Pointers & Arrays

CS2023 Winter 2004

Outcomes: Pointers & Arrays

- “C for Java Programmers”, Chapter 8, section 8.12, and Chapter 10, section 10.2
- Other textbooks on C on reserve
- After the conclusion of this section you should be able to
  - Identify how pointers and array are similar and how they are different
  - Use pointers to traverse C arrays (1D and 2D) and pointer subtraction to count how many elements have been traversed
  - Use pointers to pass portions of C arrays to functions

Arrays and Pointers

- A single-dimensional array is a *typed constant* pointer initialized to point to a block of memory that can hold a number of objects.

```
int id[1000];
int *pid;
```

id is an **int** pointer that points to a block of memory that can hold 1000 integer objects

pid is a pointer to **int**.

```
id = pid;
pid = id;
```

Arrays and Pointers

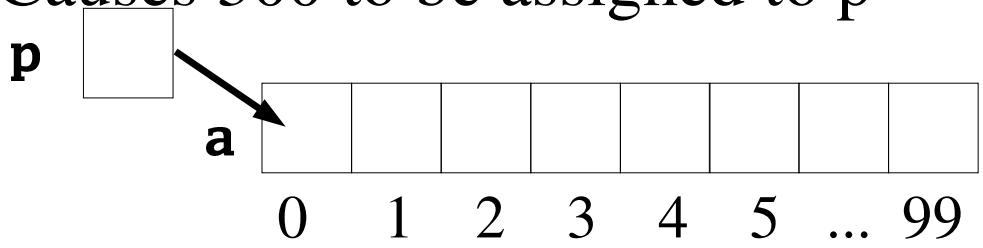
- When an array is declared, compiler allocates enough contiguous space in memory to contain all the elements of array

```
int a[100], *p;
```

- Base address of array is address of first element of array. Say bytes numbered 300, 304, 308, ..., 696 are alocated as addresses of a[0], a[1],... a[99]

`p = a;` is equivalent to `p = &a[0];`

Causes 300 to be assigned to p



Arrays Assignment and Pointers

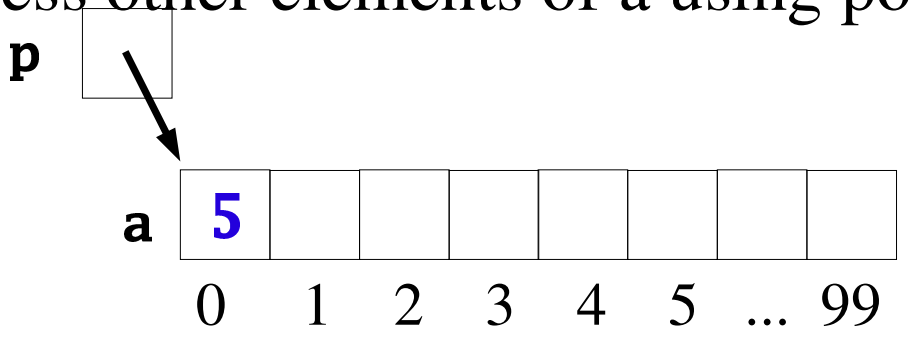
- Can access first element of an array using:

```
a[0] = 5;
```

- or through pointer:

```
p = a;

*p = 5;
```

- Can access other elements of a using pointer arithmetic



Pointer Arithmetic

Valid operations on pointers include:

- the *sum* of a pointer and an integer
- the *difference* of a pointer and an integer
- pointer *comparison*
- the *difference* of two pointers.

Sum of Pointer and Integer

- To access other objects in the block of memory pointed to **`p`**, use

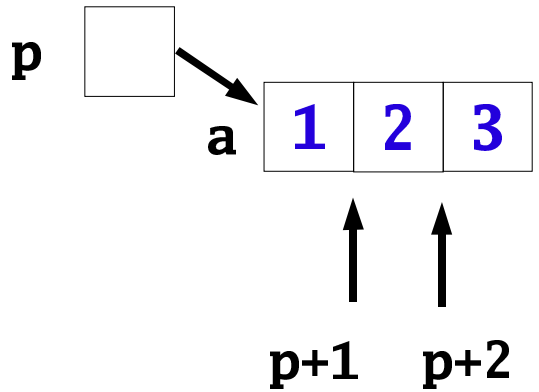
  `p + n`

  where **`n`** is an integer. This expression yields a pointer to the n-th *object* beyond the one that **`p`** currently points to.

- The exact meaning of "object" is determined by the type of **`p`**.

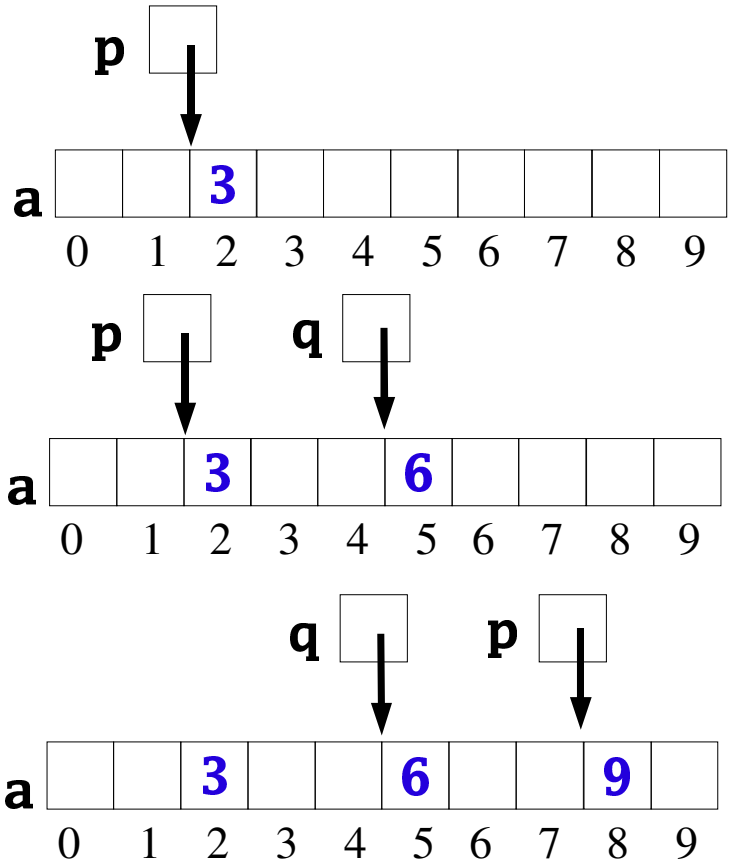
Sum of Pointer and Integer

```
int a[3], *p;
p = a;
*p = 1;
*(p + 1) = 2;
*(p + 2) = 3;
```



Sum of Pointer and Integer

```
int a[10], *p, *q;

p = a + 2;
*p = 3;

q = p + 3;
*q = 6;

p += 6;
*p = 9;
```



Sum of Pointer and Integer: *i*th object

To access *the i-th object* in a memory block:

```
*(p+i)
```

or

```
p[i]
```

`p[0]` and `*p` are equivalent.

Sum of Pointer and Integer: *i*th object

```
#define SIZE 3
double a[SIZE],*p;

p = a;
for(i = 0; i < SIZE; i++)
  if(scanf("%lf", p+i) == 0) /* no & */
    exit(EXIT_FAILURE);

*(p+SIZE) = 1.2;
```

Difference of Pointer and Integer

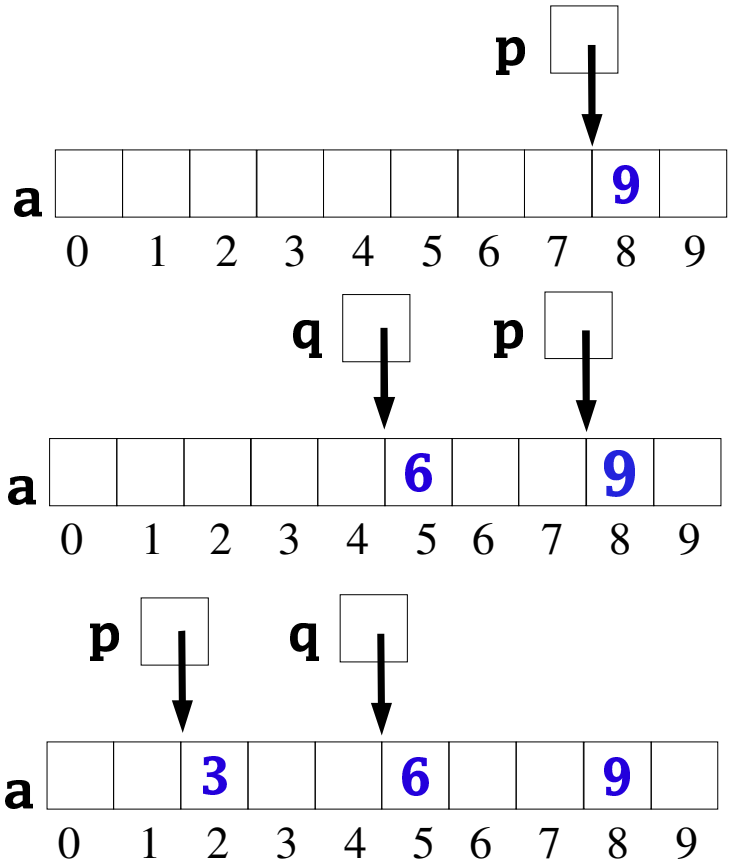
Often, we need to access objects *preceding* the object pointed to by a pointer `q`.

```
q - n
```

yields a pointer to the n-th *object before* the one that `q` currently points to.

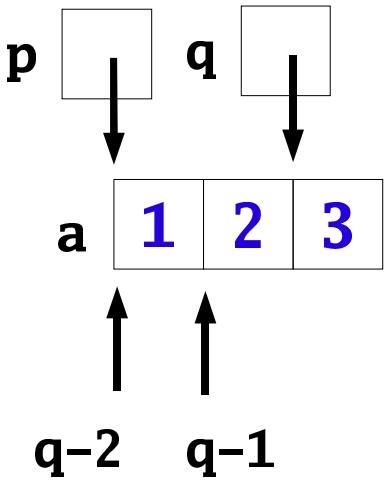
Difference of Pointer and Integer

```
int a[10], *p, *q;

p = a + 8;
*p = 9;

q = p - 3;
*q = 6;

p -= 6;
*p = 3;
```



Example

```
#define SIZE 3
 int *p, *q, a[SIZE];
 p = a;
 … /* initialize the block */
 /* output in reverse order */
 for(i = 0, q = p+SIZE-1; i < SIZE; i++)
    printf("%f\n", *(q-i));
```



Careful!

Given a block of memory of `SIZE` objects, pointed to by `p`, you can set `q` to point to the last object in the block using:

```
p+SIZE-1
```

rather than

```
p+SIZE
```

('off by one' error).

Pointer Comparison

- Two pointers of the same type, **`p`** and **`q`**, may be compared as long as *both of them* point to objects within a *single* memory block

- Pointers may be compared using the <, >, <=, >=, == , !=

- When you are comparing two pointers, you are comparing the values of those pointers *rather than* the contents of memory locations pointed to by these pointers

Pointers and traversals

Assuming

```
double *p, *pi;
```

and `p` pointing to an array of `SIZE` doubles:

```
for(pi = p, product = 1; pi < p+SIZE; pi++)
        product *= *pi;

/* print backwards */
for(pi = p+SIZE-1; pi >= p; pi--)
        printf("%f\n", *pi);
```

Block Traversal Idiom

```
for(pi = p; pi < p+SIZE; pi++)
    use pi here
```

Example: largest element

```
/* Return the largest element in array p */
double find_largest(double p[], int n)
{
 double *max, *pi;

 for(max = p, pi = p+1; pi < p+n; pi++)
    if(*max < *pi)
       max = pi;

 return *max;
}
```

Example: block copy

Copy the contents of a block (or array) pointed to by **p** of size **SIZE** to another block pointed to by **q**:

```
double *pi, *qi;
double p[SIZE], q[SIZE]

for(qi = q, pi = p; qi < q+SIZE; qi++,
  pi++)
    *qi = *pi;
```

Pointer Subtraction

Given two pointers, **`p`** and **`q`**, such that:

- both pointers are of the same type,
- **`p`** > **`q`**
- both point to objects in a single memory block,

the expression

```
p - q
```

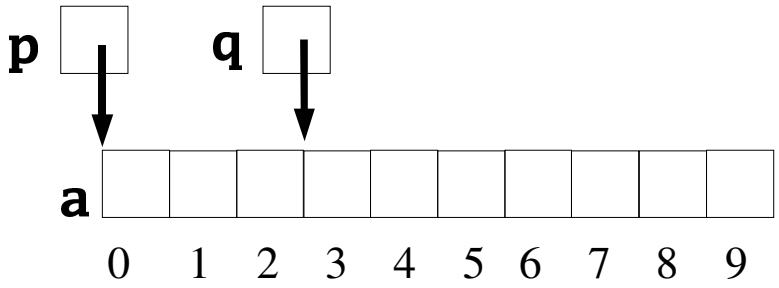
yields the number of *objects* between **`p`** and **`q`**,

including the object pointed to by **`q`**.

The type of the result of pointer difference is **`ptrdiff_t`**, defined in **`stddef.h`**.

Pointer Subtraction

```
int a[10], *p, *q;

p = a;
q = a + 3;

printf("%d", q-p);
```



Output: 3

Example

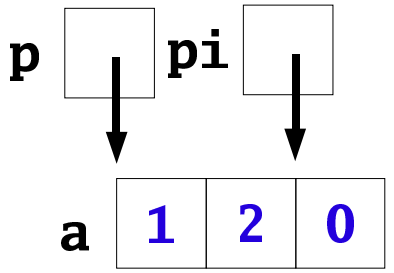
Find the *first* occurrence of the value **0** in a block of integers (**EOF** if not found):

```
int position;
int *p, *pi;
/* p initialized */
...
for(pi = p; pi < p+SIZE; pi++)
    if(*pi == 0)
        break;
position = (pi == p+SIZE) ? EOF : pi-p+1;
```

Example

Find the *first* occurrence of the value **0** in a block of integers (**EOF** if not found):

```
int position;
int *p, *pi;
/* p initialized */
...
for(pi = p; pi < p+SIZE; pi++)
    if(*pi == 0)
       break;
position = (pi == p+SIZE) ? EOF : pi-p+1;
```



position = 3

Arrays as Parameters

When arrays are used as function parameters, they are actually

treated as pointers. The following two declarations are equivalent:

```
int maxiA(double arr[], int size);

int maxiP(double *arr, int size);
```

The second parameter is *necessary* to specify the size of the array.

```
int readArray(int x[], int size) {
    int *pi;

    for(pi = x; pi < pi + size; pi++)
        if(scanf("%d", pi) != 1)
            return 0;
    return 1;
}

void printArray(int x[], int size) {
    int *pi;

    for(pi = x; pi < pi + size; pi++)
        printf(%d", *pi);
}
```

```
/* Applications of readArray and printArray */
#define SIZE 20
double d[SIZE];

if(readArray(d, SIZE) == 0)
  error;
printArray(d, SIZE);

printArray(d, SIZE - 10);    /* prefix */

printArray(d + 2, SIZE - 2);  /* suffix */

printArray(d + 2, 5);         /* segment */
```

Are Arrays and Pointers Interchangeable?

- Array *parameters* are interchangeable with pointer parameters, but array variables are not the same as pointer *variables*.
- C compiler converts name of an array to a constant pointer when necessary
- sizeof(array) returns number of bytes occupied by array, whereas sizeof(pointer) returns number of bytes used to store pointer

Are Arrays and Pointers Interchangeable?

```
int id [1000];
int *pointerId;
```

`sizeof(id)` is **`1000*sizeof(int)`**

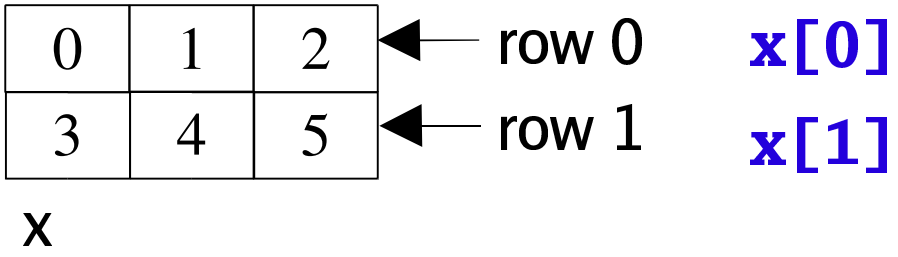
`sizeof(pointerId)` is the number of *bytes* used to store a pointer to an **`int`**.

Pointers & Multi-Dimensional Arrays

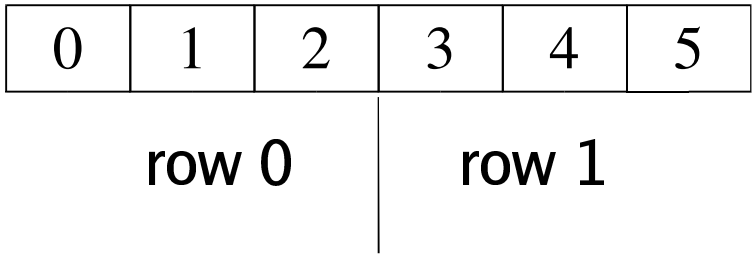
`type arr[s1][s2];`

- example:

  `int x[2][3];`



- Arrays stored in row-major order



Pointers & Multi-Dimensional Arrays

```
int a[NUM_ROWS][NUM_COLS];
int row, col;

for(row = 0; row < NUM_ROWS; row++)
  for(col = 0; col < NUM_COLS; col++)
     a[row][col] = 0;
```

View **a** as a one-dimensional array of integers:

```
int *p;

for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1];
     p++)
  *p = 0;
```

Pointers & Multi-Dimensional Arrays

Point to row **`i`** in array **`a`**:

```
int *p;
p = &a[i][0];
```

or:

```
p = a[i];
```

since **`a[i]`** is the **`i`**th row of **`a`**

contrast with single-dimensional array where **`a[i]`** equivalent to **`*(a+i)`**

Pointers & Multi-Dimensional Arrays

Clear row **i** of array **a**:

```
int int a[NUM_ROWS][NUM_COLS], *p, i;

for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

Can pass **a[i]** to a function that is expecting a one-dimensional array as its argument. Find largest element in row **i** of **a**:

```
largest = find_largest(a[i], NUM_COLS);
```

Pointers & Multi-Dimensional Arrays

Caution:

- The base address of a two-dimensional array `a` is `&a[0][0]`, not `a`.
- The array name `a` by itself is equivalent to `&a[0]`

Storage Mapping Function

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

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

- Note that column size is required in storage mapping function, which is why it must be specified in function declaration

```
int sum(int a[][NCOLS], NROWS)
```