

60-141-02 LECTURE 3: ARRAYS

Edited by Dr. Mina Maleki

Outline

2

- Arrays
- Multi-Dimensional Arrays
- Passing Arrays to Functions
- General Array-based Algorithms
 - ▣ Searching Array Elements
 - ▣ Sorting Array Elements

Array Definition

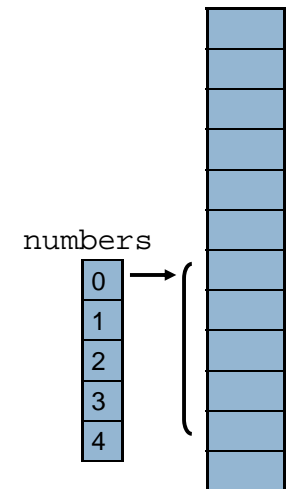
3

- An **array** is a data structure containing a number of data values, all of which have the same type.
- Two requirements:
 - ▣ All data values, known as **elements**, are of the same DATA TYPE
 - ▣ Each element has a unique INDEX (position) to access it
- The simplest kind of array has just one dimension.
- The elements of a one-dimensional array **a** are conceptually arranged one after another in a single row (or column):



Array Definition ...

- The array name is the same name used for all elements.
- It helps to remember that the array name acts as a pointer to the first element in the array, and array elements are sequentially arranged one after the next.
- Example
 - ▣ To define an integer array called **numbers** of size 5:
 - `int numbers[5];`



Array Declaration

5

- Declaring an array variable:
`Data-type <variable_name>[size];`
 - Declaring the **type of elements** (int, float, char, ...)
 - Declaring the **maximum number of elements** that will be stored inside the array (Length of array)
- The elements may be of any type
- The length of the array can be any (integer) constant expression.
- Example
 - To define an integer array called `numbers` of size 5:
 - `int numbers[5];`

Array Declaration

6

- The range for valid index values in C:
 - **First** element is at index **0**
 - **Last** element is at index **[size-1]**
- It is the task of the programmer to make sure that array elements are referred by indexes that are in the valid range !
 - The compiler cannot verify this, and it comes to severe runtime errors !

Array Declaration Scenario

7

- Now consider the declaration
 - `int A [9] ;`
- The entire allocation unit is called **A** – the array name
- There must be 9 integer sized allocations in RAM
- Each element is located contiguously (in sequence and “touching”)



Invalid Array Declarations

8

- `int MyArray;`
 - **X This is a variable declaration**
- `int MyArray();`
 - **X This is a function prototype**
- `int MyArray[];`
 - **X Invalid size for array (must initialize or provide size)**
- `int MyArray[- 10];`
 - **X Array size must be a positive integer**

Array Initialization

- The most common form of **array initializer** is a comma-separated list of constant enclosed in braces
 - `int a[10] = {1,2,3,4,5,6,7,8,9,10};`
 - `char letters[5] = {'a','b','c','d','e'};`
- It is illegal for an initializer to be completely empty.
- If an initializer is present, the length of the array may be omitted:
 - `int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};`
 - `char name[] = { 'a', 'l', 'i'};`
 - Size of an array is automatically set as the number of elements within `{ }`

Array Initialization ...

- The size of array implied based on the number of values
 - ▣ in the {list} .
 - ▣ OR in the [].
- `int A[5] = {1, 2, 3, 4, 5};`
 - ▣ Declares and initialize an array of size 5
- `int A[] = { 10, 20, 30, 40 };`
 - ▣ Declares and initializes an array of size 4

Array Initialization ...

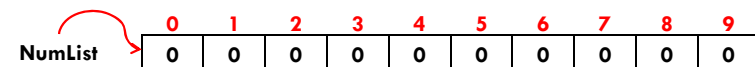
- Initializing part of an array, and other numbers will set as 0 if the initializer is shorter than the array.
- Example
 - `float sample_data[5] = { 10.0, 30.0, 50.5 };`
 - Size of array = 5

10.0	30.0	50.5	0	0
------	------	------	---	---
 - `int a[10] = {3,1};`
 - Size of array = 10

3	1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---
- Note: It is illegal for an initializer to be longer than the array it initializes.

Example

- initialize an array to all zeros
- `int NumList[10] = {0,0,0,0,0,0,0,0,0,0};`
- `int NumList[] = {0,0,0,0,0,0,0,0,0,0};`
- `int NumList[10] = {0};`
- Declares an array called NumList of type int and size 10.
- At the same time, the elements are initialized to 0



Designated Initializers (C99)

13

- **Designated initializers** can be used if only few elements of an array need to be initialized explicitly

```
int a[15] = {0,0,29,0,0,0,0,0,0,0,7,0,0,0,0,48};
```
- Using a designated initializer:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```
- Order of elements is not important

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```
- Each number in brackets is said to be a **designator**.
- Designated initializers are shorter and easier to read.

Designated Initializers ...

14

- Designators must be integer constant expressions.
- If the array being initialized has length n , each designator must be between 0 and $n - 1$.
- If the length of the array is omitted, a designator can be any nonnegative integer.
 - ▣ The compiler will deduce the length of the array from the largest designator.
 - ▣ Example: The following array will have 24 elements:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

Example

15

- ```
int a[10] = { [0] = 3, [7] = 1};
```

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|
- ```
int x=3;
```

```
int a[10] = { [0] = x+2, [5] = 1, [9] = 8};
```

5	0	0	0	0	1	0	0	0	8
---	---	---	---	---	---	---	---	---	---
- ```
int c[10] = {5, 1, 9, [4] = 3, 7, 2, [8] = 6};
```

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

## Accessing Array Elements

16

- To access an array element, write the **array name** followed by an integer value in square brackets, known as **index** (or position).  $A[i]$ ,  $\text{num}[3]$ ,  $\text{val}[6]$
- This is referred to as **subscripting** or **indexing** the array.
- The elements of an array of length  $n$  are indexed from 0 to  $n - 1$ .
- If  $a$  is an array of length 10, its elements are designated by  $a[0]$ ,  $a[1]$ , ...,  $a[9]$ :

|        |        |        |        |        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |        |        |        |        |
| $a[0]$ | $a[1]$ | $a[2]$ | $a[3]$ | $a[4]$ | $a[5]$ | $a[6]$ | $a[7]$ | $a[8]$ | $a[9]$ |

## Example 1

17

- Declaration: `int A[10];`
- Expressions of the form `A[i]` are lvalues, so they can be used in the same way as ordinary variables.
- `A[0] = 10;`
  - ▣ Assigns the value of 10 to the first element in the array → index = 0
- `printf("%d\n", A[5]);`
  - ▣ Prints the 6th element of array
- `++A[i];`
  - ▣ Increments the value of the ith index
- `A [ J ] < A [ K ]`
  - ▣ relational expression

## Example 2

18

```
int values[10];
```

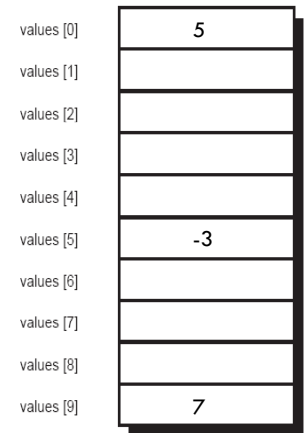
Valid indexes:

```
values[0] = 5;
value [5] = -3;
values[9] = 7;
```

Invalid indexes:

```
values[10] = 3;
values[-1] = 6;
```

In memory: elements of an array are stored at consecutive locations



## Working with Array Elements

19

- Many programs contain `for` loops whose job is to perform some operation on every element in an array
  - ▣ Read data into an array
  - ▣ Populate the array
  - ▣ Sum
  - ▣ Find the maximum number and its index
  - ▣ Swapping
  - ▣ Reverse the array data
  - ▣ ...

## Read data into an array

20

- Get a set of numbers from the user

```
int nums[15]; // array declaration of size 15
int i; // i is used as an element index
for (i = 0; i < 15; i++)
 scanf("%i", &nums[i]);
```

## Populate the array

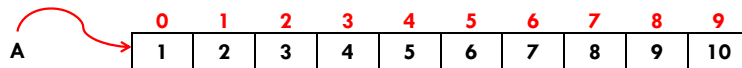
21

- Populate the array A with 1,2,3,...

```
int A[10]; /* declare array A */
A[0] = 1;
A[1] = 2;
A[2] = 3;
A[3] = 4;
A[4] = 5;
A[5] = 6;
A[6] = 7;
A[7] = 8;
A[8] = 9;
A[9] = 10;
```

**The hard way**

```
for (int i = 0; i < 10; i++)
 A[i] = i+1;
```



## Populate the array ...

22

- Populate the array A with 5's  
for (int i = 0; i < 10; i++)  
    A[i] = 5;
- Populate the array A in reverse order starting with 5 to -4.  
eg. {5, 4, 3, 2, 1, 0, -1, -2, -3, -4}  
for (int i = 0; i < 10; i++)  
    A[i] = 5-i;
- Populate the array A with {10, 15, 20, ...}  
for (int i = 0; i < 10; i++)  
    A[i] = (5 \* i) + 10;

## Sum

23

```
/* SUM */
int A[] = { 5, 3, 2, 7, 1};
int i, sum = 0;
for (i = 0; i < 5; i++)
{
 sum = sum + A[i]; //Sum += A[i];
}
printf("%d", sum);
```



## Find Maximum

24

```
/* Find Index of the Maximum Number*/
int A[5] = { 5, 3, 2, 7, 1};
int i, max ;
max = A[0];

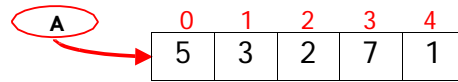
for (i = 1; i < 5; i++)
{
 if (A[i] > max)
 max = A[i];
}
printf("%d", max);
```



## Find Index of Maximum

25

```
/* Find the Index of the Maximum Number */
int A[] = { 5, 3, 2, 7, 1};
int i, max, index ;
max = A[0];
index = 0;
for (i = 1; i < 5; i++) {
 if (A[i] > max){
 max = A[i];
 index = i;
 }
}
printf("A[%d] = %d is max", index, max);
```



## Swapping

26

```
/* SWAPPING element values */
int A[] = { 5, 3, 2, 7, 1};
int i, temp;
temp = A[0];
A[0] = A[1];
A[1] = temp;
printf("%d %d", A[0], A[1]);
```



## Reversing a Series of Numbers

27

- The `reverse.c` program prompts the user to enter a series of numbers, then writes the numbers in reverse order:

Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31  
In reverse order: 31 50 11 23 94 7 102 49 82 34

- The program stores the numbers in an array as they're read, then goes through the array backwards, printing the elements one by one.

## Reversing a Series of Numbers ...

28

```
/* Reverses a series of numbers */
#include <stdio.h>
#define N 10

int main(void)
{
 int a[N], i;

 printf("Enter %d numbers: ", N);
 for (i = 0; i < N; i++)
 scanf("%d", &a[i]);

 printf("In reverse order:");
 for (i = N - 1; i >= 0; i--)
 printf(" %d", a[i]);

 return 0;
}
```

## Fibonacci Numbers

$$F_n := F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

|      |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|
| n    | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| F(n) | 0 | 1 | 1 | 2 | 3 | 5 | 8 |

## Fibonacci Numbers...

```
// Program to generate the first 15 Fibonacci numbers
#include <stdio.h>
int main() {
 int f[15], i;
 f[0] = 0; //set initial values
 f[1] = 1; //set initial values

 for (i = 2; i < 15; i++)
 f[i] = f[i - 2] + f[i - 1];

 for (i = 0; i < 15; i++)
 printf("%i\n", f[i]);
 return 0;
}
```

## Common Mistakes

- Forgetting that an array with  $n$  elements is indexed from 0 to  $n - 1$ , not 1 to  $n$ :

```
int a[10], i;

for (i = 1; i <= 10; i++)
 a[i] = 0;
```

- With some compilers, this innocent-looking for statement causes an infinite loop.

## Multi-Dimensional Arrays



## Multi-Dimensional Arrays

33

- in C, an array may have any number of dimensions.
- 2-D array declaration: (eg. grid or table)
  - `int B[3][4];`
- 3-D array declaration: (eg. cube)
  - `int D[5][5][5];`

## Two-dimensional Arrays

34

- Declaration: **Data\_Type arrayName [nRows][nCols]**
- The following declaration creates a two-dimensional array:
 

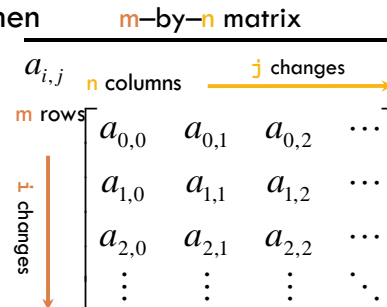
```
int m[5][9];
```
- m has 5 rows and 9 columns.
- Both rows and columns are indexed from 0:

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 |   |   |   |   |   |   |   |   |   |
| 1 |   |   |   |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |   |   |   |
| 3 |   |   |   |   |   |   |   |   |   |
| 4 |   |   |   |   |   |   |   |   |   |

## Accessing Array Elements

35

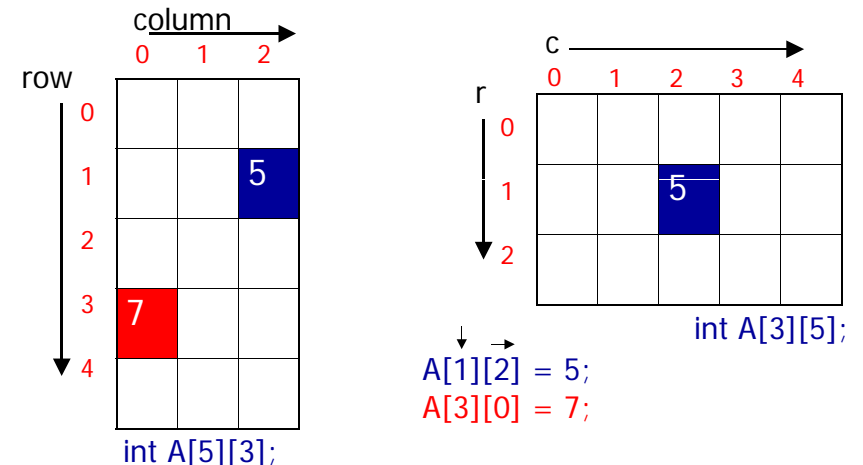
- To access the element of m in row i, column j, we must write `m[i][j]`.
- The expression `m[i]` designates row i of m, and `m[i][j]` then selects element j in this row.
- Wrong access:
  - `m[i, j]`
  - `m(i)(j)`



## Accessing Array Elements ...

36

- Accessing Elements in a 2D array



## Example 1

37

- Nested for loops are ideal for processing multidimensional arrays.

- Example:

```
int num[4][4];
int row, col;

for (row = 0; row < 4; row++)
 for (col = 0; col < 4; col++)
 {
 if (row == col)
 num[row][col] = 1;
 else
 num[row][col] = 0;
 }
```

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

## Example 2

38

- Initialize the grid to serial numbers

```
int A[3][5];
int r, c, count=1;
for (r = 0; r < 3; r++)
{
 for (c = 0; c < 5; c++)
 {
 A[r][c] = count;
 count++;
 }
}
```

|   |     |    |    |    |    |
|---|-----|----|----|----|----|
|   | c → |    |    |    |    |
|   | 0   | 1  | 2  | 3  | 4  |
| 0 | 1   | 2  | 3  | 4  | 5  |
| 1 | 6   | 7  | 8  | 9  | 10 |
| 2 | 11  | 12 | 13 | 14 | 15 |

## Initializing a 2D Array

39

- Initializing by nesting one-dimensional initializers:

```
int m[4][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
 {0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 1, 0, 1, 1, 0, 0, 1, 0},
 {1, 1, 0, 1, 0, 0, 0, 1, 0}};
```

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0.

```
int m[4][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
 {0, 1, 0, 1, 0, 1, 0, 1, 0}};
```

- It fills only the first two rows of m;
- The last two rows will contain zeros

## Initializing a 2D Array ...

40

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[4][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
 {0, 1, 0, 1, 0, 1, 0, 1, 1},
 {0, 1, 0, 1, 1, 0, 0, 1, 1},
 {1, 1, 0, 1, 0, 0, 0, 1, 1}};
```

- We can even omit the inner braces:

```
int m[4][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
 0, 1, 0, 1, 0, 1, 0, 1, 0,
 0, 1, 0, 1, 1, 0, 0, 1, 0,
 1, 1, 0, 1, 0, 0, 0, 1, 0};
```

Once the compiler has seen enough values to fill one row, it begins filling the next.

## Initializing a 2D Array ...

41

- C99's designated initializers work with multidimensional arrays.

### Example:

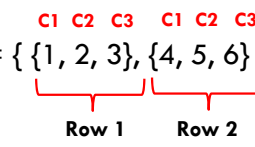
```
double A[3][3] = {[0][0] = 1.0, [1][1] = 5.7};
```

- All elements for which no value is specified will default to zero.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 5.7 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

## Example 1

42

- `int array1[2][3] = { {1, 2, 3}, {4, 5, 6} };`  

- `int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };`
- `int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };`
- `int array4 [2][3] = {[0][0] = 1,[1][2]=3}`

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 0 |

|   |   |   |
|---|---|---|
| 1 | 2 | 0 |
| 4 | 0 | 0 |

|   |   |   |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 0 | 3 |

## Example 2 – Output ?

43

```
int A[2][3]={ {1,2,3}, {3,4,5} };
int B[2][3]={ {2,2,2}, {5,4,3} }, C[2][3];
int i,j, N=2, M=3;
for (i=0;i<N;i++){
 for (j=0;j<M;j++){
 if (A[i][j]>B[i][j])
 C[i][j]=A[i][j];
 else
 C[i][j]=B[i][j];
 }
}
for (i=0;i<N;i++) {
 for (j=0;j<M;j++){
 printf("%d ",C[i][j]);
 printf("\n");
 }
}
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{bmatrix}^A$$

$$\begin{bmatrix} 2 & 2 & 2 \\ 5 & 4 & 3 \end{bmatrix}^B$$

$$\begin{bmatrix} 2 & 2 & 3 \\ 5 & 4 & 5 \end{bmatrix}^C$$

2 2 3  
5 4 5

## Memory Allocation

44

- Memory allocation of 2D arrays is done (typically) using Row Major Ordering.

- The compiler must know, in advance, what size of allocation to provide for each row
- The actual number of rows is not important

a

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

|         |         |         |
|---------|---------|---------|
| a[0][0] | a[0][1] | a[0][2] |
| a[1][0] | a[1][1] | a[1][2] |
| a[2][0] | a[2][1] | a[2][1] |
| a[3][0] | a[3][1] | a[3][2] |

|         |    |
|---------|----|
| a[0][0] | 1  |
|         | 2  |
|         | 3  |
| a[1][0] | 4  |
|         | 5  |
|         | 6  |
| a[2][0] | 7  |
|         | 8  |
|         | 9  |
| a[3][0] | 10 |
|         | 11 |
| a[3][2] | 12 |

Memory

## Scalability Tip

45

- To make your array more flexible and easy to modify, avoid hardcoding the size of the array in your code.
- One way is to use the `#define` directive to define a **symbolic constant** representing the desired size.
- `#define` directives are located at the beginning of the program, after `#include` directives
- Example:

**For changing MAX from 10 to 1000, you do not change anything else in your code!**

```
#include <stdio.h>
#define MAX 10
void main(){
 int StudentList[MAX];
 int i;
 for (i = 0; i < MAX; i++)
 StudentList[i] = 0;
}
```

## Example

46

```
#include <stdio.h>
#define N 3
#define M 4
int main(void) {
 int a[N][M];
 int i,j;
 /* read matrix elements */
 for(i = 0; i < N; i++)
 for(j = 0; j < M; j++)
 {
 printf("a[%d][%d] = ", i, j);
 scanf("%d", &a[i][j]);
 }
 /* print matrix elements */
 for(i = 0; i < N; i++)
 {
 for(j = 0; j < M; j++)
 printf("%5d", a[i][j]);
 printf("\n");
 }
 return 0;
}
```

## Passing Array to Functions

## Passing Array to Functions

48

- Arrays are passed to functions by **REFERENCE**.
- Passing by reference involves passing a pointer to the first element (such as the name of the array).
- **Advantage:** Faster, more efficient on memory usage since the array is not copied and passed to the function. Array elements can be manipulated by the function and the change persists even after the called function returns.
- **Disadvantage:** When a function modifies the array elements, the change persists... so what if the function actually damaged the contents of the array? now we lost the original values. A work around is to use the keyword '**const**' in declaring the function parameter so to allow passing by reference in 'read-only' mode.

## Passing 1D Array to Functions

49

### Sample functions :

- void PrintArray( int A[] );
- int CalculateSum( int A[] );
- int Sort( int A[], int size, int order );
- int Search( int A[], int size, int key );

### How to call it?

- PrintArray(MyArray);

## Passing N-D Array to Functions

50

- Unlike 1-D arrays, all array maximum dimensions must be declared explicitly (except for the leftmost dimension)

### Example – InputMarks( NS, NA, Mark )

- Prototype  
void InputMarks( int, int, float [ ][NUM\_ASSIGNS] );
- Function Definition  
void InputMarks( int NS, int NA, float marks[ ][NUM\_ASSIGNS] )  
{  
...  
}

## Example 2D Array

51

### Matrix Transpose - Transpose( A, AT, NR, NC )

- Interchange rows and columns
- Prototype:  
void Transpose( float [ ][MC], float [ ][MR], int, int );
- The result array must be defined to be of suitable size
  - If we define: float A [MR] [MC];  
then we must define: float AT [MC] [MR];
- Key statement
  - for( r=0; r<NR ; r++ )  
for( c=0; c<NC ; c++ )  
AT[ c ][ r ] = A[ r ][ c ] ;

## Array Based Algorithms

### Searching Array Elements

## Array Based Algorithms

53

- Searching
  - ▣ How to locate items in a list
  - ▣ Simplicity versus speed and list properties
- Sorting
  - ▣ Putting list elements in order by relative value
  - ▣ Promoting efficient search

## Search Algorithms

54

- Searching is a fundamentally important part of working with arrays
  - ▣ Example: Given a student ID number, what is the Mark they obtained on the test? Do this for all students who enquire.
- Constructing a good, efficient algorithm to perform the search is dependent on whether the IDs are in random order or sorted.
  - ▣ **Random** order – use sequential search → **Linear Search**
  - ▣ **Sorted** order – use divide-and-conquer approach → **Binary Search**

## Linear Search

55

- Search the array **A** for the first occurrence of the value **n**.
- Linear or sequential search is the simplest search technique – but not necessarily the best!
  1. Start at the first element **A[0]**
  2. Check if its contents equals to **n**
  3. if it is, the report its position
  4. else, check the element value and the next index.
- Note: Array contents need not be in any particular order. Linear search works well for unsorted arrays.
- **Best Case** Scenario: we find **n** at the first position. **O(1)**
- **Worst Case** Scenario: We search the whole array and don't find a match for **n**. Order Magnitude **O(n)**

## Linear Search Implementation

56

```
/* Find all occurrences of n */
#define MAX 5
int main(void)
{
 int A[] = { 5, 3, 2, 7, 1};
 int i, n = 2, found = 0;
 for (i = 0; i < MAX; i++){
 if (A[i] == n) {
 printf("\n found at position %d", i);
 found = 1; /* toggle flag */
 }
 }
 if (found == 0)
 printf ("n was NOT found");
 return 0;
}
```

## Linear Search Implementation

57

```
/* Find the number of occurrences of n in A */

#define MAX 10
int main(void)
{
 int A[] = { 5, 3, 2, 7, 1, 2, 8, 2, 0, 1};
 int i, n = 2, found = 0;
 for (i = 0; i < MAX; i++) {
 if (A[i] == n)
 found ++; // increment found counter
 }
 if (found == 0)
 printf ("n was NOT found");
 else
 printf("n was found %d time(s).", found);
 return 0;
}
```

## Search Algorithms - Binary

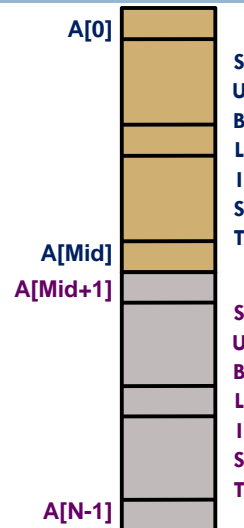
58

- Let us now consider an array **V** of N values where the elements A[K] are sorted in **ascending order** (from smallest to largest)
  - $V[0] < V[1] < \dots < V[N-2] < V[N-1]$
- Problem: Find if/where the search value **VS** is in **V**
- We develop the **Binary Search** algorithm
  - Our design technique is based on the principle of **Divide and Conquer**
  - Also called **Binary Subdivision**

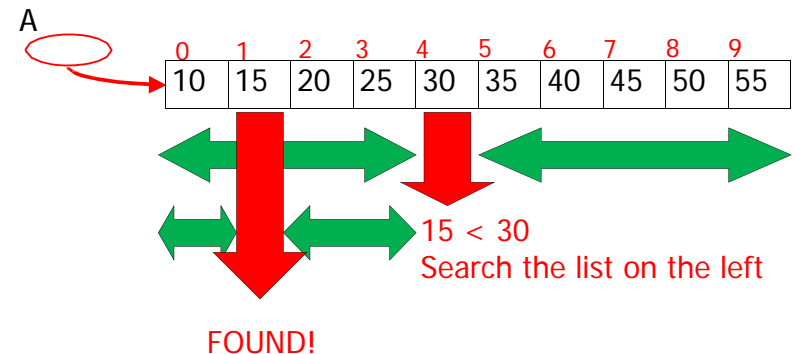
## Binary Search

59

- Our strategy involves the idea of sub-list. A **sub-list** is simply a smaller part of a list.
- By dividing a list into two sub-lists (where each sub-list contains contiguous values that are sorted) it is possible to quickly eliminate one of the sub-lists with a single comparison.
- Thereafter, we focus attention on the remaining sub-list – but, we reapply the same **divide and conquer** technique.



## Binary Search: Find 15



## Binary Search Implementation

61

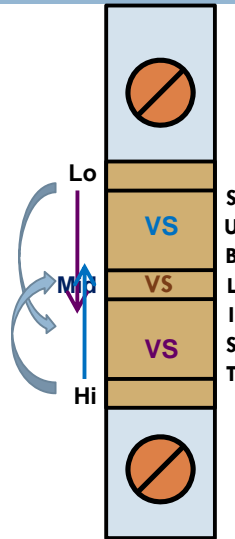
- Binary Search algorithm:

```

Lo = 0;
Hi = N-1 ; /* Use full list as sub-list */
do {
 Mid = (Lo + Hi) / 2 ; //int division
 if(VS == V[Mid]) break ;
 if(VS > V[Mid])
 Lo = Mid + 1 ;
 else
 Hi = Mid - 1 ;
} while (????) ;

printf("Search value %f ", VS) ;
if (VS == V[Mid])
 printf("found at position %d\n", Mid) ;
else
 printf("not found\n") ;

```



## Recursive Binary Search

62

```

int binarySearch(int A[], int searchKey, int low, int high)
{
 int middle;
 if(low > high)
 return -1;
 middle = (low + high) / 2;
 if(searchKey == A[middle])
 return middle;
 if (searchKey < A[middle])
 return binarySearch(A, searchKey, low, middle - 1);
 else
 return binarySearch(A, searchKey, middle + 1, high);
}

```

Best Case:  $O(1)$ , Worst Case:  $O(\log(N))$

## Binary Search – Complexity

63

- To understand the complexity assume a data set of 256 values. We consider the worst case scenario.

| Size of data set | Step # |
|------------------|--------|
| 256              | 1      |
| 128              | 2      |
| 64               | 3      |
| 32               | 4      |
| 16               | 5      |
| 8                | 6      |
| 4                | 7      |
| 2                | 8      |
| 1                | 9      |

$N = 256 = 2^8$ ,  
so it has taken  $8+1 = 9$   
steps to prove that VS  
does not exist in V.

In general, for a list of  
size  $N = 2^K$ , it takes  $K+1$   
steps, or  $O(\log_2 N)$

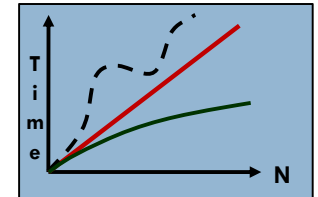
- Once we split the sub-list to size 1 we clearly determine that the list cannot contain the search value.

## Search Algorithms - Complexity

64

- The relative efficiencies, or complexities, of the various search algorithms is clearly established when  $N$  is large, and for the **worst case** scenarios.

- Random  $O(>N?)$
- Sequential (Linear)  $O(N)$
- Divide & Conquer (Binary)  $O(\log N)$



- In the **best case**, any search algorithm may be successful after only one (1) probe
- Usually, one is interested in worst case and average case in choosing an algorithm.



## Array Based Algorithms

### Sorting Array Elements

## Sorting Algorithms

66

- Fast searching using Binary Search → need for the data to be ordered within lists
- Sorting algorithms are designed to perform the ordering required.
  - Selection Sort
  - Bubble Sort
  - Improved Bubble Sort
  - Quick Sort (Recursive)
  - Insertion Sort
  - Merge Sort
  - ...
- Some of these are iterative, while others are recursive.
- A number of sorting algorithms exhibit time complexities of  $O(N^2)$ , but some achieve better efficiencies (eg.  $O(N \log N)$ ) under certain circumstances.

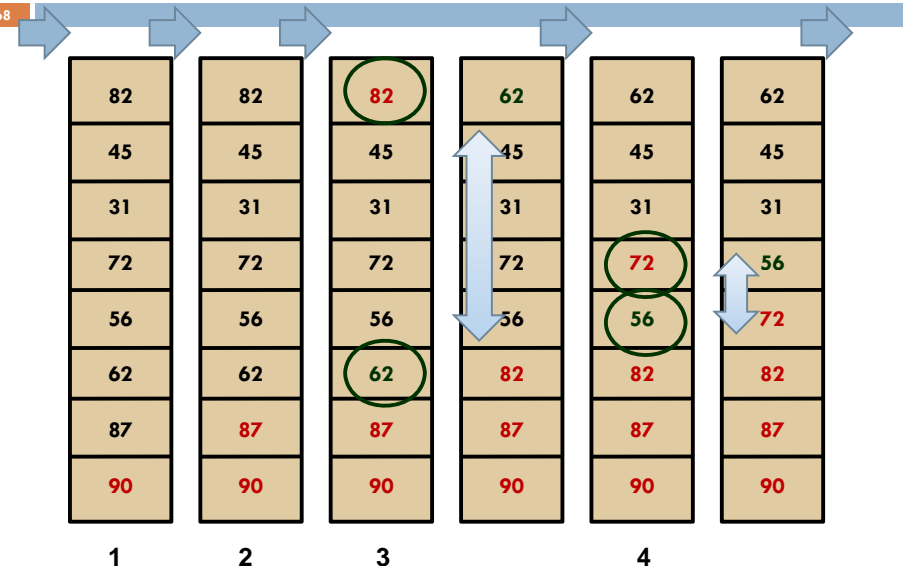
## Selection Sort

67

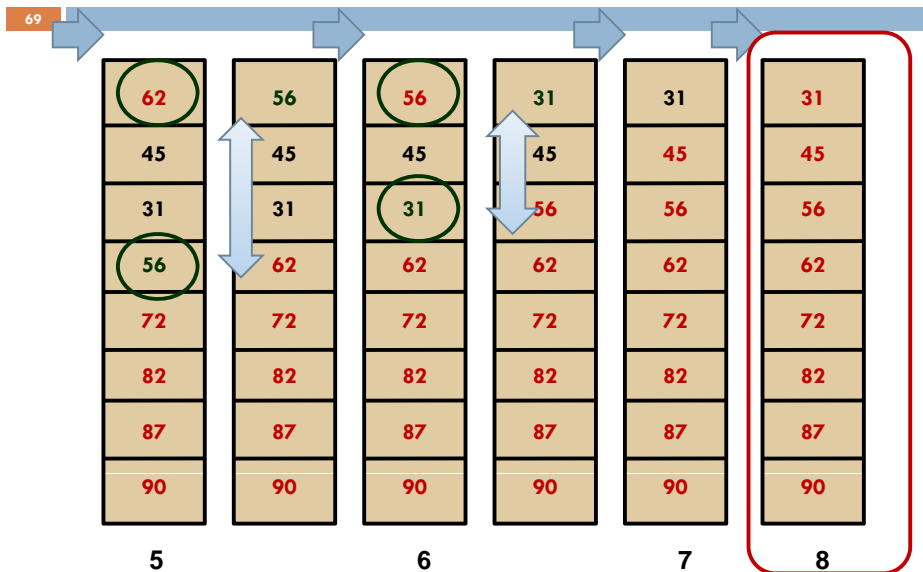
- **Selection Sort** relies on being able to find the largest (or smallest) element in a sublist
- Each time the proper value is found, it is exchanged with the element at the start/end of the sublist
- We re-apply this technique by shrinking the size of the sublist, until there is no remaining sublist (or a sublist of size 1 element – which is already sorted).
- We consider the example .....

## Selection Sort – Example

68



## Selection Sort – Example



## Selection Sort ...

- From the example we note that the final step 8 is not actually required, since a sub-list of one (1) element is automatically sorted (by definition).
- Hence, it took 7 steps to complete the sort. In general, for a list of size  $N$  elements, it takes  $N-1$  steps.
- Each step consists of two parts:
  - First, search an unordered sub-list for the largest element
    - The size of the sub-list is  $N-K$  for step  $K$  (starting from  $K=0$ )
  - Second, exchange (swap) the largest element with the last element in the sub-list
- Upon completion of each step it should be noted that the **sorted sub-list grows by one element** while the **unordered sub-list shrinks by one element**.

## Selection Sort ...

- Start with a sub-list of  $N$  unsorted values, and a sub-list of 0 sorted values.
  - The unsorted sub-list has subscripts in the sub-range  $[0 \dots N-1]$
  - The sorted sub-list has subscripts in the sub-range  $[N \dots N]$  which is not occupied physically (hence, it does not exist, it is the empty set)
- For  $K$  from 0 to  $N-1$ , in increments of 1, perform
  - Search the unordered sub-list  $[0 \dots N-1-K]$  for the largest value and store its position  $P$
  - Exchange the largest element with the last element in the sub-list using positions  $P$  and  $N-1-K$ .
  - The exchange adds the largest element to the beginning of the sorted sub-list, while removing it from the end of the unordered sub-list

## Sorting Algorithms - Selection

```

void SelectionSort (double List[], int N) {
 int J, K, P ; // store various index locations in the array
 double Temp ; // used for swapping
 for(J = 0 ; J < N-1 ; J++) {
 P = 0 ; //index of the largest number in the unordered sub-list
 for(K = 0 ; K < N - J ; K++)
 if(List[P] < List[K]) //find new maximum
 P = K ;

 /* swap the largest element with the last element in the sub-list */
 Temp = List[P] ;
 List[P] = List[N-1-J] ;
 List[N-1-J] = Temp ;
 }
}

```

## Example

73

- Sort the following array using selection sort algorithm

- $N = 10$

| K | P | N-K-1        | 0  | 1  | 2  | 3  | 4 | 5  | 6  | 7  | 8  | 9  |
|---|---|--------------|----|----|----|----|---|----|----|----|----|----|
| 0 | 0 | $10-0-1 = 9$ | 17 | 14 | 12 | 11 | 6 | 10 | 15 | 8  | 5  | 3  |
| 1 | 6 | $10-1-1 = 8$ | 3  | 14 | 12 | 11 | 6 | 10 | 15 | 8  | 5  | 17 |
| 2 | 1 | $10-2-1 = 7$ | 3  | 14 | 12 | 11 | 6 | 10 | 5  | 8  | 15 | 17 |
| 3 | 2 | $10-3-1 = 6$ | 3  | 8  | 12 | 11 | 6 | 10 | 5  | 14 | 15 | 17 |

## Example ...

74

| K | P | N-K-1        | 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
|---|---|--------------|---|---|---|----|----|----|----|----|----|----|
| 4 | 3 | $10-4-1 = 5$ | 3 | 8 | 5 | 11 | 6  | 10 | 12 | 14 | 15 | 17 |
| 5 | 3 | $10-5-1 = 4$ | 3 | 8 | 5 | 10 | 6  | 11 | 12 | 14 | 15 | 17 |
| 6 | 1 | $10-6-1 = 3$ | 3 | 8 | 5 | 6  | 10 | 11 | 12 | 14 | 15 | 17 |
| 7 | 1 | $10-7-1 = 2$ | 3 | 6 | 5 | 8  | 10 | 11 | 12 | 14 | 15 | 17 |
| 8 | 1 | $10-8-1 = 1$ | 3 | 5 | 6 | 8  | 10 | 11 | 12 | 14 | 15 | 17 |
|   |   |              | 3 | 5 | 6 | 8  | 10 | 11 | 12 | 14 | 15 | 17 |

## Selection Sort - Complexity

75

- How many operations must be performed?  
Sum from  $J = 0$  to  $N-2$       `// for( J = 0 ; J < N-1 ; J++ )`  
Sum from  $K = 0$  to  $N-1-J$    `// for( K = 0 ; K < N - J ; K++ )`  
Core loop logic (CoreOps)
- Gauss dealt with this problem and developed several formulae which bear his name.
- The maximum number of operations required to sort a list of  $N$  elements in ascending order is  
 $\frac{1}{2} N ( N - 1 )$  CoreOps
- The CoreOps consist of several fetches, one comparison and either 1 or 2 assignments (stores) → This is, essentially, a constant value
- Thus, the time complexity of the Selection Sort algorithm is  
 $O(N^2)$

## Question

76

- What are the correct intermediate steps of the following data set when it is being sorted with the Selection sort? 25,30,20,28

## Bubble sort

77

- The name "Bubble" sort is taken from the rising of the elements at the *bottom* of the list towards the *top* of the list, just like bubbles in a boiling pot.

```
void BubbleSort (double List[], int N) {
 int Pass, K;
 double temp;
 /* loop to control number of passes */
 for(Pass = 0 ; Pass < N-1 ; Pass++)
 /* loop to control number of comparisons per pass */
 for(K=0; K < N-1; K++)
 if(List[K] > List[K+1]) { //Swap the larger value
 /* swap */
 temp = A[K];
 A[K] = A[K+1];
 A[K+1] = temp;
 }
}
```

Best case = Worst case =  $O(N^2)$

## Bubble Sort - Example

78

pass=0

| K=0 | K=1 | K=2 |
|-----|-----|-----|
| 5   | 1   | 1   |
| 1   | 5   | 5   |
| 6   | 6   | 3   |
| 3   | 3   | 6   |

pass=1

| K=0 | K=1 | K=2 |
|-----|-----|-----|
| 1   | 1   | 1   |
| 5   | 5   | 3   |
| 3   | 3   | 5   |
| 6   | 6   | 6   |

pass=2

| K=0 | K=1 | K=2 |
|-----|-----|-----|
| 1   | 1   | 1   |
| 3   | 3   | 3   |
| 5   | 5   | 5   |
| 6   | 6   | 6   |

Done!

## Bubble sort ...

79

```
/* loop to control number of passes */
for(Pass = 0 ; Pass < N-1 ; Pass++)
 /* loop to control number of comparisons per pass */
 for(K=0; K < N-Pass-1; K++)
 if(List[K] > List[K+1]) {
 temp = A[K];
 A[K] = A[K+1];
 A[K+1] = temp;
 }
```

pass=0

| K=0 | K=1 | K=2 |
|-----|-----|-----|
| 5   | 1   | 1   |
| 1   | 5   | 5   |
| 6   | 6   | 3   |
| 3   | 3   | 6   |

pass=1

| K=0 | K=1 |
|-----|-----|
| 1   | 1   |
| 5   | 3   |
| 3   | 5   |
| 6   | 6   |

pass=2

| K=0 |
|-----|
| 1   |
| 3   |
| 5   |
| 6   |

Done!



## Improved Bubble Sort

80

```
void ImprovedBubbleSort(int A[], int N)
{
 int pass, K, done = 0; // used done as a flag to indicate if no swaps has occurred
 int temp; // used for swapping

 // testing each element, as long as we are not done
 for (pass= 0; pass < N-1 && !done; pass++) {
 /* assume we are done - no swapping would occur in this next pass */
 done = 1;
 for (K = 0; K < N-Pass-1; K++) {
 if (A[K] > A[K+1]) {
 temp = A[K];
 A[K] = A[K+1];
 A[K+1] = temp;
 /* since a swap occurred, then we are not done */
 done = 0;
 }
 }
 }
}
```

Best case =  $O(N)$  , Worst case =  $O(N^2)$

## Question

81

- What are the correct intermediate steps of the following data set when it is being sorted with the bubble sort? 25,30,20,28

## Performance Issues

82

- As array sizes grow, it becomes imperative that we carefully examine the performance of our techniques.
- The worst case scenario is often the one of interest.
- Many other search and sort algorithms exist. You will encounter them in other lab and assignment exercises and future courses

## Lecture 3: Summary

83

- Array Concepts & Mechanisms
- Multi-Dimensional Arrays
- Algorithms
  - ▣ Searching
  - ▣ Sorting
- Assigned Reading:
  - ▣ Chapter 6 – One- and multi-dimensional arrays
  - ▣ Chapter 7 – Pointers, call-by-value and call-by-reference
- Assignment
  - ▣ Deadline of first assignment is Jan. 29.
  - ▣ Second lab assignment has been posted on Blackboard.