CS2310 Computer Programming

LT06: Array

Computer Science, City University of Hong Kong

Semester A 2023-24

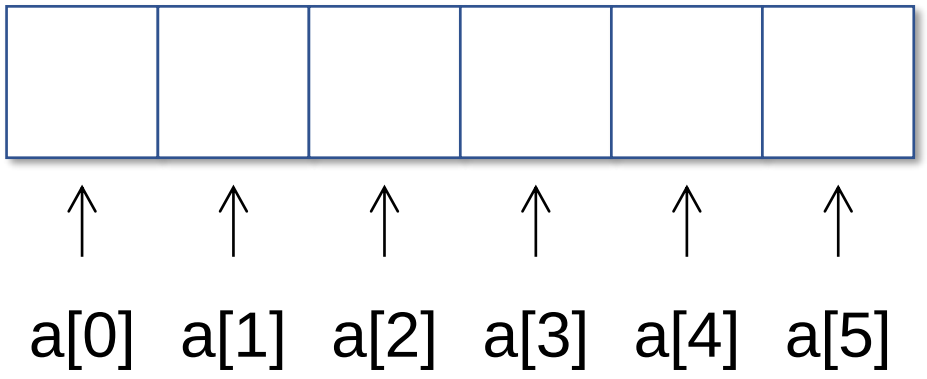
What's an Array?

- Sequence of data items of the same type

```
data_type array_name[size]
```

  - stored continuously
  - can be accessed by index, or subscript

```
int a[6];
```



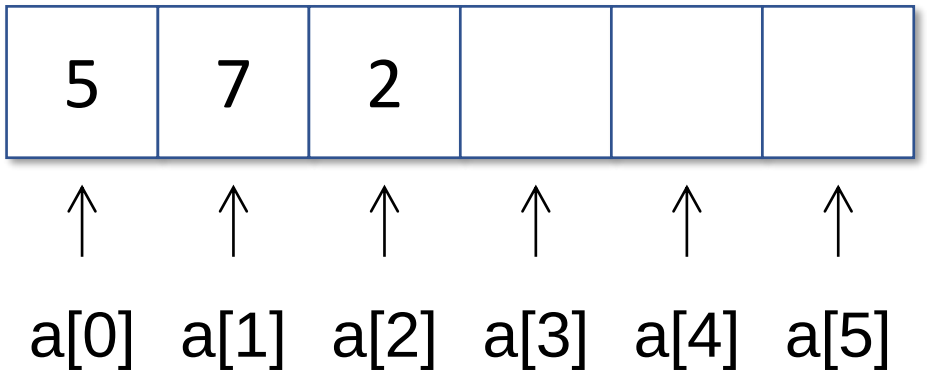
What's an Array?

- Sequence of data items of the same type

```
data_type array_name[size]
```

  - stored continuously
  - can be accessed by index, or subscript

```
int a[6];
a[0] = 5;
a[1] = 7;
a[2] = 2;
```



Today's Outline

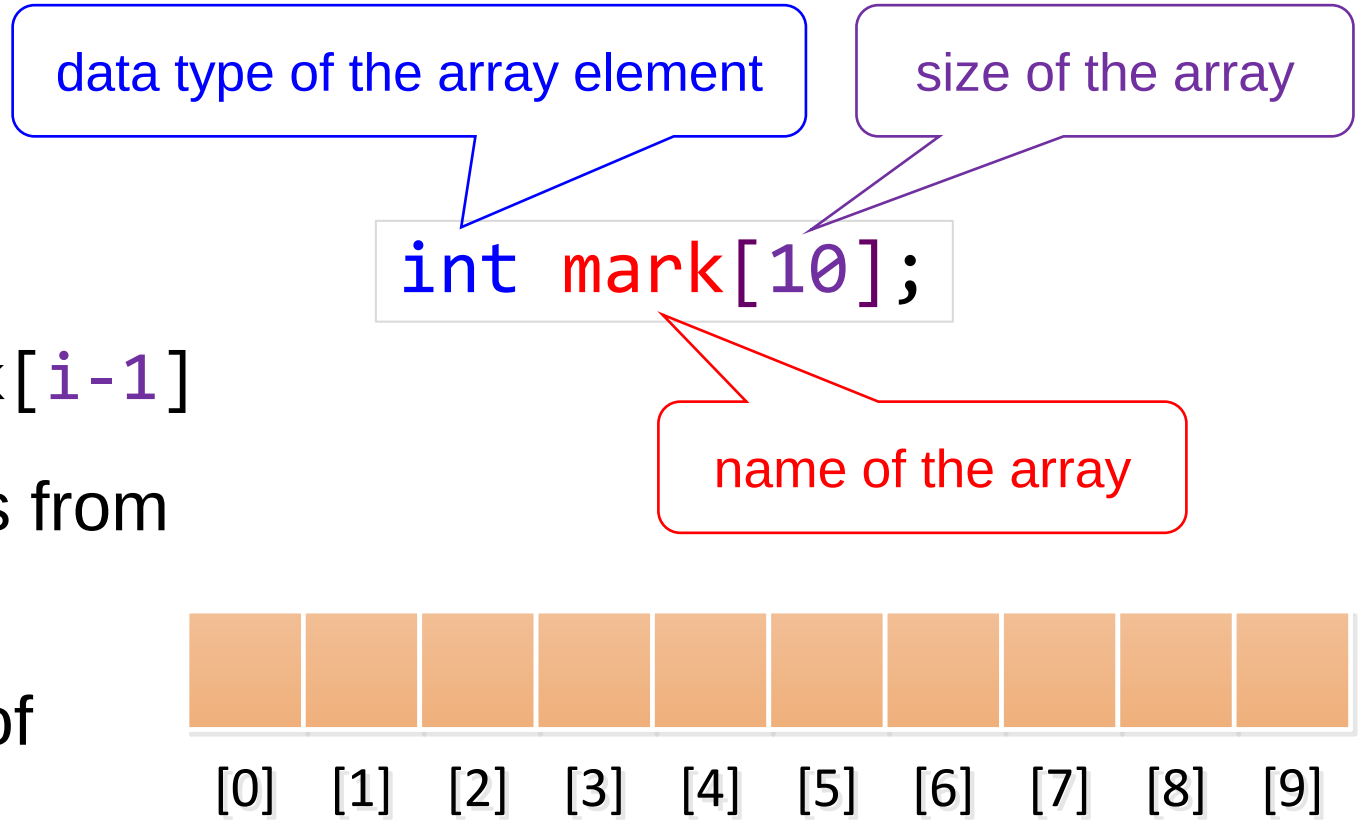
- Array definition
- Array initialization
- Passing array to functions
- Array operations
- Multi-dimensional array

Array Definition

- There're ten elements in this array

`mark[0], mark[1], …, mark[9]`

- the `i-th` array element is `mark[i-1]`
- the range of subscript `i` ranges from `0` to `array_size-1`
- `mark[10]` is invalid: array out of bound!



Array Definition

- Set array size

```
const int n = 0;
int mark[n];

int mark[50*50];

int n = 0;              // supported by some compiler versions
int mark[n];

int n; cin >> n;        // supported by some compiler versions
int mark[n];
```

Using #define to Set Array Size

- #define is a C++ predefined macro keyword

  Usage: `#define A B`

- Replace all occurrences of A to B

- Examples:

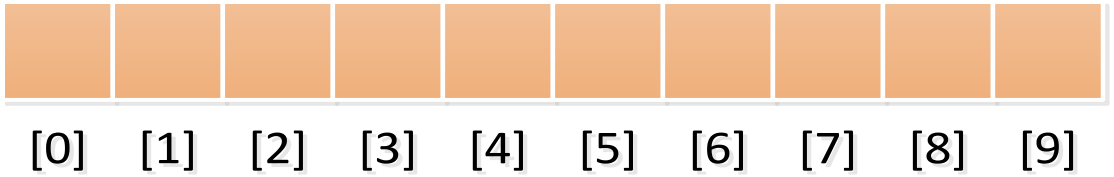
```
#define N 100
#define SIZE 10
```

- Using #define to set array size

```
#define N 10
int mark[N];
```

Assign Values to Array Elements

```
int mark[10]
```



- Suppose the mark of the first student is 30:

```
mark[0] = 30;
```

- Input the marks of the i-th student

```
cin >> mark[i];
```

- Input the marks for all 10 students

```
for (int i=0; i< 10; i++)
     cin >> mark[i];

for (int i=1; i<=10; i++)
     cin >> mark[i-1];
```

Retrieve Values of Array Elements

- Print the mark of the i-th student

```
cout << mark[i];
```

- Print the sum of the marks of all students

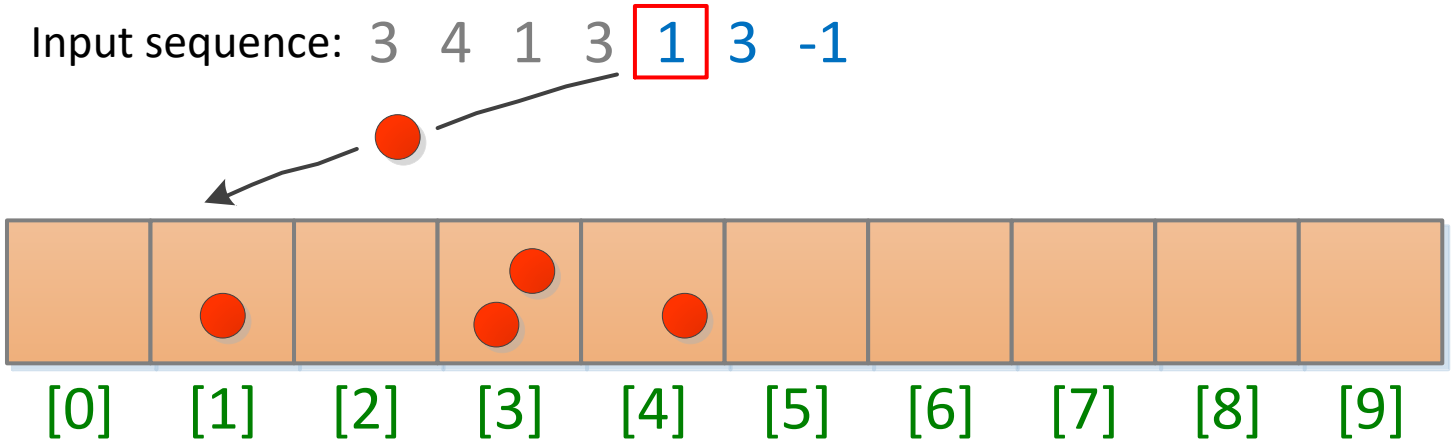
```
for (int i = 0; i < 10; i++) {
     cout << mark[i];
     sum += mark[i];
}
```

Example 1: Read and Write Array

```
#define N 10
int main() {
    int marks[N], sum=0;
    for (int i=0; i<N; i++)
        cin >> mark[i];
    cout << "The mark of the students are: ";
    for (int i=0; i<N; i++) {
        cout << mark[i];
        sum = sum + mark[i];
    }
    float average = (float)sum/N;
    cout << "Average mark=" << average << endl;
}
```

Example 2: Counting Digits

- Input a sequence of digits `{0, 1, 2, …, 9}`, which is terminated by `-1`
- Count the occurrence of each digit
- Use an integer array `count` of 10 elements
  - `count[i]` stores the number of occurrence of digit `i`



Example 2: Buggy Version

```
int count[10]; // number of occurrence of digits
int digit; // input digit

do { // read the digits
    cin >> digit;
    if (digit>=0 && digit<=9) // necessary to avoid out-of-bound
        count[digit]++;
} while (digit!=-1); //stop if the input number is -1

for (int i=0; i<10; i++) { // print the occurrences
    cout << "Frequency of " << i << " is " << count[i] << endl;
}
```

Example 2: The Actual Output

```
3 4 1 3 1 3 -1
Frequency of 0 is 2089878893
Frequency of 1 is 2088886165
Frequency of 2 is 1376256
Frequency of 3 is 3
Frequency of 4 is 1394145
Frequency of 5 is 1245072
Frequency of 6 is 4203110
Frequency of 7 is 1394144
Frequency of 8 is 0
Frequency of 9 is 1310720
```

Example 2: Correct Solution

```
int count[10]; // number of occurrence of digits
int digit; // input digit
for (int i=0; i<10; i++)
    count[i] = 0;
do { // read the digits
    cin >> digit;
    if (digit>=0 && digit<=9) // necessary to avoid out-of-bound
        count[digit]++;
} while (digit != -1); //stop if the input number is -1
for (int i=0; i<10; i++) { // print the occurrences
    cout << "Frequency of " << i << " is " << count[i] << endl;
}
```

```
// It's a good practiced to initialize arrays. Otherwise, array element
// values  will be unpredictable
```

Array Initialization

```
int a[3]={1,2,3};        // a has type int[4] and holds 1, 2, 3
```

Array Initialization

```
int a[3]={1,2,3};       // a has type int[3] and holds 1, 2, 3
int b[5]={1,2,3};       // b has type int[5] and holds 1, 2, 3, 0, 0
int c[4]={1};           // c has type int[4] and holds 1, 0, 0, 0
```

Array Initialization

```
int a[3]={1,2,3};       // a has type int[3] and holds 1, 2, 3
int b[5]={1,2,3};       // b has type int[5] and holds 1, 2, 3, 0, 0
int c[4]={1};           // c has type int[4] and holds 1, 0, 0, 0
int d[3]={0};           // d has type int[3] and holds all zeros
```

Count Digits: Correct Solution II

```
int count[10] = {0}; // number of occurrence of digits
int digit; // input digit

do { // read the digits
    cin >> digit;
    if (digit>=0 && digit<=9) // necessary to avoid out-of-bound
        count[digit]++;
} while (digit!=-1); //stop if the input number is -1


for (int i=0; i<10; i++) { // print the occurrences
    cout << "Frequency of " << i << " is " << count[i] << endl;
}
```

Array Initialization

```
int a[3]={1,2,3};       // a has type int[4] and holds 1, 2, 3
int b[5]={1,2,3};       // b has type int[5] and holds 1, 2, 3, 0, 0
int c[4]={1};           // c has type int[4] and holds 1, 0, 0, 0
int d[3]={0};           // d has type int[3] and holds all zeros
int e[2]={1,2,3};       // it's an error to provide more elements than
                        // array size
```

Array Initialization

```
int a[3]={1,2,3};       // a has type int[4] and holds 1, 2, 3
int b[5]={1,2,3};       // b has type int[5] and holds 1, 2, 3, 0, 0
int c[4]={1};           // c has type int[4] and holds 1, 0, 0, 0
int d[3]={0};           // d has type int[3] and holds all zeros
int e[2]={1,2,3};       // it's an error to provide more elements than
                        // array size
int f[] ={1,2,3};       // f has type int[3] and holds 1, 2, 3
int g[];                // it's an error to declare an array without specifying
                        // the size
```

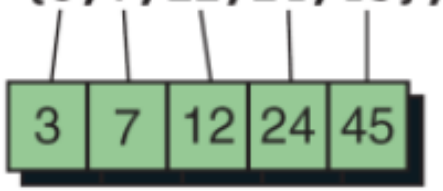
Array Initialization Summary

(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```

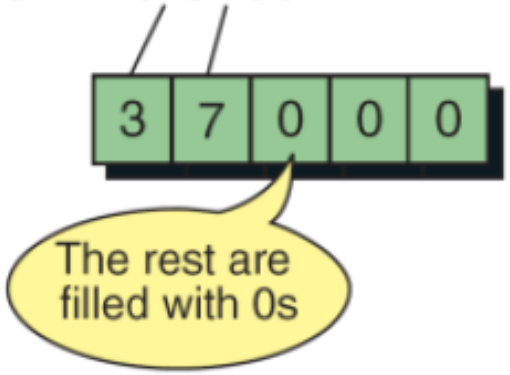
(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```



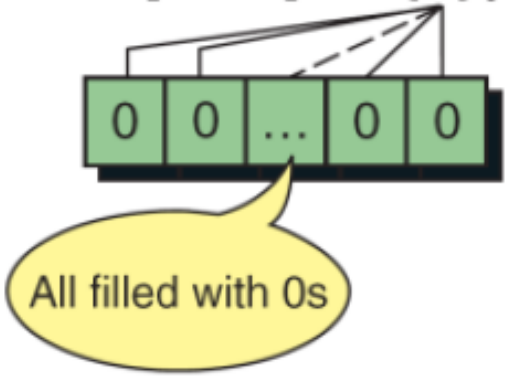
(c) Partial Initialization

```
int numbers[5] = {3,7};
```



(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```



Initialization of Char Arrays

- `char str[3]={'a','b','c'};`

- `char str[3]="abc";` // str has type char[3] and holds 'a', 'b', 'c'

- `char str[] ="abc";` // str has type char[4] and holds 'a', 'b', 'c', '\0'

  // More details in the string lecture

Passing Arrays to Function

- To pass an array to a function, we only need to specify the array name
- Array is passed by pointer

the size of the array is optional, e.g., you can write `void f(int a[])`

```
void func(int a[3]){
  cout << a[0] << endl; // print 1
  a[0]=10;
}

void main () {
  int a[3]={1,2,5};
  func(a);
  cout << a[0] << endl; // print 10
}
```

if the content of a[i] is modified in func, the modification will persist even after the function returns (Call by pointer, more in later lectures)

only need to input the array name!

Passing Arrays to Function

- The following program is invalid

```
void f(int x[20]) {
    …
}
void main() {
    int y[20];
    f(y[0]); // invalid, type mismatch
}
```

Today's Outline

- Array definition
- Array initialization
- Passing array to functions
- Array operations
  - sizeof
  - Compare two arrays
  - Sort
  - Search
- Multi-dimensional array

sizeof

- Recall: sizeof(data_byte) gives the number of bytes of the data_type

```
cout << sizeof(int); // will print 4
```

- sizeof Array gives the total number of bytes occupied by that array

```
int a[4];
cout << sizeof(a); // will print 16
```

sizeof

- Recall: sizeof(data_byte) gives the number of bytes of the data_type

```
cout << sizeof(int); // will print 4
```

- sizeof Array gives the total number of bytes occupied by that array

```
int a[4];
cout << sizeof(a); // will print 16
```

- How to calculate number of elements of an array?

Compare Two Arrays

- We have two integer arrays, each with 5 elements

```
int array1[5] = {10, 5, 3, 5, 1};
int array2[5]; // will be entered by the user
```

- Compare whether array1 and array2 are equal
  - array equality: two arrays are equal if all of their elements are equal.
  - you have to compare all array elements one by one
  - the following code will generate wrong result

```
if (array1 == array2)
    cout << "the arrays are equal\n";
```

Compare Two Arrays

```
void main() {
  int array1[5] = {10, 5, 3, 5, 1};
  int array2[5];
  cout << "input 5 elements to array2\n";
  for (int i=0; i<5; i++)
    cin >> array2[i];
  bool arrayEqual = true;
  for (int i=0; i<5 && arrayEqual; i++) {
    if (array1[i] != array2[i])
      arrayEqual = false;
  }
  if (arrayEqual)
    cout << "The arrays are equal\n";
  else
    cout << "The arrays are not equal\n";
}
```

Sorting

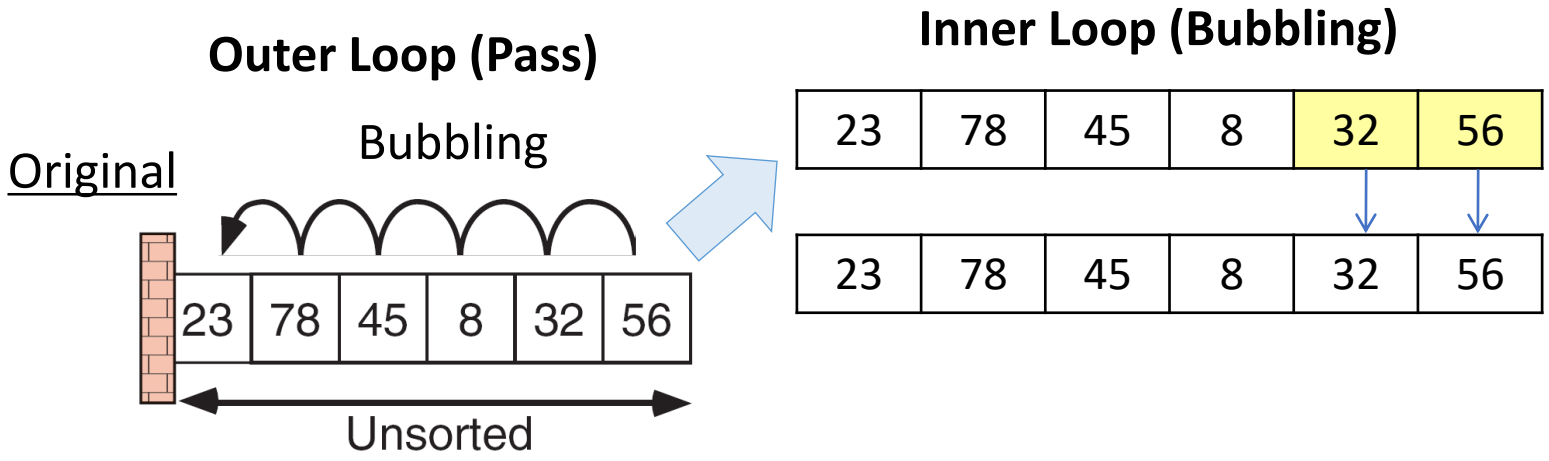
- One of the most common application in CS
  - arranging data by their values: {1, 5, 3, 2} -> {1, 2, 3, 5}
- Many algorithms for sorting

Selection sort Bubble sort Insertion sort	Slow but easy to code	Quick sort Merge sort Heap sort	Faster but more complex to code

- Based on iteratively swapping two elements in the array so that eventually the array is ordered
  - sorting algorithms differ in how they choose the two elements

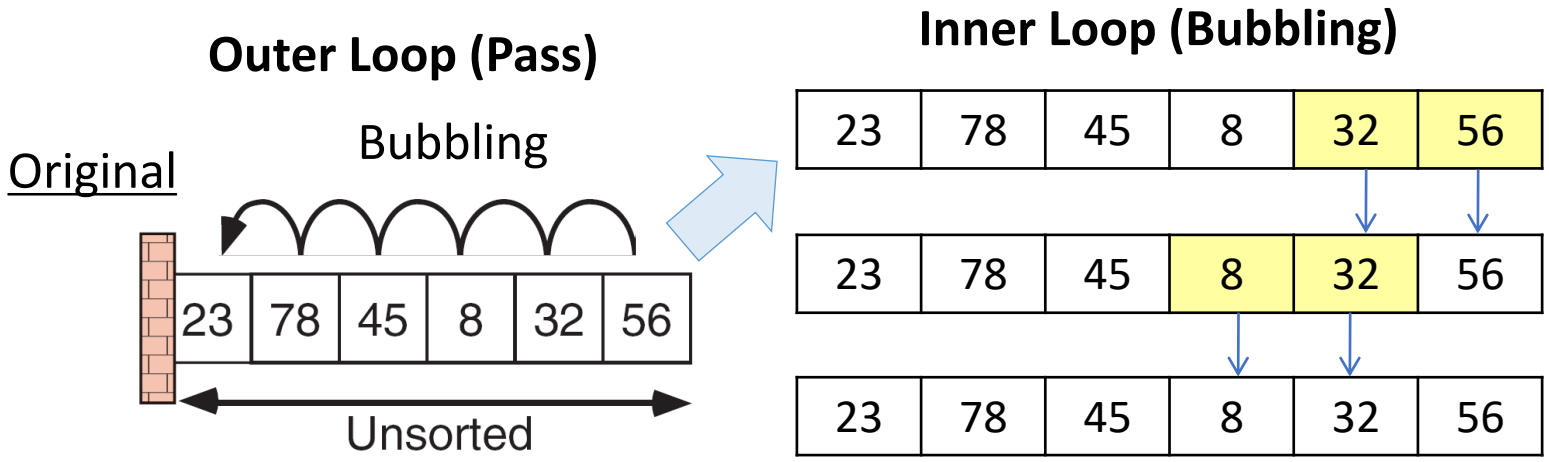
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



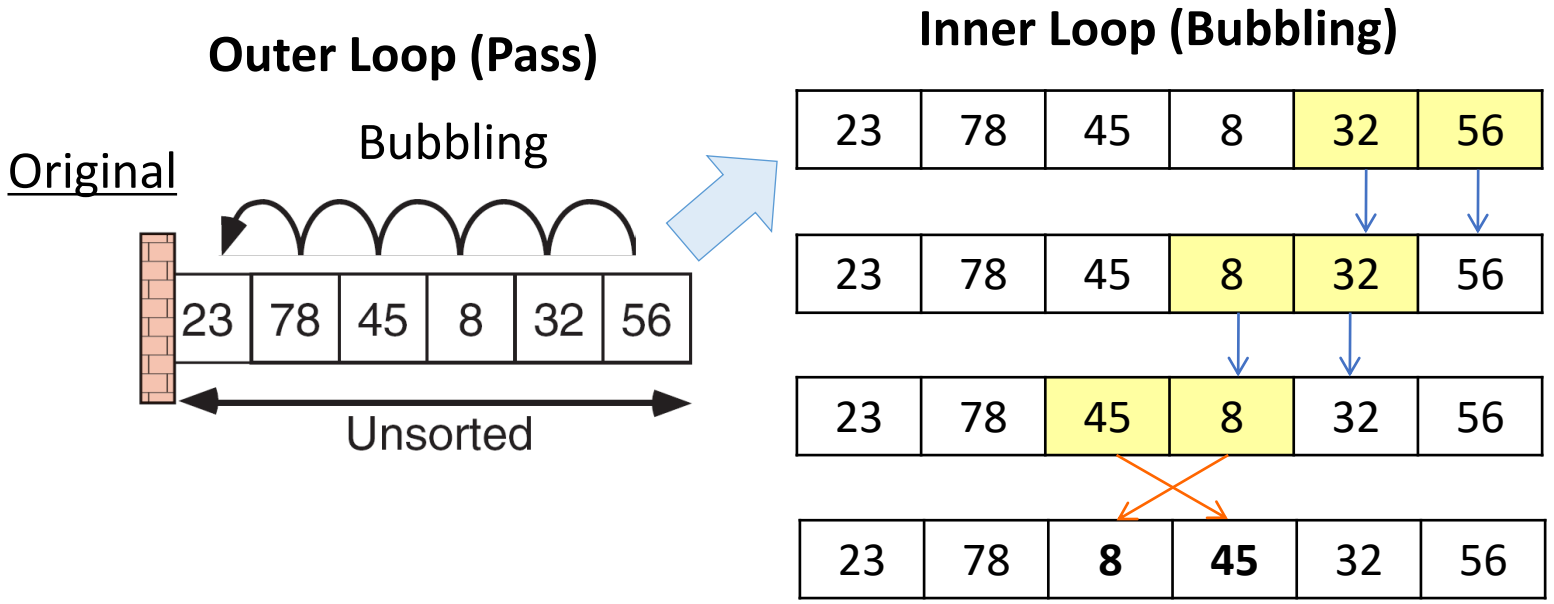
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



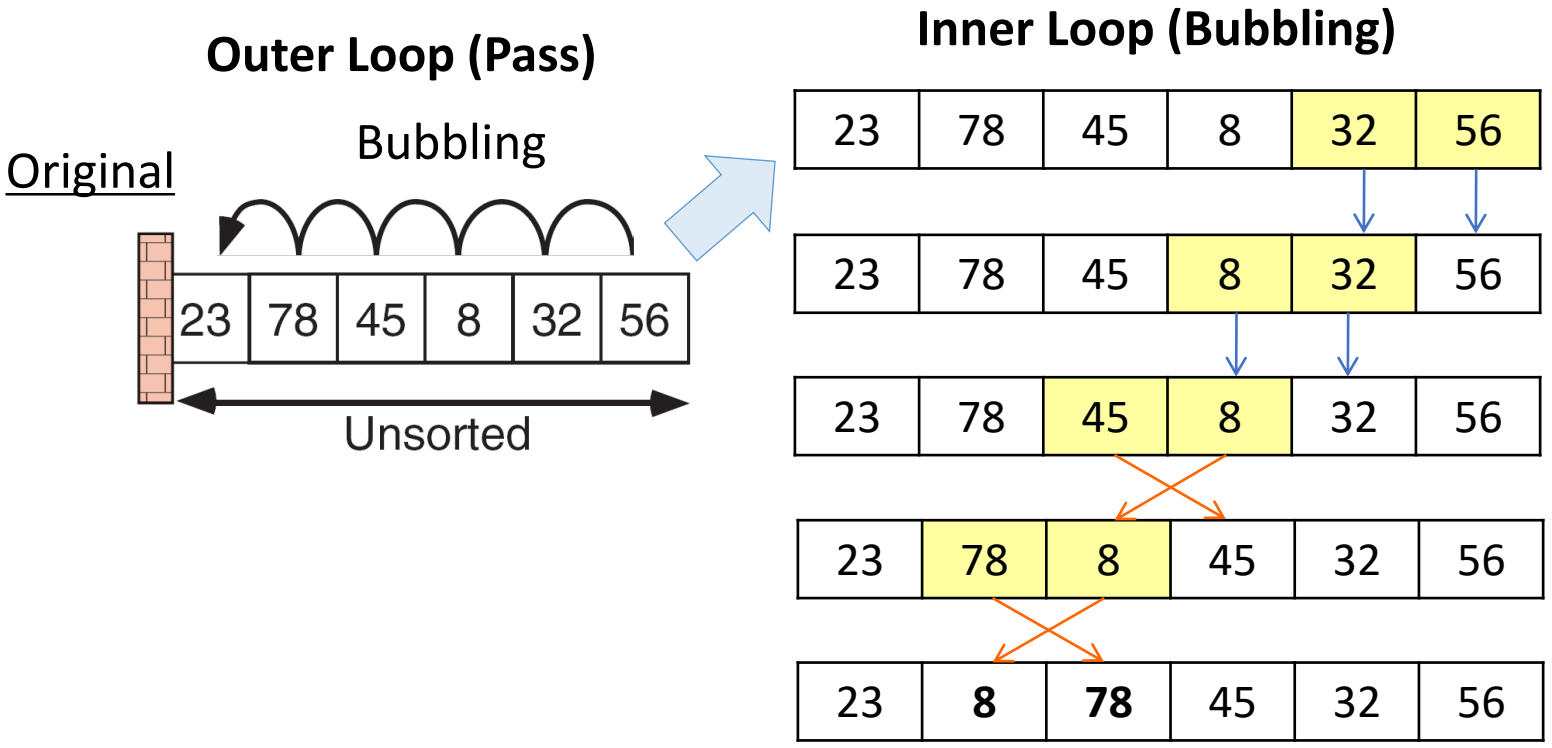
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



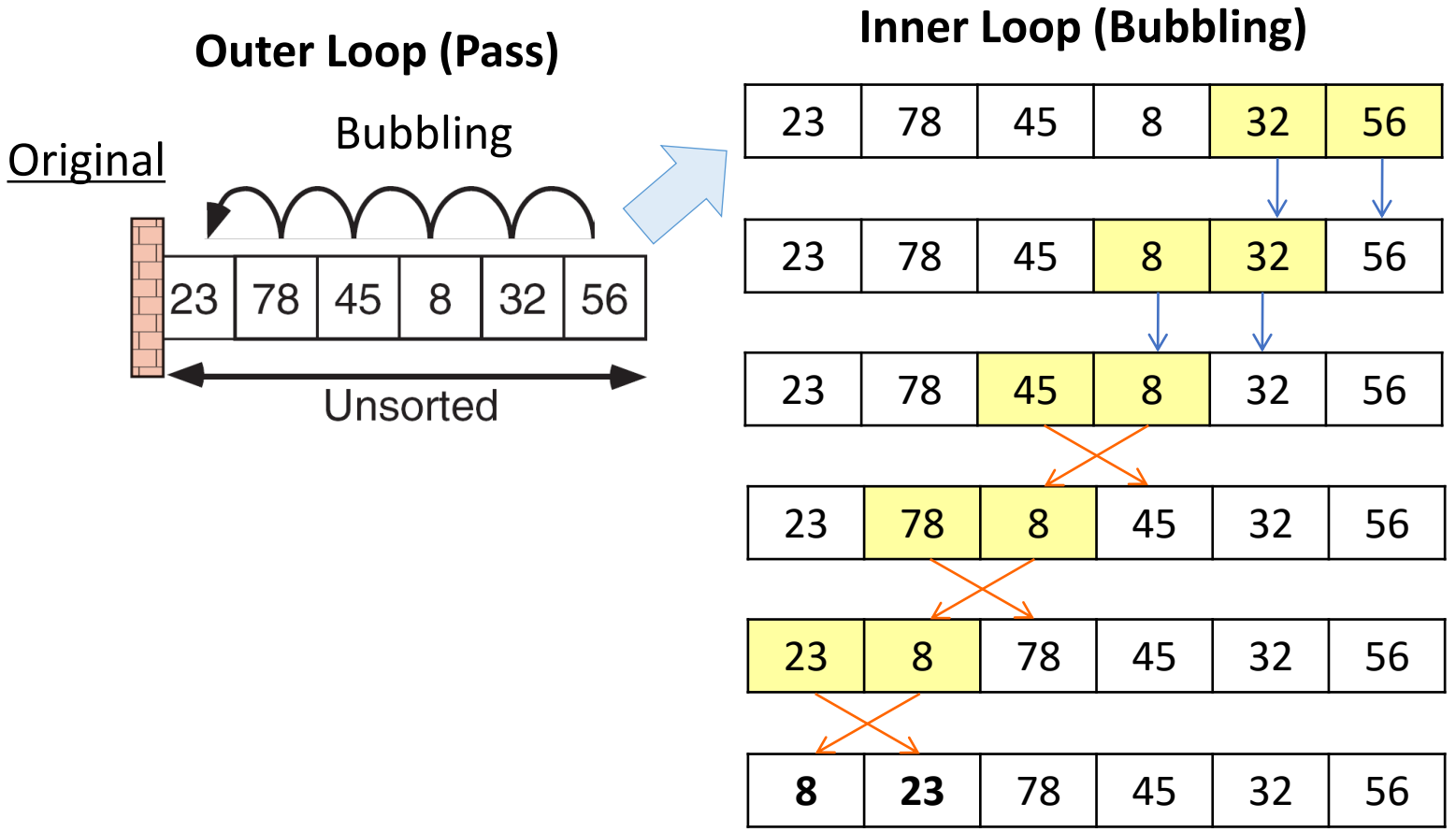
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



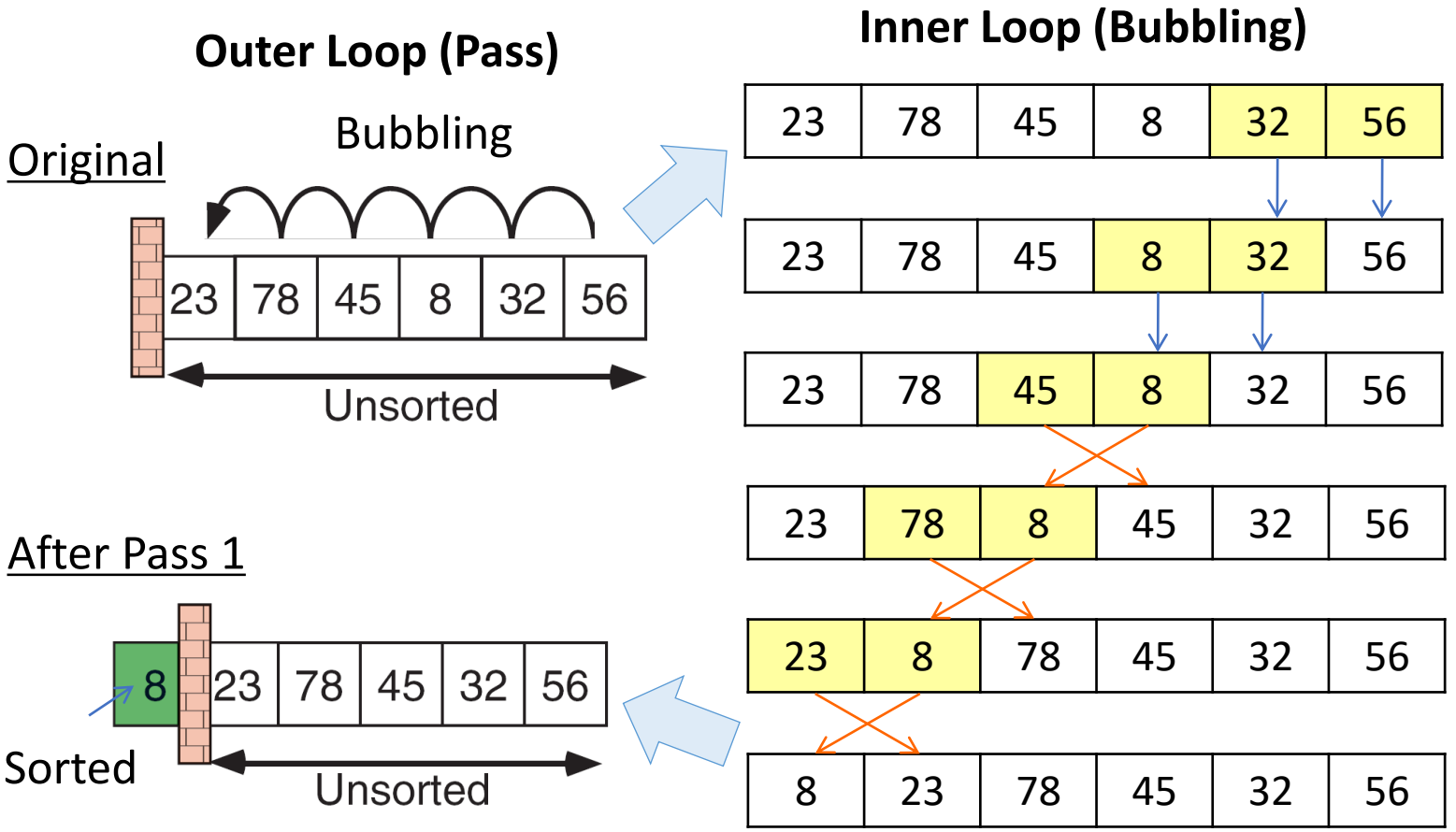
Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence



Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence
- After the pass, the smallest element is moved ("bubbled up") to the front of the array



Bubble Sort

- In each pass, start at the end, and swap neighboring elements if they are out of sequence
- After the pass, the smallest element is moved ("bubbled up") to the front of the array

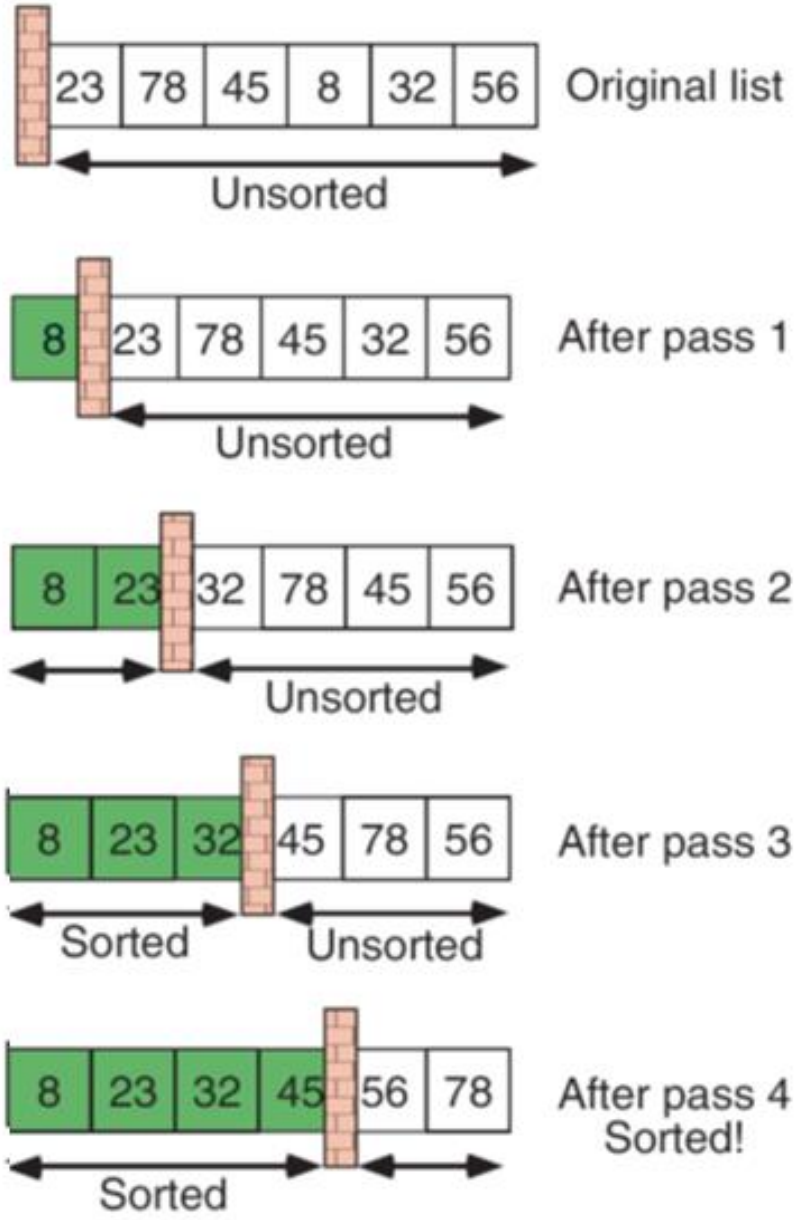
```
const int n = 6;
int a[n] = {23, 78, 45, 8, 32, 56};
int k, tmp;

for (k=n-1; k>0; k--) { // bubbling
   if (a[k]<a[k-1]) {
      tmp    = a[k];    // swap
      a[k]   = a[k-1];
      a[k-1] = tmp;
   }
}
```

Bubble Sort

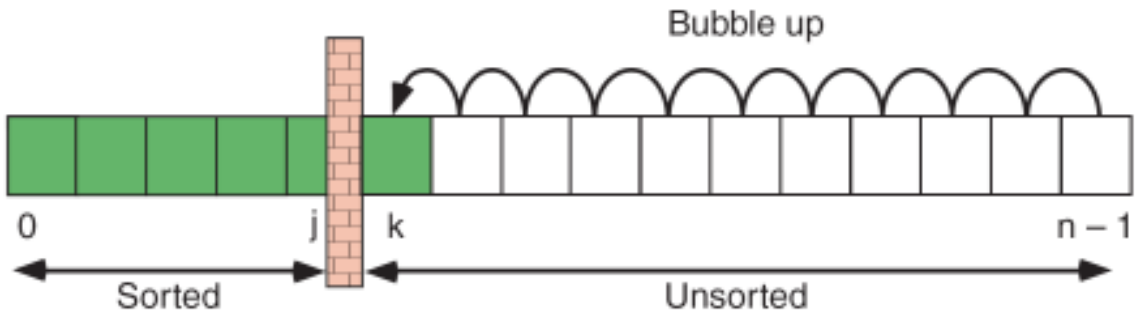
- Repeat bubbling up for *n* rounds, where *n* is array size
- After round j (start from round 0), the array is divided into two parts:
- sorted (green): from a[0] to a[j]
- unsorted: from a[j+1] to a[n-1]

bubble-sort dance: http://youtu.be/lyZQPjUT5B4
insert-sort dance: http://youtu.be/ROalU379l3U



Bubble Sort

- Repeat bubbling up for *n* rounds, where *n* is array size
- After round j (start from round 0), the array is divided into two parts:
- sorted (green): from a[0] to a[j]
- unsorted: from a[j+1] to a[n-1]




```
const int n = 6;
int a[n] = {23, 78, 45, 8, 32, 56};
int j, k, tmp;

for (j=0; j<n-1; j++) {     // outer loop
   for (k=n-1; k>j; k--) { // bubbling
      if (a[k]<a[k-1]) {
         tmp    = a[k];    // swap
         a[k]   = a[k-1];
         a[k-1] = tmp;
      }
   }
}

cout << "sorted: ";
for (j=0; j<n; j++)
   cout << a[j] << ' ';
cout << "\n";
```

Bubble Sort

- Repeat bubbling up for *n* rounds, where *n* is array size
- After round j (start from round 0), the array is divided into two parts:
- sorted (green): from a[0] to a[j]
- unsorted: from a[j+1] to a[n-1]

- Early stop: stop when the array is already sorted, no need to go through all n-1 passes

```
const int n = 6;
int a[n] = {23, 78, 45, 8, 32, 56};
int j, k, tmp;
bool sorted = false;
for (j=0; j<n-1 && !sorted; j++) {
   sorted = true; // optimistically assume sorted
   for (k=n-1; k>j; k--) { // bubbling
      if (a[k]<a[k-1]) {
         tmp    = a[k];    // swap
         a[k]   = a[k-1];
         a[k-1] = tmp;
         sorted = false;
      }
   }
}
cout << "sorted: ";
for (j=0; j<n; j++)
   cout << a[j] << ' ';
cout << "\n";
```

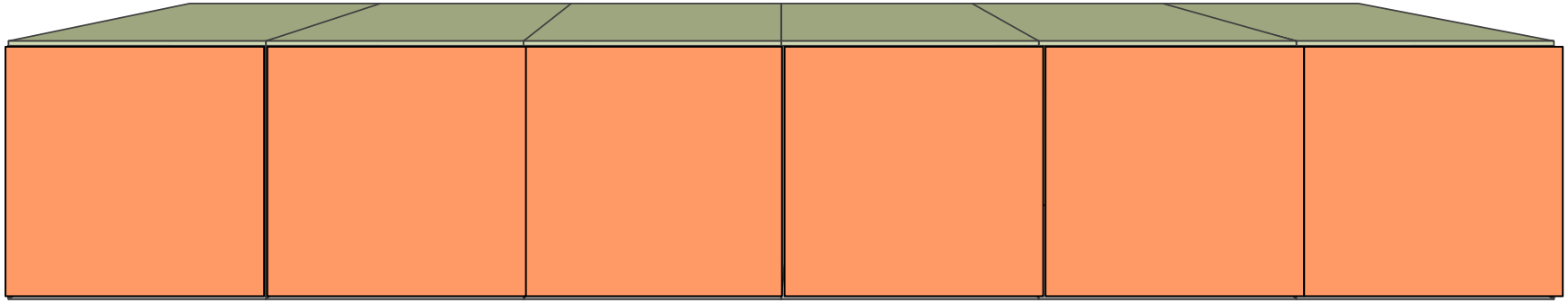
Searching

- Search: check if an element is in an array

- Example:
  - read 10 numbers from the user and store them in an array
  - user input another number x
  - write a program to check if x is an array element of the array
  - if yes, output the index of the element
  - if no, output -1

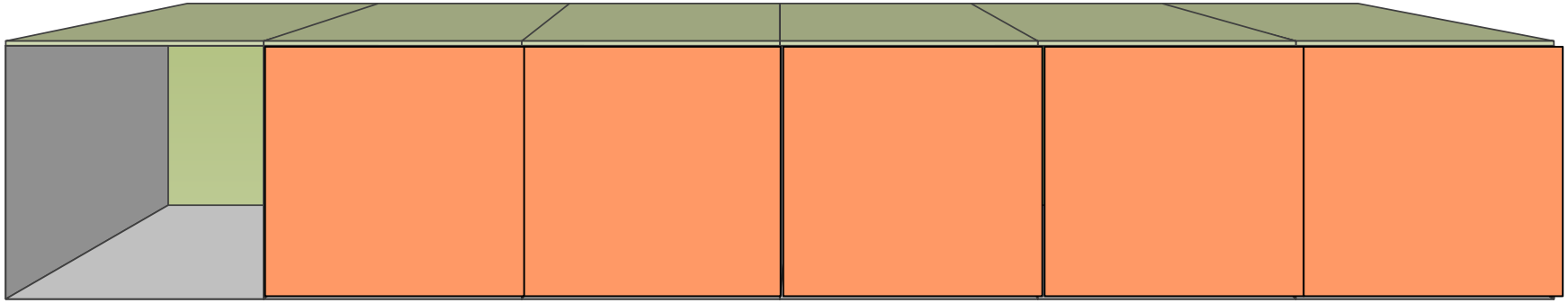
Searching for the Rabbit (case 1)

- Search sequentially for rabbit



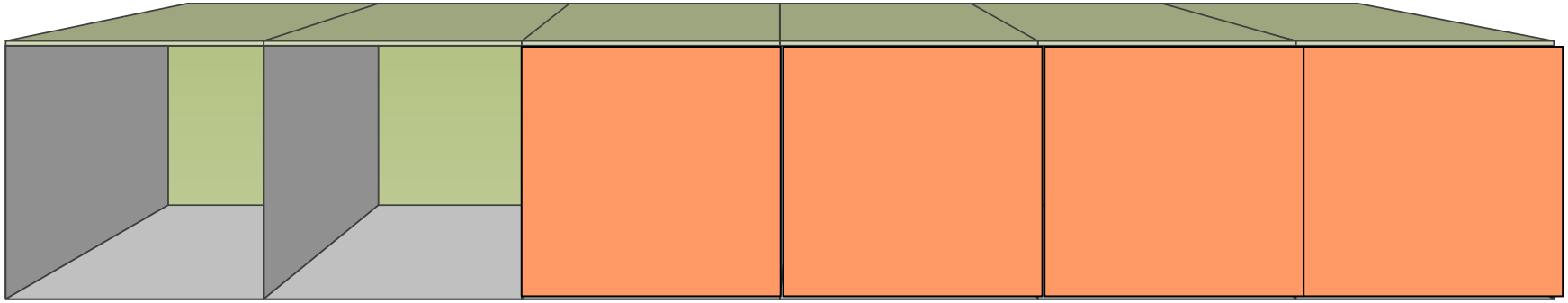
Searching for the Rabbit (case 1)

- Search sequentially for rabbit



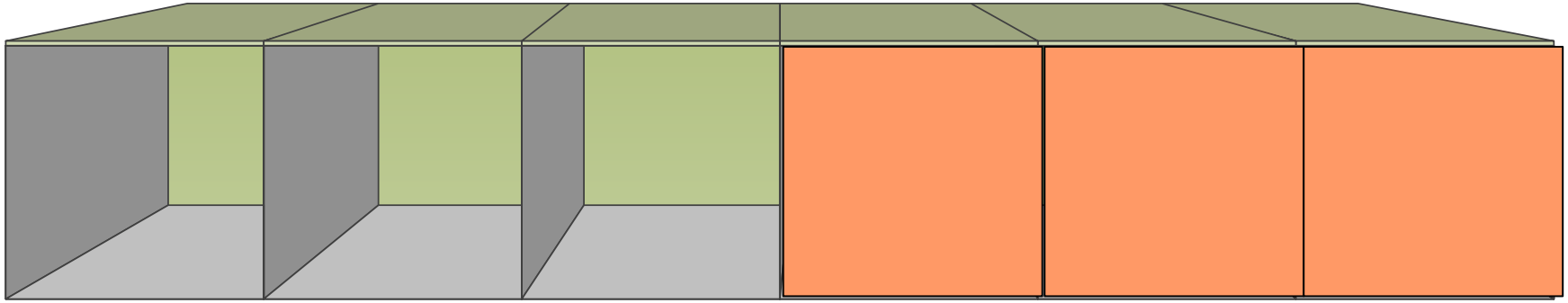
Searching for the Rabbit (case 1)

- Search sequentially for rabbit



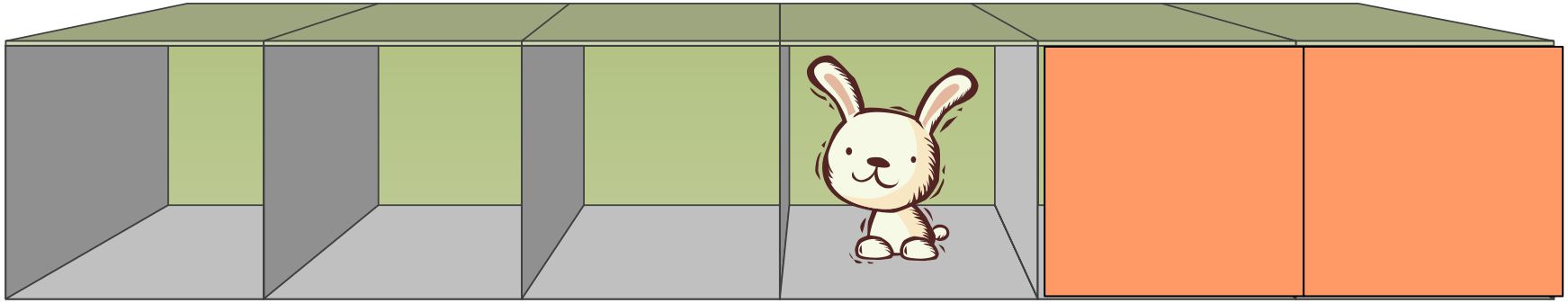
Searching for the Rabbit (case 1)

- Search sequentially for rabbit



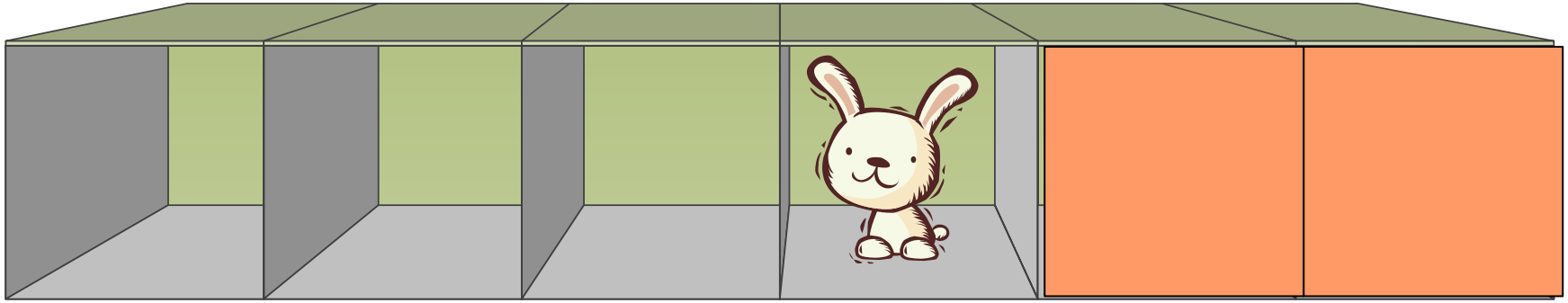
Searching for the Rabbit (case 1)

- Search sequentially for rabbit



Searching for the Rabbit (case 1)

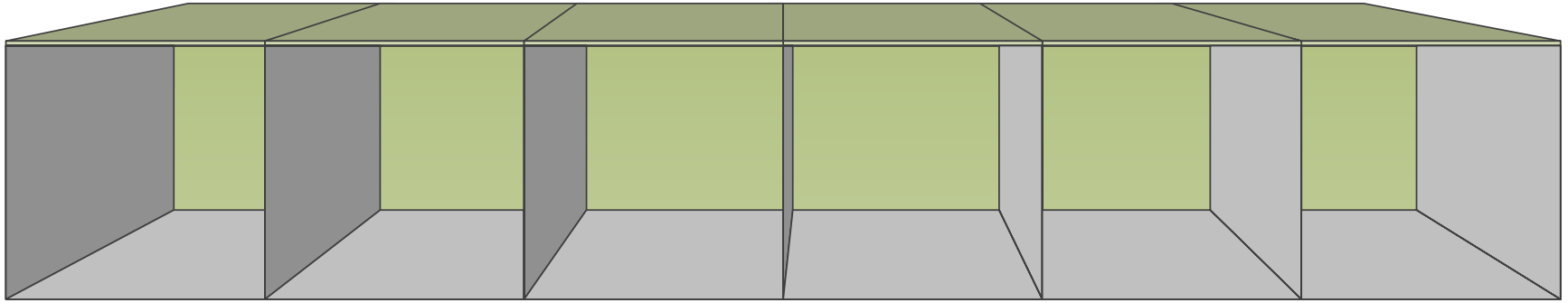
- Search sequentially for rabbit



If found, skip the rest

Searching for the Rabbit (case 2)

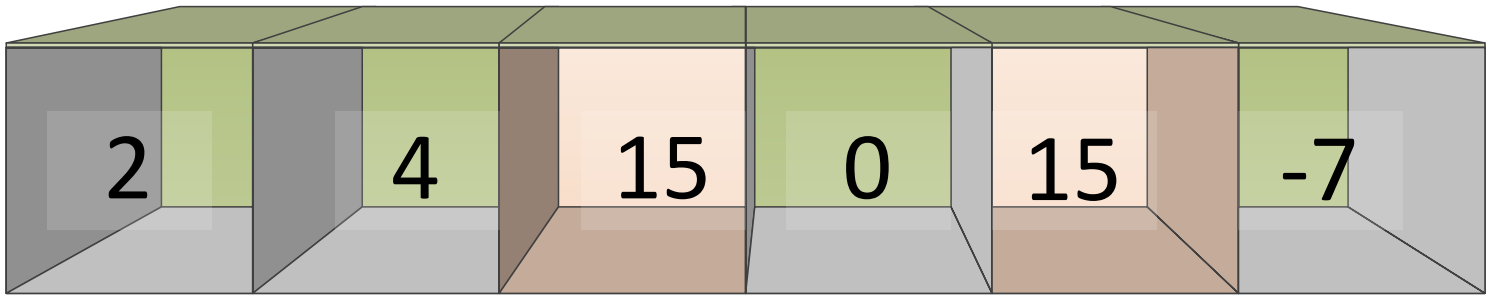
- Search sequentially for rabbit



No rabbit found, return -1

Searching for Element (case 1)

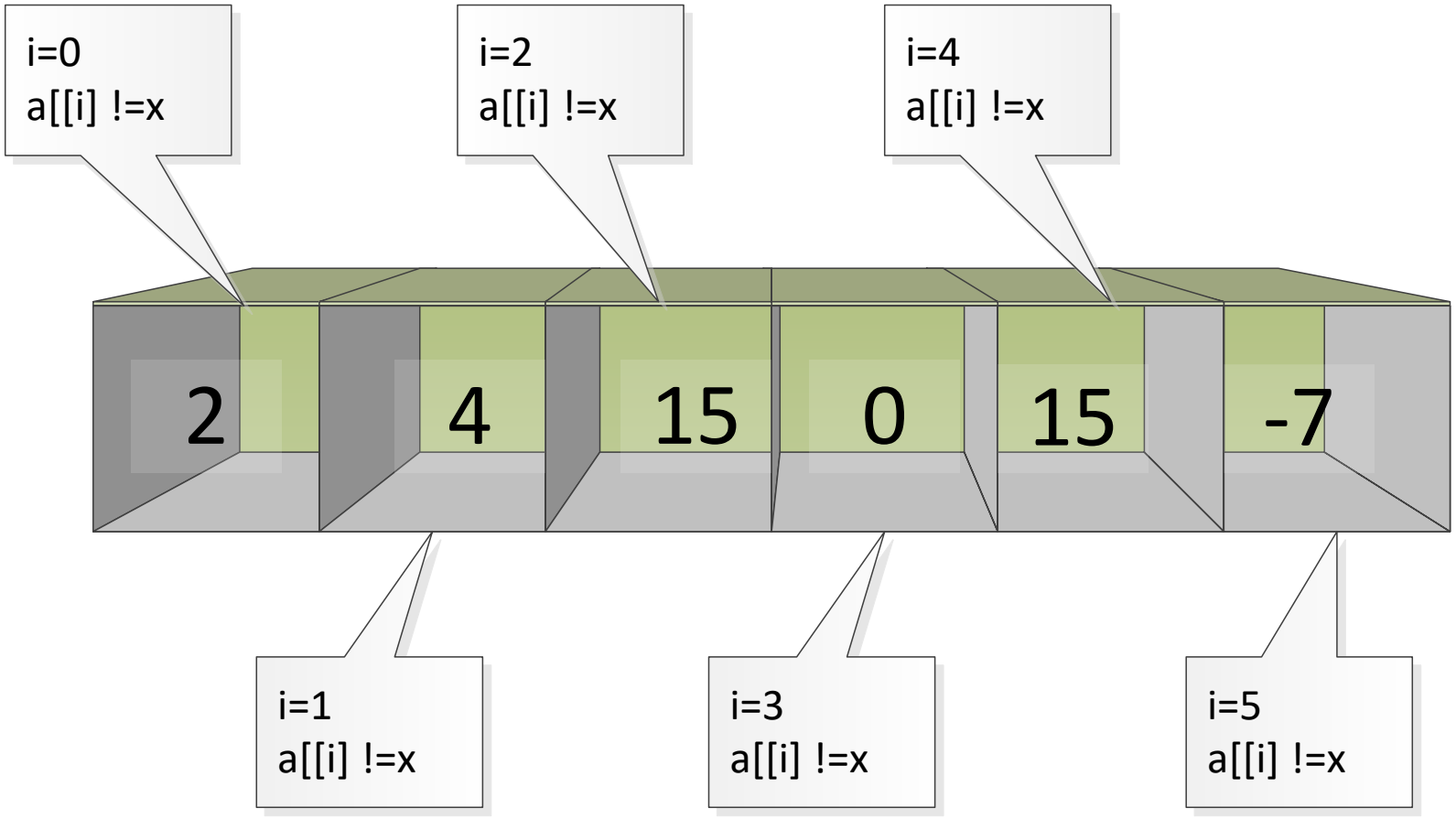
- Search sequentially for x=15



Output i=2, break out of the loop

Searching for Element (case 2)

- Search sequentially for x=8



Output -1

Searching

- Search sequentially

```
#define N 6
int sequentialSearch(int target, int a[N]) {
   for (int i=0; i<N; i++) {
      if (a[i] == x)
         return i;
   }
   return -1;
}
```

Searching in Sorted Arrays

- Sequential search in an array of size N takes (N+1)/2 rounds in average
  - (1 + 2 … + N-1 + N)/N = (N+1)/2

- Assume the array is already sorted, e.g.,

1	2	4	5	8	9	11	22	23	28	32

- Suppose the target is 22, can you do faster than sequential search?

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the middle element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the middle element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the middle element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found

Binary Search

- Assume the array is sorted (in ascending order)
- Compare the target with the middle element of the array
- If smaller, search in left
- If larger, search in right
- If equal, found

Binary Search

- In each round, binary search eliminates N/2 elements

- In comparison, sequential search eliminates only one

Binary Search

```
int binarySearch(int target, int a[N]) {
   int first=0, last=N-1, mid;
   while (target>=a[first] && target<=a[last]) {
      mid = (first+last)/2;
      if (target==a[mid])
         return mid;
      else if (target>a[mid])
         first = mid+1;
      else
         last = mid;
   }
   return -1;
}
```

Sort + Search

- Assume that you have a huge amount of data (out-of-order) and you need to frequently search in the database for different elements

- What should you do?

- What about if you only need to search once?

Today's Outline

- Array definition
- Array initialization
- Passing array to functions
- Array operations
- Multi-dimensional array

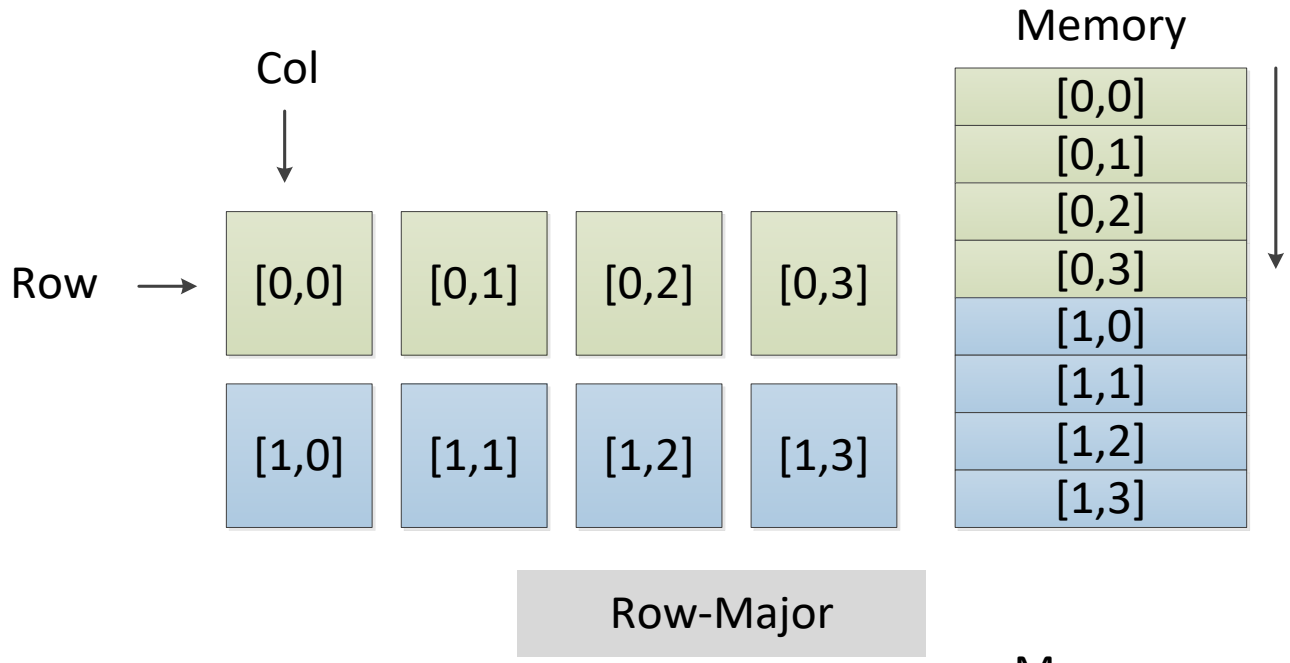
Multi-dimensional Array

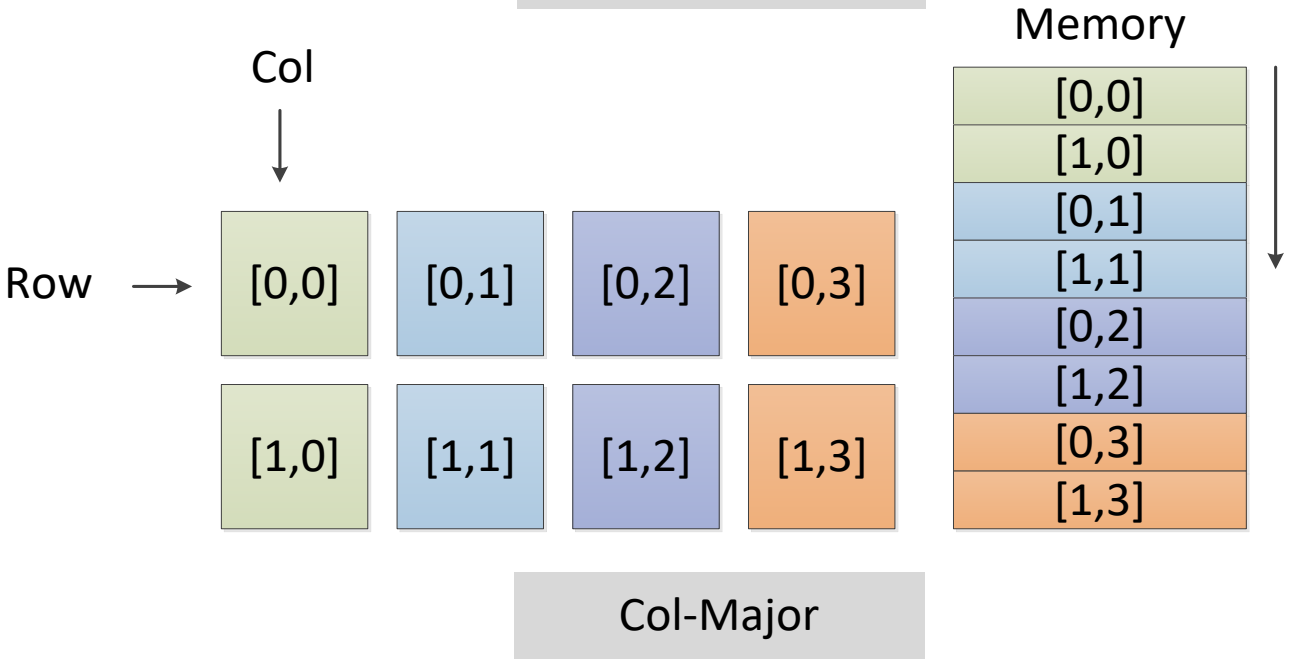
- Array with more than one index
  - on physical storage, multi-dimensional array is same as 1d array (*stored continuously in memory space*)
  - logical representation
- To define a 2d array, we specify the size of each dimension as follows

```
int page[2][3]; // [row][column]
```

```
12 34 11
13 56 99
```

- Stored in the "row-major" order, i.e.,
  - see next slide


Col
Row
[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]
Memory
[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]
Row-Major


Col
Row
[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]
Memory
[0,0]
[1,0]
[0,1]
[1,1]
[0,2]
[1,2]
[0,3]
[1,3]
Col-Major

2D Array Initialization

- Assign initial values row by row

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

- Assign initial values to the elements in the order they are arranged:

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

- Only assign initial values to some elements:

```
int page[2][3] = {{1},{4,5}};
```

```
1  0  0
4  5  0
```

- If all elements are assigned initial values, the length of the first dimension can be left unspecified:

```
int page[][3] = {1,2,3,4,5,6};
int page[2][] = {1,2,3,4,5,6}; X
```

Passing 2D Array to Function

- The way to pass a 2D array is similar as the 1D array
- For example: define a function which reads a 2D array as the input and sort each row of the input 2D array

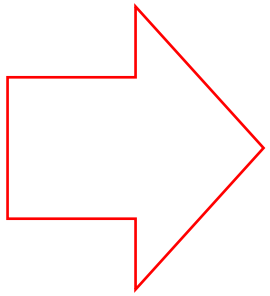
```
void sort2D(int x[][10]) {
    …
}
void main() {
    int y[20][10];
    sort2D(y);
}
```

the size of the first dimension is optional, while the size of the second dimension must be given

Example: Swapping Row and Column

- Swap elements of rows and columns of a 2D array with 2 rows and 3 cols
- Output the swapped array (please control the precision of each element, so that it contains only one digit in the decimal part)

```
1.2   1.34    2.564
3     4.123   4
```



```
1.2   3.0
1.3   4.1
2.6   4.0
```

Example: Swapping Row and Column

```
// the original array
double array1[2][3] = {{1.2,1.34,2.564},{3,4.123,4}};
// the swapped array
double array2[3][2];
// swap elements of rows and columns
for (int i = 0; i < 2; ++i){
    for (int j = 0; j < 3; ++j){
        array2[j][i] = array1[i][j];
    }
}
// output the swapped array
for (int i = 0; i < 3; ++i){
    for (int j = 0; j < 2; ++j){
        cout << fixed << setprecision(1) << array2[i][j] << "   ";
    }
    cout << endl;
}
```

Example: BMI Program

```
void main() {
  const int N=10;
  double data[N][2]; // N records, each holds weight and height
  int i, position;
  for (i=0; i<N; i++){
      cout << "Weigth(kg) Height(m):";
      cin >> data[i][0];
      cin >> data[i][1];
   }
   for (i=0; i<count; i++) {
       cout << "BMI for " << i+1 << "is: ";
       cout << data[i][0]/(data[i][1]*data[i][1]) << endl;
   }
}
```

Complex Problem: Maze

- A maze can be defined as a 2D array: bool maze[H][W];
  - where H is the height and W is the width
  - maze[0][0] is the entry and maze[H-1][W-1] is the exit
  - there's a Wall at grid (y, x) if maze[y][x] == true, otherwise space.

1. Generate a random maze where the ratio of walls is less then R
   - Hint: use (rand()%100)/100.0 to generate a random number between [0, 1)

2. Print the maze. You may use "char Wall = 177;" to represent a wall, and "char space = ' ';" to represent a space

Complex Problem: Maze (cont'd)

3. Find a path from entry (0, 0) to exit (H-1, W-1) (Hint: use recursion)

Problem: find a path from (y0, x0) to the exit

Complex Problem: Maze (cont'd)

3. Find a path from entry (0, 0) to exit (H-1, W-1) (Hint: use recursion)

Problem: find a path from (y0, x0) to the exit

Base case?

Recursive problem representation?

How to avoid loops?

How to remember the path when returning from recursion?

Complex Problem: Maze (cont'd)

- Global constants and variables:
  - maze dimensions (W, H),
  - cell types (Wall, Space, Flag),
  - arrays (maze, visited, flag)

```
#define W 32
#define H 16
const char Wall  = 177;
const char Space = ' ';
const char Flag  = '@';
// tell a cell is a wall or not
bool maze[H][W]
// cells visited during pathfinding
bool visited[H][W];
// mark the correct path once found
bool flag[H][W];
```

- Generates the maze with a given ratio R of walls
  - Ensures that the start (top-left) and end (bottom-right) are spaces.

```
void generateMaze(float R)
{
    for (int row = 0; row < H; row++) {
        for (int col = 0; col < W; col++)
            maze[row][col] = (rand()%100)/100.0 < R;
    }
    maze[H-1][W-1] = false;
    maze[0][0] = false;
}
```

Complex Problem: Maze (cont'd)

- Visualizing the Maze
  - Uses special characters
    - start ('S'),
    - end ('E'),
    - const char Flag = '@'
    - const char Wall = 177
    - const char Space = ' '

```
void printMaze()
{
    for (int row = 0; row < H; row++) {
        for (int col = 0; col < W; col++) {
            if (row==0 && col==0)
                cout << 'S'; // start
            else if (row==H-1 && col==W-1)
                cout << 'E'; // exit
            else if (flag[row][col])
                cout << Flag;
            else if (maze[row][col])
                cout << Wall;
            else
                cout << ' ';
        }
        cout << "\n";
    }
}
```

Complex Problem: Maze (cont'd)

- **Recursive Path Finding**
  - findPath(int y0, int x0): Tries to find a path recursively.
  - Checks all four possible directions (up, down, left, right).
  - Uses backtracking to explore potential paths

```
bool findPath(int y0, int x0){
    visited[y0][x0] = true;
    if (y0==H-1 && x0==W-1) // base case
        return true;   // exit found
    if (/*go right*/(x0+1 <  W && !maze[y0][x0+1] &&
       !visited[y0][x0+1] && findPath(y0, x0+1)) ||
        /*go down*/ (y0+1 <  H && !maze[y0+1][x0] &&
       !visited[y0+1][x0] && findPath(y0+1, x0)) ||
        /*go up*/   (y0-1 >= 0 && !maze[y0-1][x0] &&
       !visited[y0-1][x0] && findPath(y0-1, x0)) ||
        /*go left*/ (x0-1 >= 0 && !maze[y0][x0-1] &&
       !visited[y0][x0-1] && findPath(y0, x0-1)))
    {
        flag[y0][x0] = true;
        return true;
    }
    return false;
}
```

Breadth-First Search (BFS)

```
// Helper function to check if a position
       is valid for traversal.
bool isValid(int y, int x) {
    // True if position is within bounds
       and not a wall or already visited.
    return y >= 0 && y < H && x >= 0 && x
       < W && !maze[y][x] &&
       !visited[y][x];
}

bool findPath() {
    // p represents the previous
       positions in the pathfinding
       process.
    int p[H][W];
    // q is the queue used for breadth-
       first search.
    int q[H*W];
    int front = 0, end = 1;
    q[0] = 0; // Starting position (0,0).
    // While the destination is not visited and
    there are positions to process.
    while (!visited[H-1][W-1] && front < end) {
        // Convert 1D position in q to 2D
    coordinates.
        int y0 = q[front] / W, x0 = q[front] % W;
        // Directions for potential movement: up,
    down, left, right.
        int dy[] = {-1, 1, 0, 0};
        int dx[] = {0, 0, -1, 1};
        // Check all four directions.
        for (int i = 0; i < 4; i++) {
            int newY = y0 + dy[i], newX = x0 + dx[i];
            // If the new position is valid, mark it
    and enqueue it.
            if (isValid(newY, newX)) {
                q[end++] = q[front] + dy[i]*W +
                 dx[i];
                p[newY][newX] = q[front];
                visited[newY][newX] = true;
            }
        }
        front++;
    }
```

- Putting It All Together

```
int main() {
    float R;
    cout << "input the ratio of walls: ";
    cin >> R;
    srand(time(NULL));
    generateMaze(R);
    printMaze();
    // proper array init
    for (int row = 0; row < H; row++) {
        for (int col = 0; col < W; col++) {
            visited[row][col] = false;
            flag[row][col] = false;
        }
    }
    if (findPath(0, 0)) // DFS recursive
    // if (findPath) // BFS queuing
        cout << "\nFound Path\n\n";
    else
        cout << "\nNo Path\n\n";
    printMaze();
    return 0;
}
```

Summary

- Array is a sequence of variables of the *same* data type
- Array elements are *indexed* and can be *accessed* by the use of subscripts, e.g. array_name[1], array_name[4]
- Array elements are *stored contiguously* in memory space
- Array Declaration, Initialization, Searching and Sorting