

# Pointers to pointers & multi-dimensional arrays

# Pointers to pointers

```
int i=3;
```

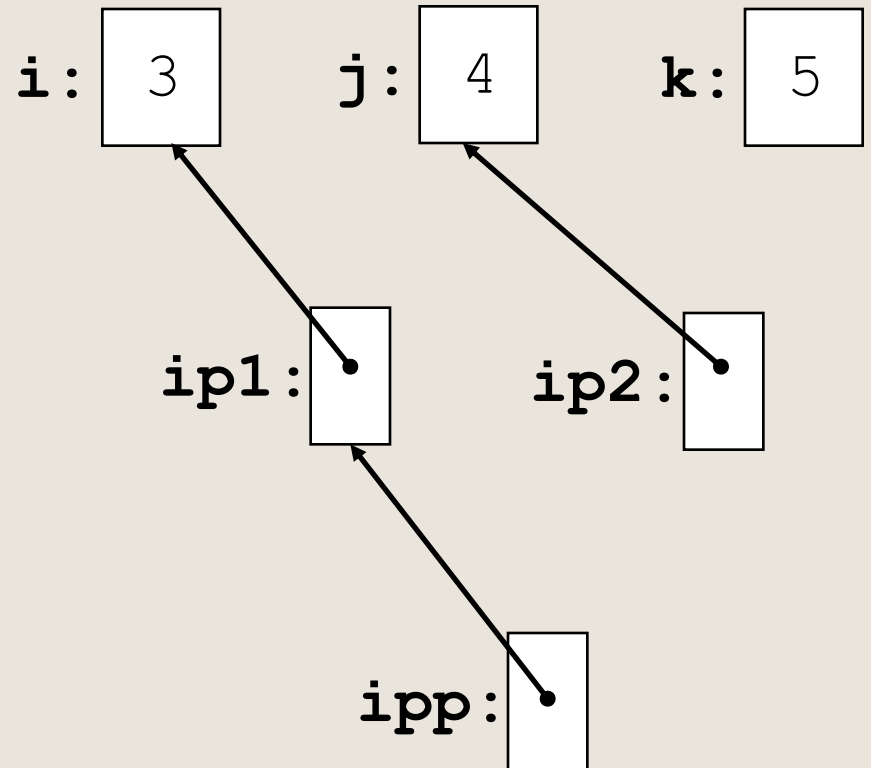
```
int j=4;
```

```
int k=5;
```

```
int *ip1 = &i;
```

```
int *ip2 = &j;
```

```
int **ipp = &ip1;
```



# Pointers to pointers

```
int i=3;
```

```
int j=4;
```

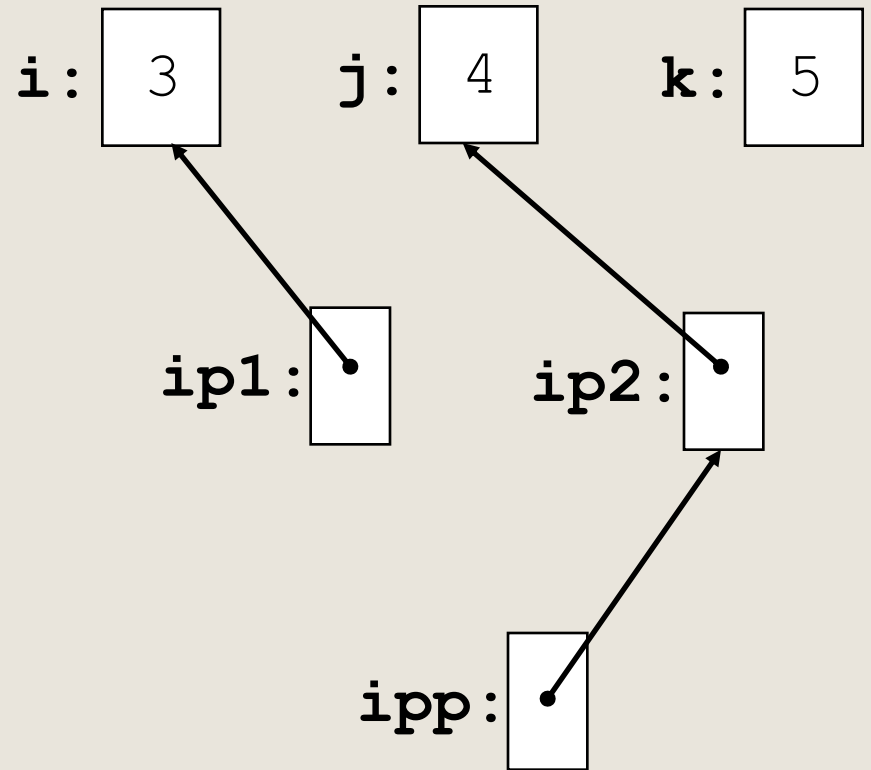
```
int k=5;
```

```
int *ip1 = &i;
```

```
int *ip2 = &j;
```

```
int **ipp = &ip1;
```

```
ipp = &ip2;
```



# Pointers to pointers

```
int i=3;
```

```
int j=4;
```

```
int k=5;
```

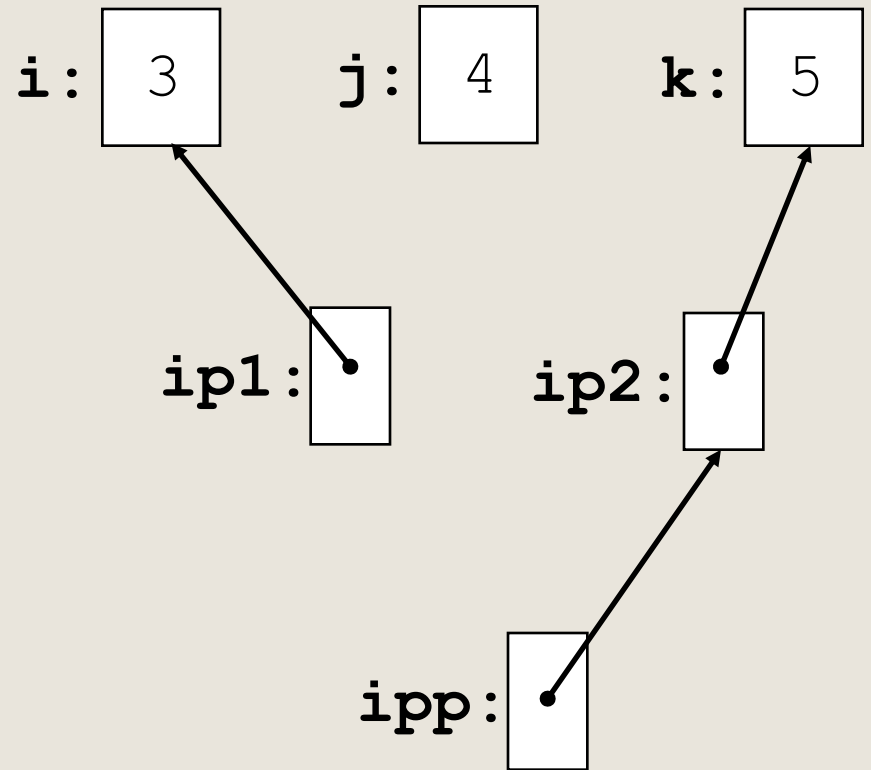
```
int *ip1 = &i;
```

```
int *ip2 = &j;
```

```
int **ipp = &ip1;
```

```
ipp = &ip2;
```

```
*ipp = &k;
```



# Reminder – the swap function

## Does nothing

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x, y;
    x = 3; y = 7;
    swap(x, y);
    // now x==3, y==7
}
```

## Works

```
void swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}

int main()
{
    int x, y;
    x = 3; y = 7;
    swap(&x, &y);
    // x == 7, y == 3
}
```

# Pointers to pointers: example

```
//put pointer to an allocated string in pp
int allocString( int len, char ** pp)
{
    char *str = (char*)malloc(len + 1);
    if (str==NULL)
    {
        return 1;
    }
    *pp = str;
    return 0;
}
```

## Pointers to pointers: example

```
// copy a string using "allocString"
```

```
void main()
```

```
{
```

```
    char *s = "example";
```

```
    char *copy;
```

```
    allocString( strlen(s), &copy );
```

```
    strcpy( copy, s);
```

```
    free (copy);
```

```
}
```

# Multi-dimensional arrays



# Multi-dimensional arrays

Can be created in few ways:

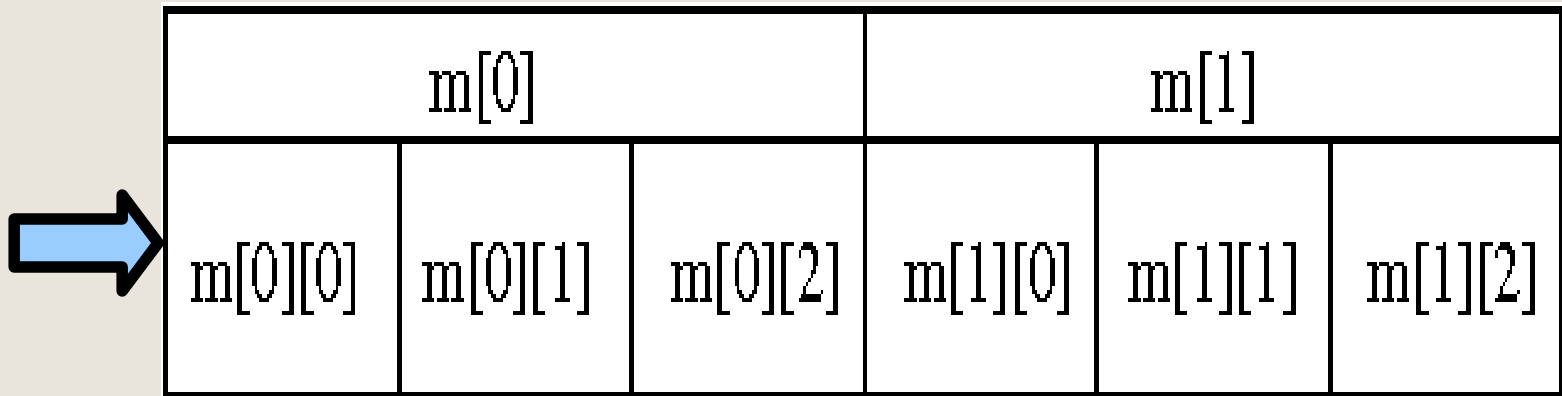
## 1. Automatically:

```
int m[2][3]; // 2 rows, 3 columns
```

- continuous memory (divided to 2 blocks)
- access: `m[row][col] = 0;`

# Automatic multi-dimensional arrays

```
#define R 2  
#define C 3  
int m[R][C];
```



```
int* m_ptr= &(m[0][0]);  
for (int i=0; i<R*C; ++i) {  
    printf(“%d\n”, *m_ptr++);  
}
```

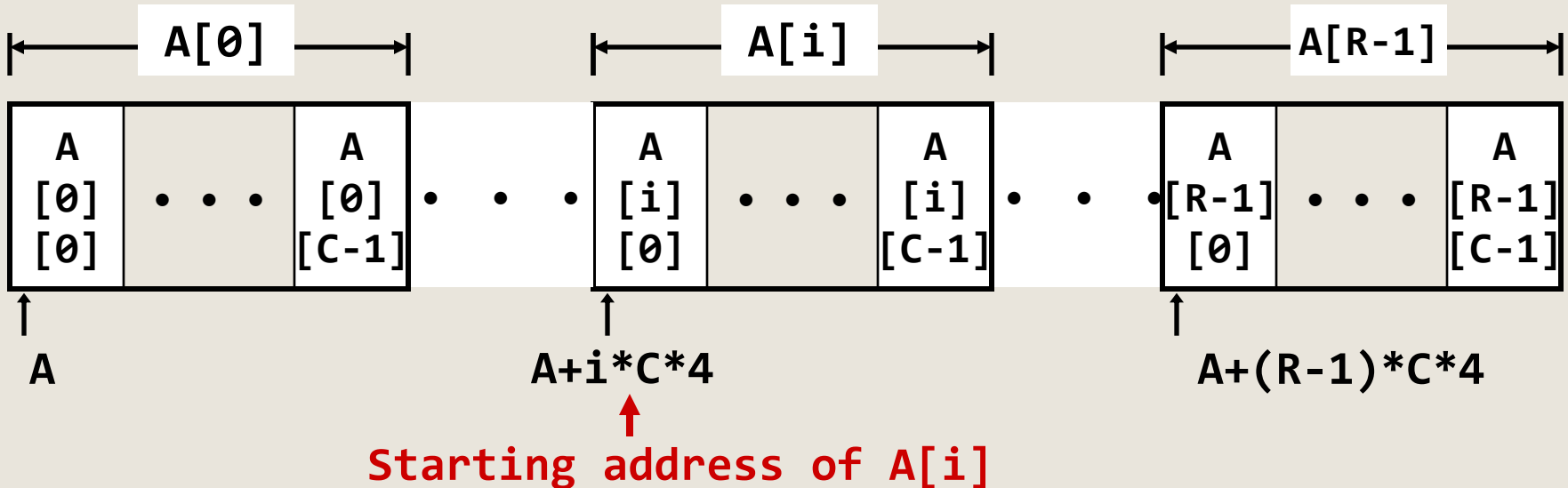
# Automatic multi-dimensional arrays

```
int A[R][C];
```

Like a matrix →

$A[0][0]$	$A[0][1]$	...	$A[0][c-1]$
...	...	...	...
$A[i][0]$	$A[i][1]$	...	$A[i][c-1]$
...	...	...	...
$A[R-1][0]$	$A[R-1][1]$	...	$A[R-1][C-1]$

Row-major ordering →



# Semi-dynamic multi-dimensional arrays

## 2. Semi-dynamic:

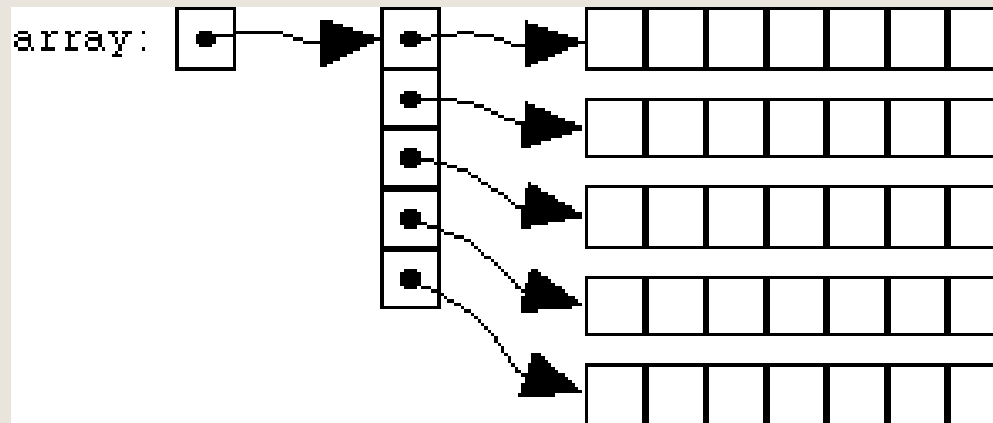
Define an array of pointers:

```
int *pa[5]; // allocates memory for 5 pointers
for (i=0; i<5; i++)
{
    pa[i] = (int*) malloc( 7*sizeof(int) );
    // pa[i] now points to a memory for 7 ints
}
```

# Dynamic multi-dimensional arrays

## 3. Dynamically:

```
int ** array;  
array = (int**) malloc( 5*sizeof(int*) );  
for (i=0; i<5; i++)  
{  
    array[i] = malloc( 7*sizeof(int) );  
}
```



# Semi/Dynamic multi-dimensional arrays

Memory not continuous

- Each pointer can be with different size
- Access: `arr[ i ][ j ]`
- Don't forget to free all the memory:

```
for (i=0; i<nrows; i++)
{
    free( array[i] );
}
free( array ); // only for dynamic
```

Dynamic arrays – more efficient way

```
int* A= (int*) malloc(R*C*sizeof(int));
```

**1-D array R=3 C=2 →**

0,0	0,1	0,2	1,0	1,1	1,2
-----	-----	-----	-----	-----	-----

- **Access:**  $A[ i*C + j ]$  //  $A [i][j]$ 
  - One access to memory vs. two in previous semi/dynamic representations.
  - Easier (& more efficient) implementation of iterators
- **But:**
  - Less readable code (can hide with macro or **much** better, inline functions)

# Pointers to pointers to ...

We also have pointers to pointers to pointers, etc.:

```
double ** mat1 = getMatrix();
double ** mat2 = getMatrix();
//allocate an array of matrices
double *** matrices =
(double*** ) malloc(n*sizeof(double**));
matrices[0] = mat1;
matrices[1] = mat2;
```



# Multi-dimensional arrays as arguments to functions

```
int arr[5][7]; // 5 rows, 7 columns
```

When sending 'arr' as an argument to a function, only the 1st index can be omitted:

- `void func( int x[5][7] ) //ok`
- `void func( int x[][7] ) //ok`
- `void func( int x[][] ) //error (always)`
- `void func( int * x[] ) //something else`
- `void func( int ** x ) //something else`

Why ?

Pointers to arrays (of size X):

```
int foo (char arr[][20]);
```

arr is a pointer to an array of 20 chars.

Therefore:

```
sizeof (arr) = sizeof (void*);
```

```
sizeof (*arr) = 20*sizeof (char);
```

# Pointers to arrays (of size X):

Explicit declaration:

```
char (*arr_2d)[20];  
        // arr_2d is a pointer  
        // not initialized
```

Using `typedef`:

```
typedef char arr_2d_20[20];  
arr_2d_20 *p_arr_2d;
```

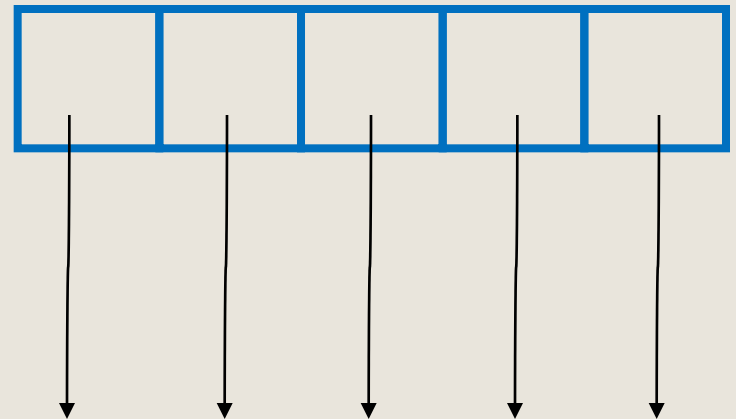
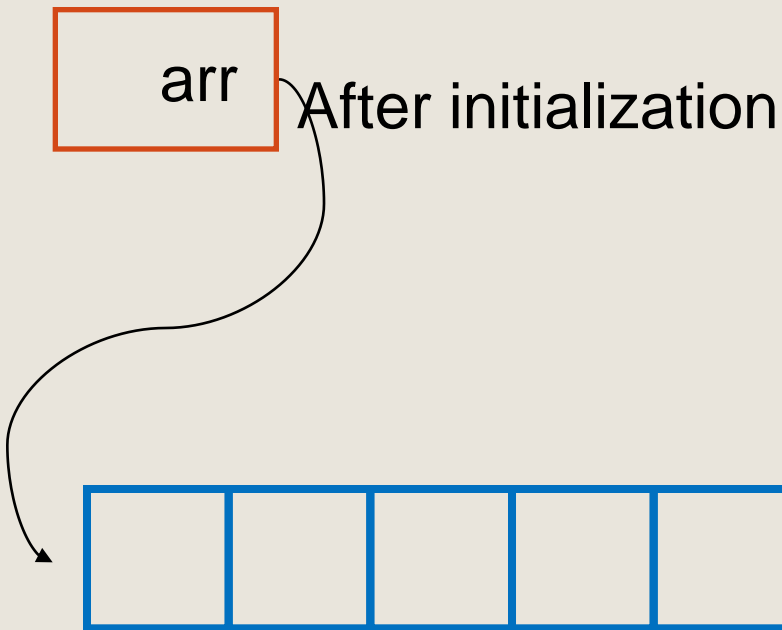
But:

```
typedef char arr_2d[][]; // comp. error
```

# Pointers to arrays (of size X):

```
char (*arr)[5];
```

```
char *arr[5];
```



# Pointers to arrays (of size X):

```
char (*arr)[5];
```

```
char *arr[5];
```

```
sizeof (arr) =  
    sizeof(void*)
```

```
sizeof (arr) =  
5*sizeof (char*) =  
5*sizeof (void*)
```

```
sizeof (*arr) =  
5*sizeof(char)
```

```
sizeof (*arr) =  
    sizeof (char*)=  
    sizeof (void*)
```

# Multi-dimensional arrays as arguments to functions

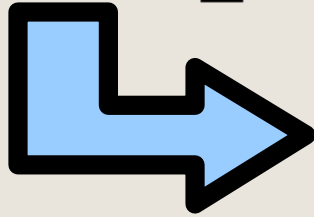
# Multi-dimensional arrays as arguments to functions

```
void foo( int matrix[ROWS_NUM][COLS_NUM] ) {
```



# Multi-dimensional arrays as arguments to functions

```
void foo( int matrix[ROWS_NUM][COLS_NUM] ) {
```



What is really sent?

# Multi-dimensional arrays as arguments to functions

```
void foo( int (*matrix)[COLS_NUM] ) {
```



**pointer to an array of  
COLS\_NUM ints!**

# Multi-dimensional arrays as arguments to functions

```
void foo( int (*matrix)[COLS_NUM] ) {
```

```
...
```

```
...     matrix[r][c]
```

# Multi-dimensional arrays as arguments to functions

```
void foo( int (*matrix)[COLS_NUM] ) {
```

```
...
```

```
...     matrix[r][c]
```

```
...     (*(matrix+r))[c]
```

# Multi-dimensional arrays as arguments to functions

```
void foo( int (*matrix)[COLS_NUM] ) {
```

...

... matrix[r][c]

... **(\*(matrix+r))[c]**



**matrix is a pointer to int[COLS\_NUM].  
addition is done in this units.**

**This is why COLS\_NUM is needed!**

# Multi-dimensional arrays as arguments to functions

```
void foo( int (*matrix)[COLS_NUM] ) {
```

```
...
```

```
...     matrix[r][c]
```

```
...     (*(matrix+r))[c]
```

```
...     *(*(matrix+r) + c)
```

# Multi-dimensional arrays as arguments to functions

```
void foo( int (*matrix)[COLS_NUM] ) {
```

```
...
```

```
...     matrix[r][c]
```

```
...     (*(matrix+r))[c]
```

```
...     *((*(matrix+r) + c) == *(matrix+r*COLS_NUM+c)
```



**The compiler is probably optimizing the internal computation to this.**

Example of dynamic multidimensional array: argv, argc


We want to pass parameters to the program running via the shell



# argv, argc

- ◆ To pass command line arguments to our program we should use the following **main** declaration:

```
main(int argc, char* argv[]) { ...
```



- ✓ `char** argv`
- ✗ `char argv[][]`

- ◆ Compare to `main(String[] args)` in java.
- ◆ Unlike java the **first argument** is the **name of the program** itself.

# argv & argc: example

◆ \$ prog1 -u danny -p 1234

argc == 5

argv[0] == "prog1"

argv[1] == "-u"

...

argv[4] == "1234"

**Always: argv[argc] == NULL**

(redundant since we are also given argc)