配列要素へのアクセスの仕方）　宣言

```
int a[3][5];
```

によって確保される2次元配列の場合、次の式は全て同等。

```
a[i][j]
(*(a+i))[j]
*(a[i]+j)
*(*(a+i)+j)
```

} ··· (1 次元配列の場合に、一般に A[i] と *(A+i) が同等であることを用いているだけ。)

```
*(&a[0][0]+5*i+j)
```

··· (配列の内部構造を考えている。)

初期設定の仕方：

1次元配列の場合の考え方を自然に拡張しただけ。

例9．8（多次元配列の初期設定）　1次元配列の場合は、例えば、

```
int a[3]={ a[0]の初期値 , ... ,  a[2]の初期値 );
```

という風に初期設定する。 いま、a[0]～a[2] をint型データ領域ではなく大きさ5のint型配列と見ることにして、同じ書き方に乗っ取って書けば、

```
int a[3][5]={ a[0]の初期値, ..., a[2]の初期値);
```

すなわち、

```
int a[3][5]={{a[0][0]の初期値, ..., a[0][4]の初期値},
             {a[1][0]の初期値, ..., a[1][4]の初期値},
             {a[2][0]の初期値, ..., a[2][4]の初期値}
};
```

ということになる。 指定が欠けている場合に0が補充されるという規則も、1次元の場合と同様に生きている。
