Two Dimensional Arrays in C

Pei-yih Ting

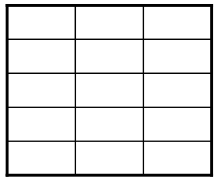
Version 1. Fixed dimensions **5** by **3**

- Both dimensions are fixed
- Allocated either in data segment or in stack
- Example

```
int i, j;
int x[5][3];

for (i=0; i<5; i++)
   for (j=0; j<3; j++)
      x[i][j] = 0;

fun(x)
```

Conceptual layout

x



Physical layout

```
void fun(int x[5][3]) {….}
void fun(int x[][3]) {….}
void fun(int (*x)[3]) {….}
void fun(int (* const x)[3]) {….}
```

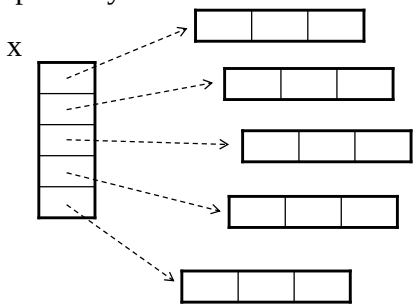

Version 2a. Dynamic allocated **5** by **n**

- Size of the first dimension is fixed as 5, size of the second dimension is variable
- Allocated on the stack (x[]) and the heap (x[i][])
- Example

```
int i, j, n=3;
int *x[5];

for (i=0; i<5; i++)
   x[i] = (int *) malloc(sizeof(int)*n);

for (i=0; i<5; i++)
   for (j=0; j<n; j++)
      x[i][j] = 0;

fun(x);

for (i=0; i<5; i++)
   free(x[i]);
```

Conceptual layout

x



```
void fun(int *intarray[]) { … } or void fun(int ** const intarray) { … }
                              or void fun(int ** intarray) { … }
```

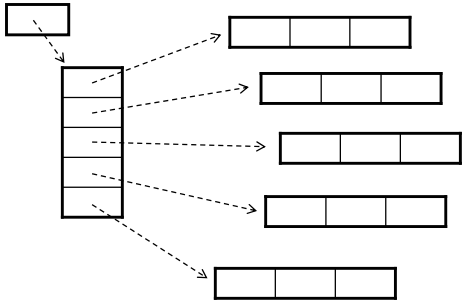

Version 2b. Dynamic allocated **m** by **n**

- Size of both dimensions are variable
- Both allocated on the heap
- Example

```
int i, j, m=5, n=3;
int **x;

x = (int **) malloc(sizeof(int *)*m);
for (i=0; i<m; i++)
   x[i] = (int *) malloc(sizeof(int)*n);

for (i=0; i<m; i++)
   for (j=0; j<n; j++)
      x[i][j] = 0;

fun(x)

for (i=0; i<m; i++)
   free(x[i]);
free(x);
```

Conceptual layout

x



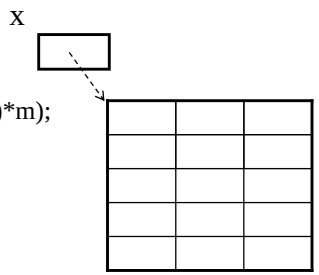
```
void fun(int **intarray) { … } or  void fun(int *intarray[]) { … }
```

Version 3. Dynamic allocated **m** by **3**

- Size of the first dimension is variable, size of the second dimension is fixed as 3
- Allocated on the heap
- Example

```
int i, j, m=5;
int (*x)[3];

x = (int (*)[3]) malloc(sizeof(int [3])*m);

for (i=0; i<m; i++)
   for (j=0; j<3; j++)
      x[i][j] = 0;

fun(x);

free(x);
```

Conceptual layout



Physical layout

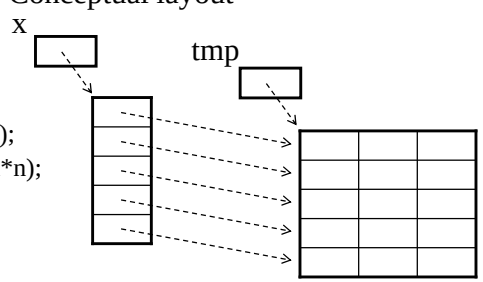
```
void fun(int (*intarray)[3]) { … }
void fun(int (*const intarray)[3]) { … }
void fun(int intarray[][3]) { … }
```


Version 4. Dynamic allocated **m** by **n**

- Sizes of both dimensions are variable
- Allocated on the heap
- Example

```
int i, j, m=5, n=3;
int **x, *tmp;

x = (int **) malloc(sizeof(int*)*m);
tmp = (int *) malloc(sizeof(int)*m*n);
for (i=0; i<m; i++)
   x[i] = &tmp[i*n];

for (i=0; i<m; i++)
   for (j=0; j<n; j++)
      x[i][j] = 0;

fun(x);
free(x[0]);
free(x);
```

Conceptual layout

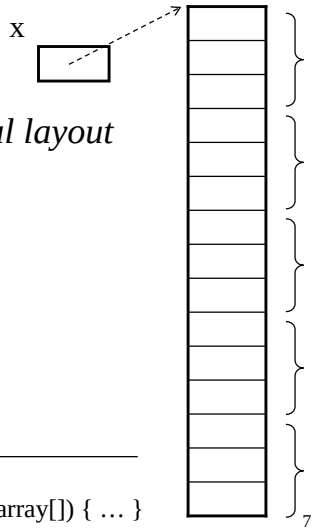



```
void fun(int **intarray) { … }
void fun(int ** const intarray) { … }
void fun(int *intarray[]) { … }
```


Version 5. Dynamic allocated **m** by **n**

- Sizes of both dimensions are variable, emulate with 1-D array syntax
- Allocated on the heap
- Example

```
int i, j, m=5, n=3;
int *x;
x = (int *) malloc(sizeof(int)*m*n);
for (i=0; i<m; i++)
   for (j=0; j<n; j++)
      x[i*n+j] = 0; // x[i][j] does not work
                    // (&x[i*n])[j] is OK
fun(x);
free(x);
```




```
void fun(int * const intarray) { … } or
void fun(int * intarray) { … } or void fun(int intarray[]) { … }
```


Array with Negative Index

- One dimensional

static

```
int i, array0[21], *array;

for (i=0; i<21; i++)
   array0[i] = i-10;
array = &array0[10];
for (i=-10; i<=10; i++)
   printf("%d ", array[i]);
```

dynamic

```
int i, *array0, *array;

array0 = (int *) malloc(21*sizeof(int));
for (i=0; i<21; i++)
   array0[i] = i-10;
array = array0 + 10;
for (i=-10; i<=10; i++)
   printf("%d ", array[i]);
free(array0);
```

ensures that *(array+i) and *(array0+10+i) are the same

-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10

Array with Negative Index (cont'd)

- Two dimensional

static

```
int i,j, mat0[11][11], (*mat)[11];
for (i=0; i<11; i++)
   for (j=0; j<11; j++)
      mat0[i][j] = (i-5)*10 + (j-5);
mat = (int (*)[11])((int *)(mat0+5)+5);
for (i=-5; i<=5; i++)
{
   for (j=-5; j<=5; j++)
      printf("%3d ", mat[i][j]);
   printf("\n");
}
printf("\n");
```

dynamic, version 1

```
int i, j, (*mat0)[11], (*mat)[11];
mat0 = (int (*)[11]) malloc(11*sizeof(int[11]));
for (i=0; i<11; i++)
   for (j=0; j<11; j++)
      mat0[i][j] = (i-5)*10 + (j-5);
mat = (int (*)[11])(&mat0[5][5]);
for (i=-5; i<=5; i++)
{
   for (j=-5; j<=5; j++)
      printf("%3d ", mat[i][j]);
   printf("\n");
}
free(mat0);
```

ensures that *(*(mat+i)+j) and *(*(mat0+5+i)+5+j) are the same

Array with Negative Index (cont'd)

- Two dimensional, dynamic, version 2

```
int i, j, **matrix0, **matrix1, **matrix;
matrix0 = (int **) malloc(11*sizeof(int *));
matrix1 = (int **) malloc(11*sizeof(int *));
for (i=0; i<11; i++)
{
   matrix0[i] = (int *) malloc(11*sizeof(int));
   matrix1[i] = matrix0[i] + 5;
}
matrix = matrix1 + 5;
for (i=0; i<11; i++)
   for (j=0; j<11; j++)
      matrix0[i][j] = (i-5)*10 + (j-5);
```

```
for (i=-5; i<=5; i++)
{
   for (j=-5; j<=5; j++)
      printf("%3d ", matrix[i][j]);
   printf("\n");
}
for (i=0; i<11; i++) free(matrix0[i]);
free(matrix0);
free(matrix1);
```


Array with Arbitrary Index

- Two dimensional, static

```
int i,j, mat0[3][2]={1,2,3,4,5,6};
int (*mat)[2]=(int (*)[2])((int *)(&mat0[1][4]);
for (i=-1; i<=1; i++)
{
   for (j=-4; j<=-3; j++)
      printf("%3d ", mat[i][j]);
   printf("\n");
}
printf("\n");
```

mat	7	8
4	1	2
5	3	4
6	5	6

mat	-4	-3
-1	1	2
0	3	4
1	5	6

```
int i,j, mat0[3][2]={1,2,3,4,5,6};
int (*mat)[2]=(int (*)[2])((int *)(&mat0[-4][-7]);
for (i=4; i<=6; i++)
{
   for (j=7; j<=8; j++)
      printf("%3d ", mat[i][j]);
   printf("\n");
}
printf("\n");
```