Data Structures

---

4. Array basics and C++ Implementations

Abstract Data Types

- To understand the concept, let's decompose the ADT:

1. **Data Type:**
   - Set of values
   - Possible operations on these values
2. **Abstract:**
   - Any thing existing as a thought or as an idea but not having a physical or concrete existence

So,

"ADT is a general logical/ formal / mathematical model that defines set of values, and possible operations on these value without specifying implementation details"

Abstract Data Types – Examples

- Set of integers (i.e., Z)
- Operations
  - arithmetic operations (addition, subtraction, etc.)

- Flight reservation
  - List of seats
  - Operations
    - Find empty seat
    - Reserve a seat
    - Cancel a seat assignment

Data Structures

- A data structure is a physical implementation of an ADT
  - Each operation associated with ADT is implemented by one or more subroutines in the implementation

- Data structure usually refer to an organization of data in the main memory

Example: Airplane Flight Reservation (1)

- Consider example of an airplane flight with 10 seats to be assigned

- Data: set of 10 seats
- Operations
  - List available seats
  - Reserve a seat

- Implementation: How to store, access data?
  - 10 individual variables

Implementation: 10 Individual Variables

List available seats:

```
1. if seat1 == ' ';
       display 1
2. if seat2 == ' ';
       display 2
.
.
.

10. if seat10 == ' ';
       display 10
```

Reserve a seat:

```
1. Set Done to false
2. if seat1 ==' ';
   print "do you want seat #1??"
   Get answer
   if answer=='Y';
       set seat1 to 'X'
       set Done to True
3. if seat2 ==' ' and Done == false;
   print "do you want seat #2??"
   Get answer
   if answer=='Y';
       set seat2 to 'X'
       set Done to True
.
.
.
```

Example: Airplane Flight Reservation (2)

- Consider example of an airplane flight with 10 seats to be assigned

- Operations
  - List available seats
  - Reserve a seat

- Implementation: How to store, access data?
  - 10 individual variables
  - An array of variables

Implementation: An array of variables

List available seats:

```
For number ranging from 0 to max_seats-1, do:
        if seat[number] == ' ';
                Display number
```

Reserve a seat:

```
Reading number of seat to be reserved
if seat[number] is equal to ' ';
        set seat[number] to 'X'
Else
        Display a message that the seat having this number is
        occupied
```

Arrays ADT

- An array is defined as
  - Ordered collection of a fixed number of elements
  - All elements are of the same data type

- Basic operations
  - Direct access to each element in the array
  - Values can be retrieved or stored in each element

Properties of an Array

- **Ordered**
  - Every element has a well-defined position
  - First element, second element, etc.
- **Fixed size or capacity**
  - Total number of elements are fixed
- **Homogeneous**
  - Elements must be of the same data type (and size)
  - Use arrays only for homogeneous data sets
- **Direct access**
  - Elements are accessed directly by their position
  - Time to access each element is same
  - Different to sequential access where an element is only accessed after the preceding elements

Recap: Declaring Arrays in C/C++

```
dataType arrayName[intExp];
```

- `datatype` – Any data type, e.g., integer, character, etc.
- `arrayName` – Name of array using any valid identifier
- `intExp` – Constant expression that evaluates to a positive integer

Why constant?

- Example:
  - `const int SIZE = 10;`
  - `int list[SIZE];`

- Compiler reserves a block of consecutive memory locations enough to hold `SIZE` values of type `int`

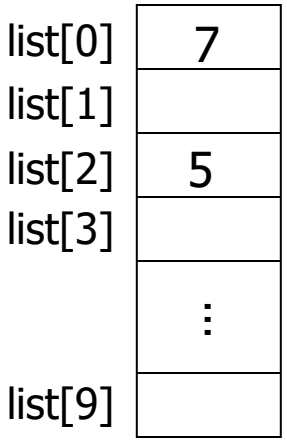
Recap: Accessing Arrays in C/C++

```
arrayName[indexExp];
```

- `indexExp` – called index, is any expression that evaluates to a positive integer

- In C/C++
  - Array index starts at 0
  - Elements of array are indexed 0, 1, 2, …, `SIZE-1`
  - `[ ]` is called array subscript operator

- Example
  - `int value = list[2];`
  - `list[0] = value + 2;`

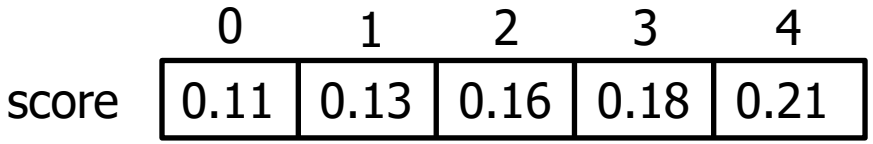


Recap: Array Initialization in C/C++ (1)

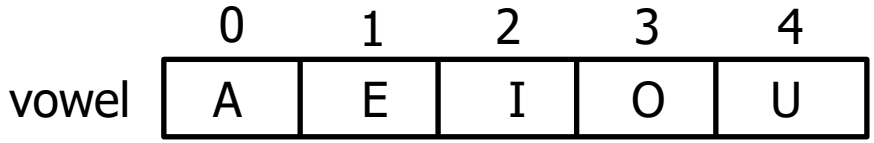
```
dataType arrayName[intExp]= {list of values}
```

- In C/C++, arrays can be initialized at declaration
  - `intExp` is optional: Not necessary to specify the size

- Example: Numeric arrays
  - `double score[ ] = {0.11, 0.13, 0.16, 0.18, 0.21}`

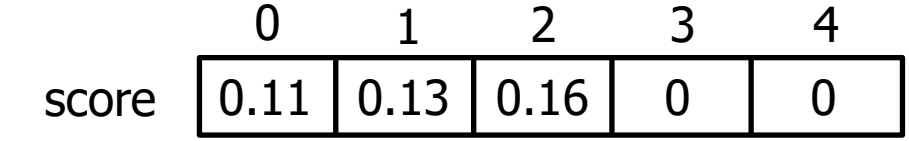


- Example: Character arrays
  - `char vowel [5] = { 'A', 'E', 'I', 'O', 'U' }`

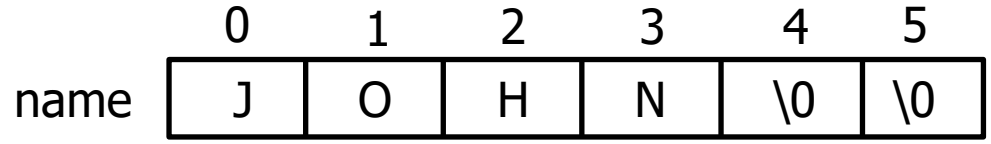


Recap: Array Initialization in C/C++ (2)

- Fewer values are specified than the declared size of an array
  - Numeric arrays: Remaining elements are assigned zero
  - Character arrays: Remaining elements contain null character '\0'
    - ASCII code of '\0' is zero
- Example
  - `double data[5] = {0.11, 0.13, 0.16}`



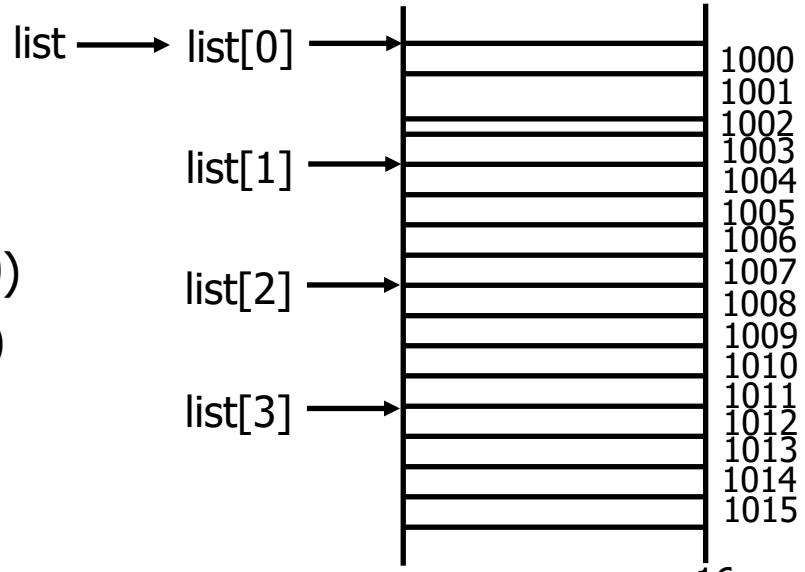
  - `char name[6] = {'J', 'O', 'H', 'N'}`



- If more values are specified than declared size of an array
  - Error is occurres: Handling depends on compiler

Array Address Translation (1)

- Consider an array declaration: `int list [4] = { 1, 2, 4, 5}`
  - Compiler allocates a block of four memory spaces
  - Each memory space is large enough to store an `int` value
  - Four memory spaces are contiguous
- Base address
  - Address of the first byte (or word) in the contiguous block of memory
  - Address of the memory location of the first array element
    - Address of element `list[0]`
- Memory address associated with `arrayName` stores the base address
- Example
  - `cout << list << endl;` (Print 1000)
  - `cout << *list << endl;` (Print val)
- `*` is dereferencing operator
  - Returns content of a memory location



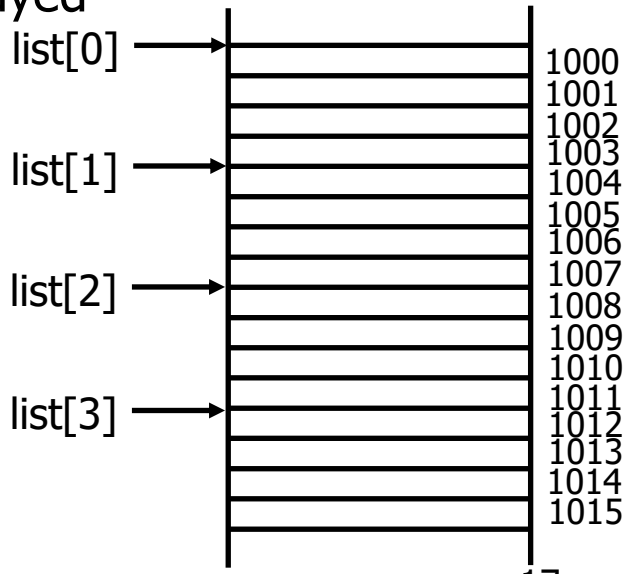
Array Address Translation (2)

- Consider a statement: `cout << list[3];`
  - Requires array reference `list[3]` be translated into memory address
  - Offset: Determines the address of a particular element w.r.t. base address

- Translation
  - Base address + offset = 1000 + 3 x `sizeof(int)` = 1012
  - Content of address 1012 are retrieved & displayed

- An address translation is carried out each time an array element is accessed

- What will be printed and why?
  - `cout << *(list+3) << endl;`



Questions

- Why does an array index start at zero?
- Why are arrays not passed by value?

Multidimensional Arrays

- Most languages support arrays with more than one dimension
  - High dimensions capture characteristics/correlations associated with data

- Example: A table of test scores for different students on several tests
  - 2D array is suitable for storage and processing of data

	Test 1	Test 2	Test 3	Test 4
Student 1	99.0	93.5	89.0	91.0
Student 2	66.0	68.0	84.5	82.0
Student 3	88.5	78.5	70.0	65.0
:	:	:	:	:
:	:	:	:	:
Student N	100.0	99.5	100.0	99.0

Multidimensional Arrays

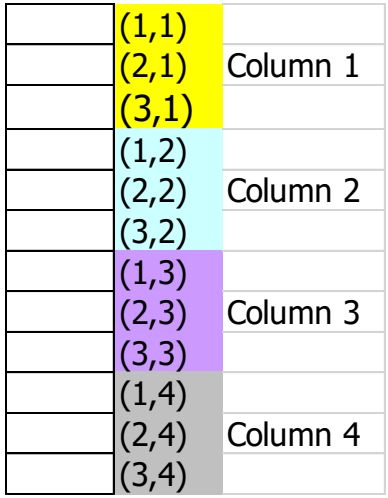
- More complicated than one dimensional arrays
- Memory is organized as a sequence of memory locations
  - One-dimensional (1D) organization
- How to use a 1D organization to store multidimensional data?
- Example:

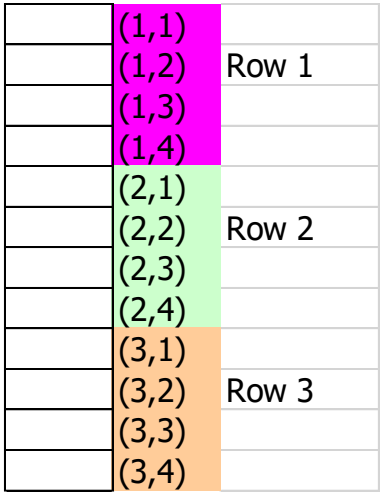
A	B	C	D
E	F	G	H
I	J	K	L

  - A character requires single byte
  - Compiler request to reserve 12 consecutive bytes
  - Two way to store consecutively, i.e., row-wise and column-wise

Two-dimensional Arrays in Memory

- Two ways to be represented in memory
  - Column majored
    - Column by column
  - Row majored
    - Row by row
  - Representation depends upon the programming language





Two Dimensional Arrays – Declaration

```
dataType arrayName[intExp1][intExp2];
```

- `intExp1` – constant expression specifying number of rows
- `intExp2` – constant expression specifying number of columns

- Example:
  - `const int NUM_ROW = 2, NUM_COLUMN = 4;`
  - `double score[NUM_ROW][NUM_COLUMN];`

- Initialization:
  -
```
double score[ ][4] = { {0.5, 0.6, 0.3},
                       {0.6, 0.3, 0.8}};
```
  - List the initial values in braces, row by row
  - May use internal braces for each row to improve readability

Two Dimensional Arrays – Processing

```
arrayName[indexExp1][indexExp2];
```

- `indexExp1` – row index
- `indexExp2` – column index

- Rows and columns are numbered from 0
- Use nested loops to vary two indices
  - Row-wise or column-wise manner

- Example
  - `double value = score[2][1];`
  - `score[0][3] = value + 2.0;`

score	[0]	[1]	[2]	[3]
[0]				2.7
[1]				
[2]		0.7		
[3]				
	⋮	⋮	⋮	⋮
[9]				

Higher Dimensional Arrays

- Example: Store and process a table of test scores
  - For several different students
  - On several different tests
  - Belonging to different semesters

```
const int SEMS = 10, STUDENTS = 30, TESTS = 4;
double gradeBook[SEMS][STUDENTS][TESTS];
```

- What is represented by `gradeBook[4][2][3]`?
  - Score of 3rd student belonging to 5th semester on 4th test

- All indices start from zero

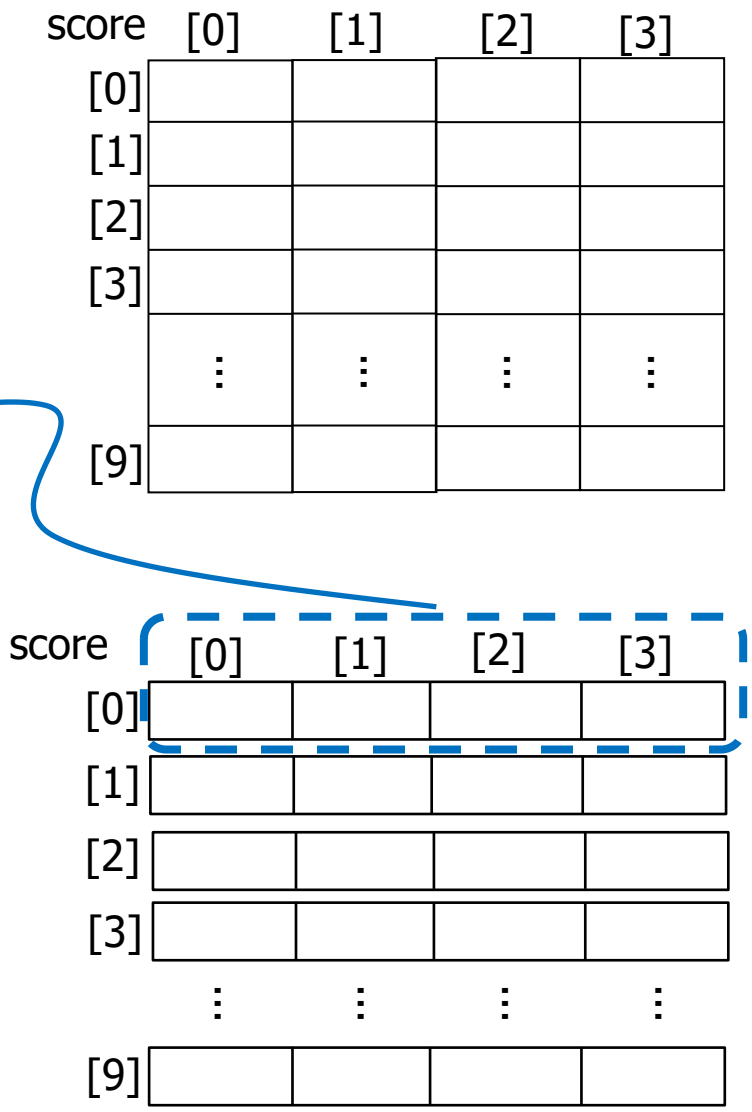
Array of Arrays (1)

- Consider the declaration
  - `double score[10][4];`

- Another way of declaration
  - One-dimensional (1D) array of rows

```
typedef double RowOfTable[4];
RowOfTable score[10];
```

- In detail
  - Declare score as 1D array containing 10 elements
  - Each of 10 elements is 1D array of 4 real numbers (i.e., double)



Array of Arrays (2)

- `score[i]`
  - Indicates i^{th} row of the table

- `score[i][j]`
  - Can be thought of as `(score[i])[j]`
  - Indicates j^{th} element of `score[i]`

Generalization:
An n-dimensional array can be viewed (recursively) as a 1D array whose elements are (n-1)-dimensional arrays

Array of Arrays – Address Translation

- How to access the value of `score[5][3]`?
- Suppose base address of score is 0x12348
- Address of 6th element of score array, i.e., `score[5]`
  - 0x12348 + 5 x `sizeof(RowOfTable)` = 0x12348 + 5 x (4 x 8)
    = 0x12348 + 0x A0
    =0x123E8
- Address of `score[5][3]`
  - Address of `score[5]` + 3 x `sizeof(double)` = 0x123E8 + (3 x 8)
    = 0x123E8 + 0x20
    =0x12408

```
typedef double RowOfTable[4];
RowOfTable score[10]
```

C/C++ Implementation of an Array ADT

As an ADT	In C/C++
Ordered	Index: 0,1,2, … SIZE-1
Fixed Size	`intExp` is constant
Homogeneous	`dataType` is the type of all elements
Direct Access	Array subscripting operator `[ ]`

Section 3—Array as ADT

- Operations on arrays
  - Search
    - Linear Search
    - Binary Search
  - Insertion
  - Deletion

Array Operations: **Search Algorithms**

- Operation of locating a specific data item in an array
  - Successful: If location of the searched data is found
  - Unsuccessful: Otherwise

- Complexity (or efficiency) of a search algorithm
  - Number of comparisons T(n) required to locate data within array
  - n is the number of elements within array

- Two algorithms for searching in arrays
  - Linear search (or sequential search)
  - Binary search

Linear Search

- Very intuitive and simple algorithm

Algorithm works as follows:

- Start from the first element of the array
- Use a loop to sequentially step through an array
- Compare each element with the item being searched
- Stop when data item is found or end of array is reached

Linear Search Algorithm

```
// numElems - maximum number of elements in the array
// value    - integer data (item) to be searched
// position - array subscript that holds value (if success)
//            -1 if value not found

int searchList(int list[], int numElems, int value)
{
  int index = 0;       // Used as a subscript to search array
  int position = -1;   // To record position of search value

while (index < numElelments)
  {
        if (list[index] == value)
        {
            position = index;
           return position;

        }
        index++;
   }
   return position;
}
```

Calling Function searchList

```
#include <iostream.h>

// Function prototype
int searchList(int [], int, int);
const int arrSize = 5;

void main(void)
{
    int tests[arrSize] = {87, 75, 98, 100, 82};
    int result;
    result = searchList(tests, arrSize, 100);
    if (result == -1)
        cout << "You did not earn 100 points on any test\n";
    else{
        cout << "You earned 100 points on the test ";
        cout << (result + 1) << endl;
    }
}
```

Program Output:
You earned 100 points on test 4.

Discussion

- Advantage of linear search is its simplicity
  - Easy to understand
  - Easy to implement
  - Does not require array to be in order (i.e., sorted)

- Disadvantage is its efficiency (or complexity)
  - Worst case complexity: T(n) = O(n)
    - Number of steps are proportional to number n of elements in an array
  - If there are 20,000 items in an array
    - And the Searched data item is stored at the 19,999th element index
    - Entire array has to be searched

Binary Search

- The binary search is more efficient than the linear search
  - Requires array to be in sorted order (i.e., ascending order)

Algorithm works as follows:

- Start searching from the middle element of an array

- If value of data item is less than the value of middle element
  - Algorithm starts over searching the first half of the array

- If value of data item is greater than the value of middle element
  - Algorithm starts over searching the second half of the array

- Algorithm continues halving the array until data item is found

Binary Search Algorithm

```
// numElems - maximum number of elements in the array
// value    - integer data (item) to be searched
// position - array subscript that holds value (if success)
//            -1 if value not found
int binarySearch(int array[], int numelems, int value)
{
   int first = 0, last = numelems - 1, middle, position = -1;

   while (first <= last){
      middle = (first + last) / 2;  // Calculate mid point takes rounded off integer
      if (array[middle] == value) {  // If value is found at mid
         position = middle;
         return position;
      }
      else if (array[middle] > value) //If value is in lower half
          last = middle - 1;
      else
          first = middle + 1;          // If value is in upper half
   }
   return position;
}
```

Calling Function binarySearch

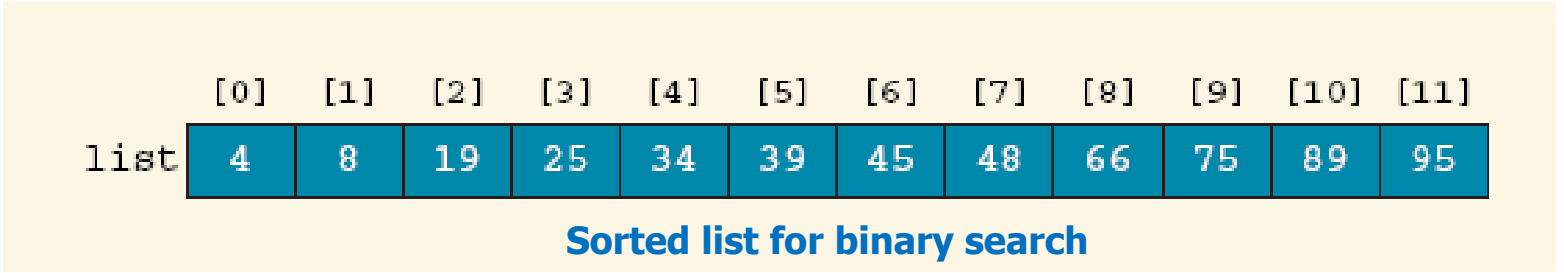
```
#include <iostream.h>
// Function prototype
int binarySearch(int [], int,
const int arrSize = 20;

void main(void)
{
    int empIDs[arrSize] = {101, 142, 147, 189, 199, 207, 222, 234, 289, 296,
                           310, 319, 388, 394, 417, 429, 447, 521, 536, 600};
    int result, empID;
    cout << "Enter the Employee ID you wish to search for: ";
    cin >> empID;
    result = binarySearch(empIDs, arrSize, empID);
    if (result == -1)
        cout << "That number does not exist in the array.\n";
    else {
        cout << "That ID is found at index " << result;
        cout << " in the array\n";
    }
}
```

Program Output:

Enter the Employee ID you wish to search for: 199
That ID is found at element 4 in the array

Binary Search Example



Sorted list for binary search

key = 89

Iteration	first	last	mid	list[mid]	
1	0	11	5	39	
2	6	11	8	66	
3	9	11	10	89	⇦ Value is found

key = 34

Iteration	first	last	mid	list[mid]	
1	0	11	5	39	
2	0	4	2	19	
3	3	4	3	25	
4	4	4	4	34	⇦ Value is found

Efficiency Of Binary Search

- Much more efficient than the linear search

- How long does this take (worst case)?
  - If the list has 8 elements
    - It takes at most 3 + 1 steps ($2^3 = 8$)
  - If the list has 16 elements
    - It takes at most 4 + 1 steps ($2^4 = 16$)
  - If the list has 64 elements
    - It takes at most 6 + 1 steps ($2^6 = 64$)

- Worst case complexity: $T(n) = O(\log_2(n))$
  - Takes $\log_2 n + 1$ steps

Array Operations

- **Insertion**
  - Operation of adding another element to an array
  - How many steps in terms of n (number of elements in array)?
    - At the end
    - In the middle
    - In the beginning
  - n steps at maximum (move items to insert at given location)
- **Deletion**
  - Operation of removing one of the elements from an array
  - How many steps in terms of n (number of elements in array)?
    - At the end
    - In the middle
    - In the beginning
  - n steps at maximum (move items back to take place of deleted item)

Algorithm for insert

Algorithm for Insert

//--- Insert *item* at position *pos* in a list.

// First check if there's room in the array

1. If *size* is equal to *capacity*
   Issue an error message and terminate this operation.

// Next check if the position is legal.

2. If $pos < 0$ or $pos > size$
   Signal an illegal insert position and terminate this operation.

   Otherwise:

   // Shift array elements right to make room for *item*

   a. For i ranging from *size* down to $pos + 1$:
      $array[i] = array[i - 1]$

   // Now insert *item* at position *pos* and increase the list size

   b. $array[pos] = item$

   c. *size*++

Here **size** indicates current number of elements in the array i.e., array has valid elements ranging from 0 to size-1

Algorithm for deletion

Algorithm for Delete

//--- Delete the element at position *pos* in a list.

// First check that list isn't empty

1. If *size* is 0
   Issue an error message and terminate this operation.

// Next check that *index* is legal

2. If $pos < 0$ or $pos \geq size$
   Issue an error message and terminate this operation.
   Otherwise:
   // Shift array elements left to close the gap
   a. For index i ranging from *pos* to $size - 2$:
      $array[i] = array[i + 1]$
   // Decrease the list size
   b. *size*– –

Limitation of Arrays

- An array has a limited number of elements
  - routines inserting a new value have to check that there is room
- Can partially solve this problem by reallocating the array as needed (how much memory to add?)
  - adding one element at a time could be costly
  - one approach - double the current size of the array
- A better approach: use a *Linked List*

Any Question So Far?