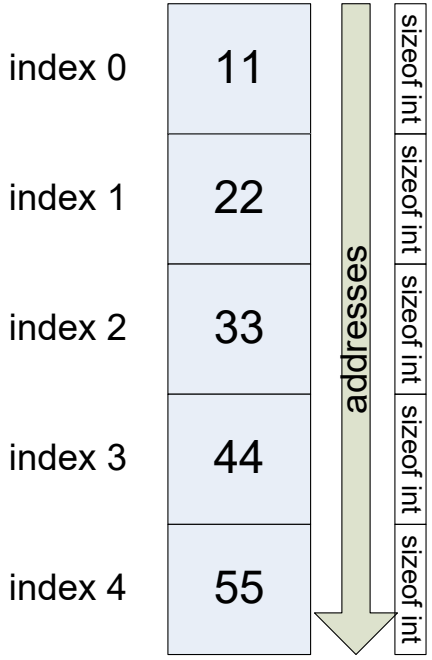
Arrays

- Array in C is very fuzzy construction.
- In effect, it represent memory block which is interpreted as if it contains some amount of consecutive elements of the same type
- Key aspect of array is its declaration:

  `element_type array_name[dimension] = {declaration_list};`
- By everything else, array is very similar to pointer, i.e. to some notion of pointer literal. For example, meaning of [] is the same

```
int array[5] = {11,22,33,44,55};

printf("element with index 3: %d\n", array[3])
```

Output: 44



Array initialization

- Same as with regular variables, if there is no explicit initialization arrays of static storage duration will be set to 0 (all of the elements), and of automatic storage duration will remain uninitialized.

```
int array[5];
```

local

NDF	NDF	NDF	NDF	NDF

global

0	0	0	0	0

- Element list in curly braces is used for initialization.
- The list can not have more element than array size. But if it has fewer, the remaining elements will be 0 (or NULL).

```
int array[5] = {1,3,5};
```

1	3	5	0	0

- If initialization list is given, array size can be omitted.
  In that case array size will be deduced from the size of initialization list

```
int array[] = {1, 2, 3}; <=> int array[3] = {1, 2, 3};
```

- In case that we want to initialize only some elements of an array, C99 offers a solution:

```
int array[100] = {[13] = 5, [77] = 6};
```

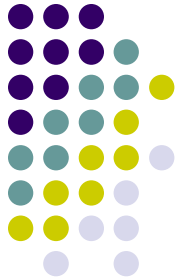
Relationship between pointers and arrays 1/3

- Arrays and pointers are closely related.

```
int array[] = {1,3,5,7};
int* ptr = array;
if (array[0] == ptr[0]
  && array[1] == ptr[1]
  && array[2] == ptr[2]
  && array[3] == ptr[3])
{
  printf("equal");
}
else
{
  printf("not equal");
}
```

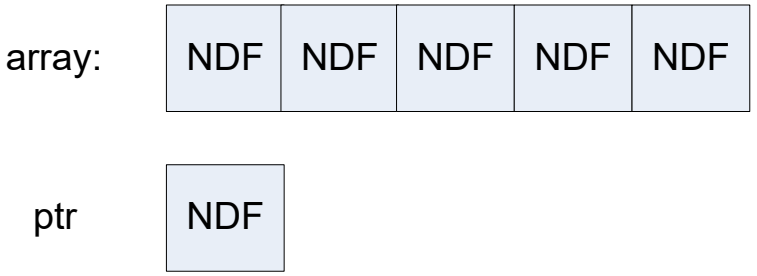
Output: equal

Relationship between pointers and arrays 2/3



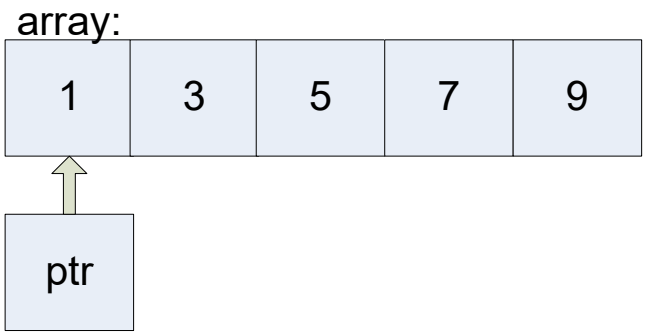
- ...but they are not the same thing.
- When array is defined, memory is allocated for all of its elements.
- When pointer is defined, resources are allocated only for itself.

```
char array[5];
char* ptr;
```



- Array, i.e. array name, represents an address
- Pointer is a variable that can store address as a value
- Roughly speaking, in this sense array (its identifier) is like address literal

```
int array[] = {1,3,5,7,9};
int* ptr = array;
```



Relationship between pointers and arrays 3/3

- Difference in the result of sizeof operation
  - For array, it returns size of the whole array
  - For pointer – just how much bytes are needed for an address

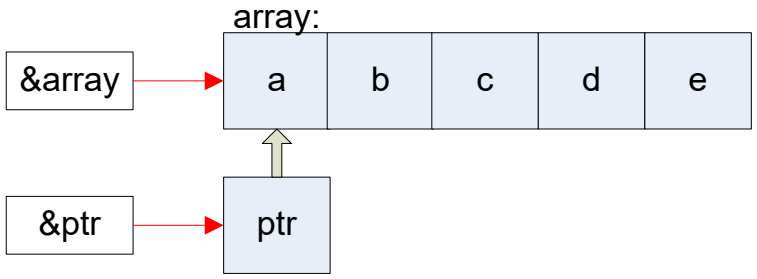
```
char array[15];
char* ptr;

printf("sizeof array: %d\n", sizeof(array));
printf("sizeof pointer: %d\n", sizeof(ptr));
```

Output:
sizeof array: 15
sizeof pointer: 4

- Difference in applying & operator
  - For array it returns adders of the first element
  - For pointer – address of the pointer

```
char array[] = {'a','b','c','d','e'};
char* ptr = array;
```

```
array[0] == ptr[0]
&array != &ptr
```



Multidimensional arrays

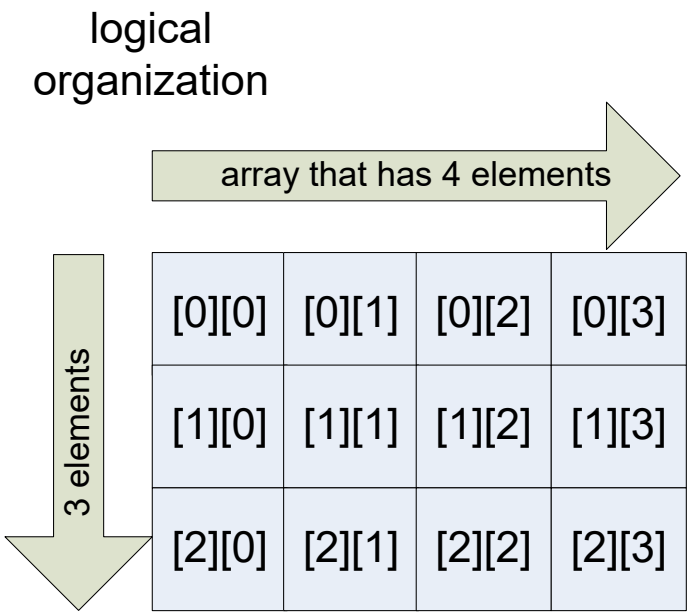
- In C, multidimensional arrays are actually arrays of arrays
- Example of two-dimensional array:

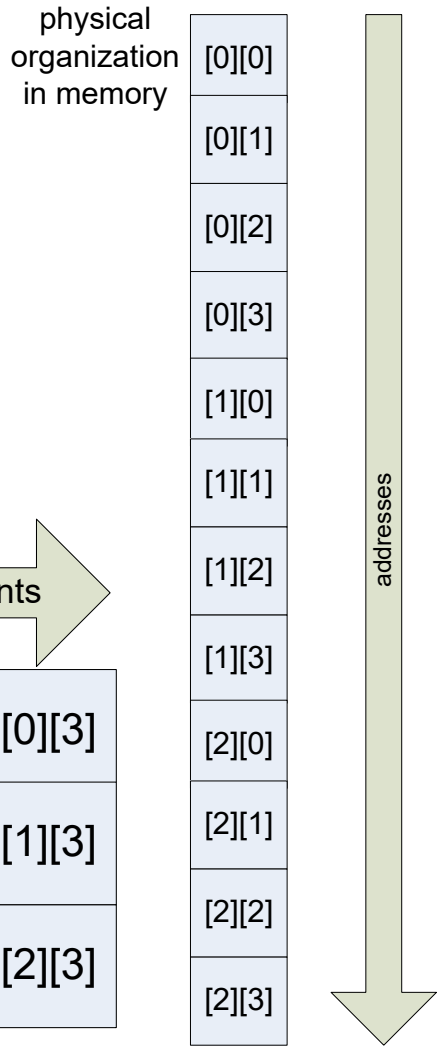
```
int matrix[3][4];
```

  - it's an array of 3 elements
  - and elements' type is array of 4 ints

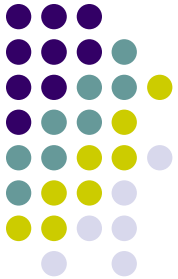
Equivalent definition:

```
typedef int niz4[4];
niz4 matrix[3];

int a;
niz4 y; <=> int y[4];

y[a]
*(y + a*sizeof(int))

int b;
niz4 x[3]; <=> int x[3][4];

x[a][b]
(x[a])[b]
(*(x+a*sizeof(niz4)))[b]
*(x+a*sizeof(niz4)) <=> niz4 T
T[b]
*(T+b*sizeof(int))
```





Passing arrays to functions



- Can not be done by value, in the full sense
- Always just the address is passed
- Consequence is that the following declarations are practically the same:

```
int func(int arr[10]);
int func(int arr[]);
int func(int* arr);
```

- Number of elements stated in squared brackets is ignored
- For multidimensional arrays, sizes have to defined for all except the first squared brackets

```
int func (int arr[][7]);
int func (int* arr[7]);
```

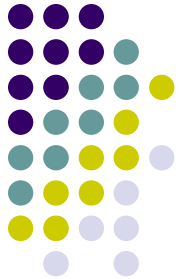
```
element_type name[] [depth1_] … [depth_n]
```

- Why?

```
typedef int niz4[4];
niz4 x[3]; <=> int x[3][4];
x[a][b]
*(T + b*sizeof(int)) // T <=> *(x + a*sizeof(niz4))
*(*(x + a*sizeof(niz4)) + b*sizeof(int))
```

Number 4 in declaration is important for calculating sizeof(niz4) - number 3 is not.

Array of function pointers



- And you can declare arrays of function pointers.

```
return_type (*name[]) (param_type, param_type);
```

```
char* (*fptr[3])(int x) = {func1, func2, func3};

char* pc = fptr[1](7);
char c = *pc;
c = *fptr[1](7); //?
```

```
typedef char* (*fptr_t)(int x);
fptr_t fptr[] = {func1, func2, func3};
```

- Useful for jump tables, or implantations of finite stat automata.

```
int (*fptr[])(int x) = {state1, state2, state3, state4}

int new_state(int current_state, int input)
{
  return fptr[current_state](input);
}
```

Array of function pointers

```
int new_state(int current_state, int input)
{
  switch (current_state)
  {
  case 0:
    return state1(input);
    break;
  case 1:
    return state2(input);
    break;
  case 2:
    return state3(input);
    break;
  case 3:
    return state4(input);
    break;
  }
}
```

```
int (*fptr[])(int x) = {state1, state2, state3, state4}

int new_state(int current_state, int input)
{
  return fptr[current_state](input);
}
```