

SEMINAR IN C++ vs. Java

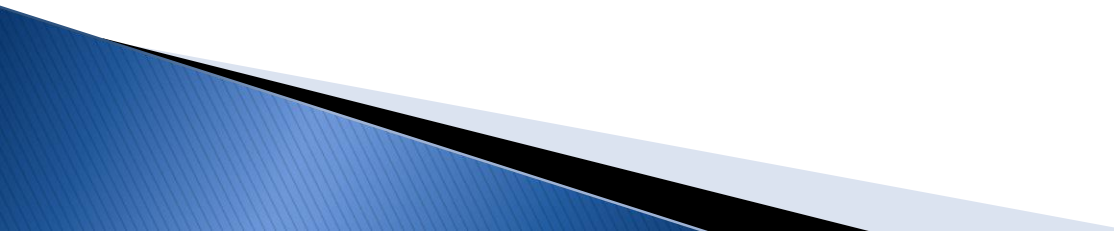
ARRAY IN C++

AMMAR QASEEM

Wed. 12 Jan. 2011



OUTLINES

- INTRODUCTION.
 - STSTIC ARRAY.
 - REPRESENTATION OF 1D AND 2D ARRAY
 - DYNAMIC MEMORY ALLOCATION.
 - DYNAMIC ARRAY (VECTORS).
- 

INTRODUCTION

- ▶ Array is containers in the memory for several values of the same type and have same name.
- ▶ An array is a series of elements placed in contiguous memory locations
- ▶ Most programs use arrays.
- ▶ **Advantages of arrays**
 - Random access in $O(1)$.
 - Ease of use.

INTRODUCTION

- Types :
 - Static Array, is a *fix-size* array.
 - Dynamic Array. **dynamic array, growable array, or resizable array**, is a random access, *variable-size* list data structure that allows elements to be added or removed.

STATIC ARRAY

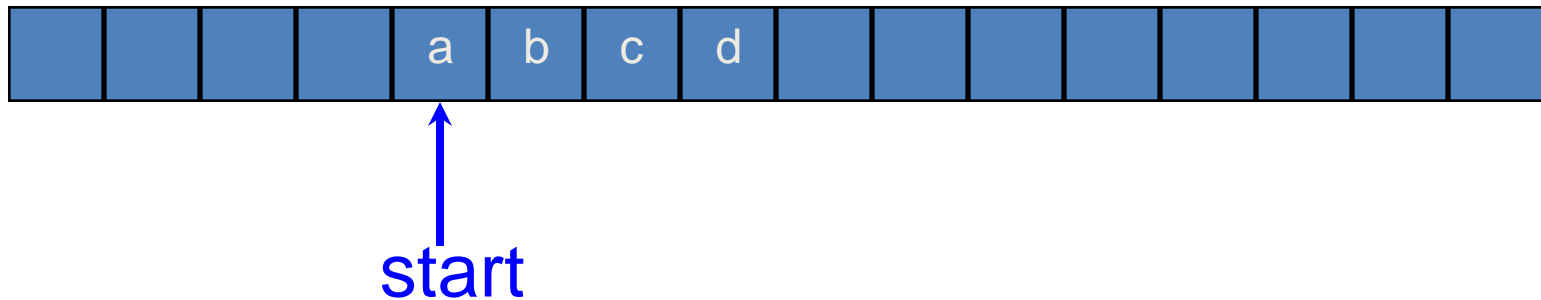
- ▶ Arrays store a constant-sized sequential set of blocks.
- ▶ Each block containing a value of the elected type under a single name.
- ▶ Individual elements are accessed by their position in the array which called *index*.
- ▶ What type of values and how many values to store must be defined as part of an array declaration.

STATIC ARRAY

- ▶ The *size* of array must be a const (integer greater than zero).
- ▶ You cannot use user input to declare an *array*, so the size of an array has to be known at compile time.

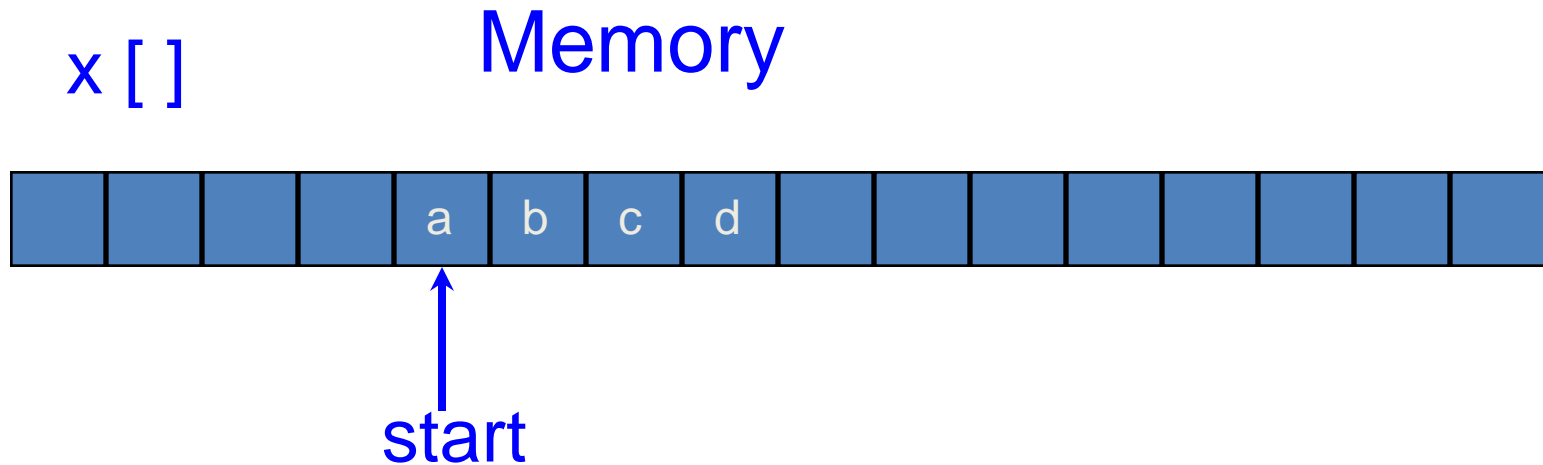
1D Array Representation

Memory



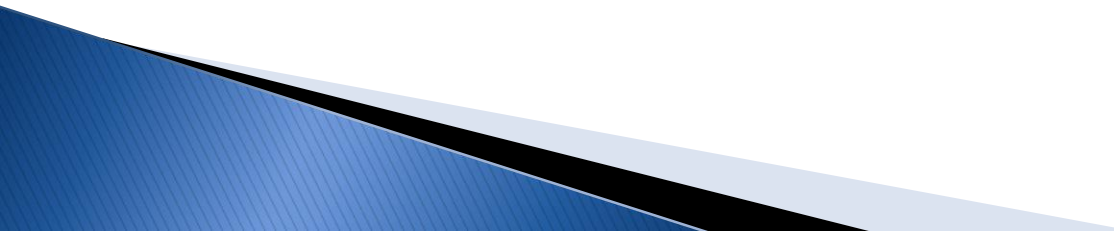
- ▶ 1-dimensional array $x = [a, b, c, d]$
- ▶ Map into contiguous memory locations.
- $\text{location}(x[i]) = \text{start} + i$

Space Overhead of 1D Array



space overhead = 4 bytes for `start`
 + 4 bytes for `x.length`
 = 8 bytes
(excludes space needed for the elements of `x`)

2D Array Representation

- ▶ There are three ways to represent 2 dimensional array in the memory:
 - (1) **Row-Major mapping representation,**
 - (2) **Column-Major mapping representation, and**
 - (3) **Array-of-array representation.**
- 

Row-Major Mapping

- Convert into 1D array by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.



- Example 3 x 4 array:

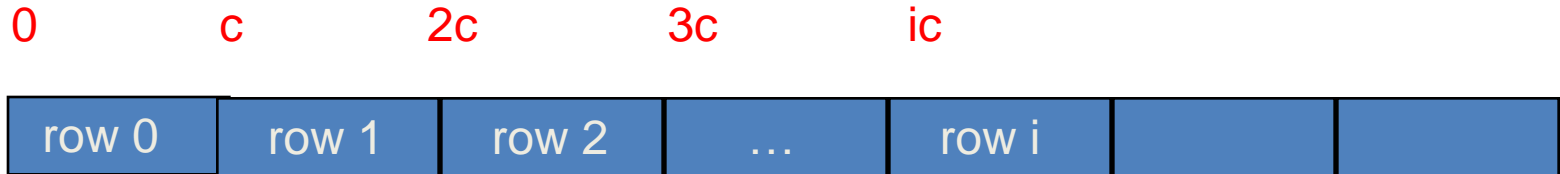
a , b , c , d

e , f , g , h

i , j , k , l

- We get {a, b, c, d, e, f, g, h, i, j, k, l}

Locating Element $x[i][j]$



- Assume $x [][]$ has r rows and c columns.
- Each row has c elements
- i rows to the left of row i
- so ic elements to the left of $x[i][0]$
- so $x[i][j]$ is mapped to position
 $ic + j$ of the 1D array

Space Overhead

Row-Major Mapping



↑
start

= 4 bytes for **start** of 1D array +
4 bytes for **length** of 1D array +
4 bytes for **c** (number of columns)
= **12** bytes

(**number of rows = length / c**)

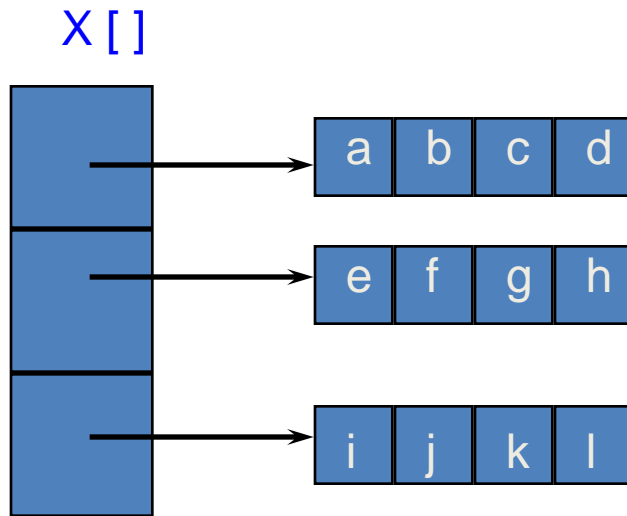
Column-Major Mapping

- ▶ Convert into 1D array by collecting elements by columns.
- ▶ Within a column elements are collected from top to bottom.
- ▶ Columns are collected from left to right.

a , b , c , d
e , f , g , h
i , j , k , l

- ▶ We get $y = \{a, e, i, b, f, j, c, g, k, d, h, l\}$

Array-of-array Representation



- ▶ This representation is called the **array-of-arrays** representation.
- ▶ Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- ▶ 1 memory block of size number of rows and number of rows blocks of size number of columns (**no. of rows + no. of rows * size of column**)

Array-of-array Representation

2-dimensional array x

a , b , c , d
e , f , g , h
i , j , k , l

view 2D array as a 1D array of rows

$x = [\text{row0}, \text{row1}, \text{row 2}]$

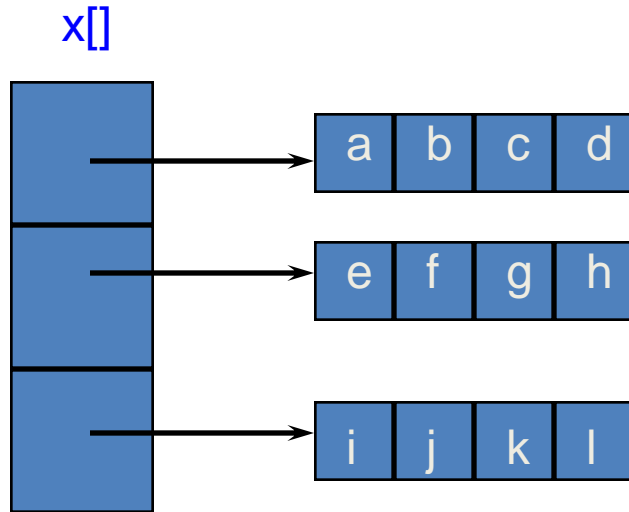
row 0 = [a, b, c, d]

row 1 = [e, f, g, h]

row 2 = [i, j, k, l]

and store as 4 1D arrays

Array-of-array Representation



4 separate

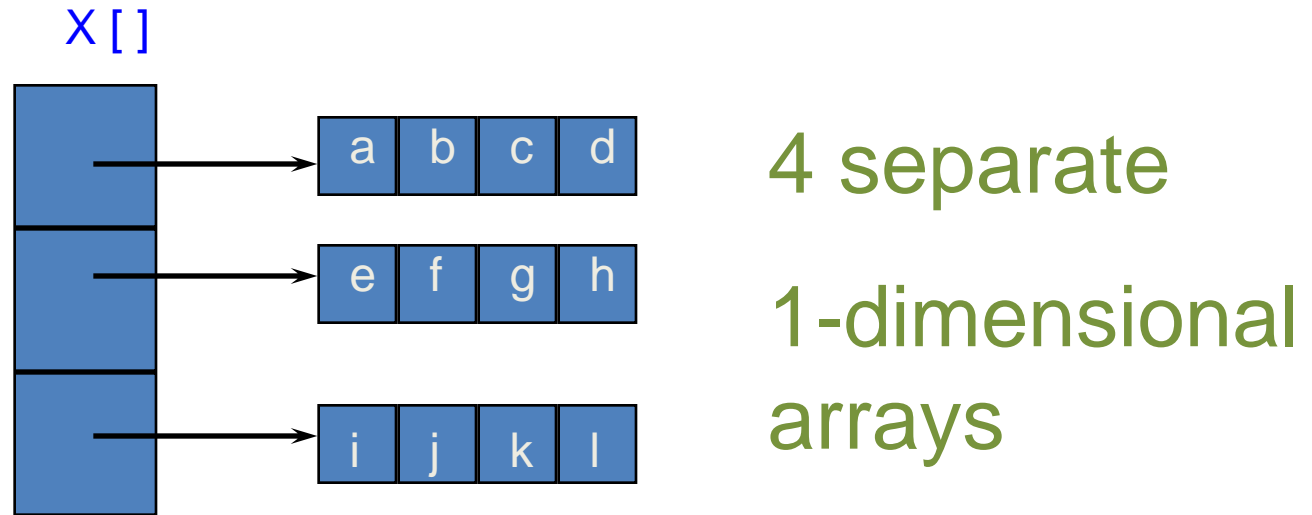
1-dimensional
arrays

`x.length = 3`

`x[0].length = x[1].length = x[2].length = 4`

Space Overhead

Array-of-array Representation



space overhead = $4 * 4$ bytes
= 16 bytes
= (number of rows + 1) x 4bytes

The 4 separate arrays are x , $x[0]$, $x[1]$, and $x[2]$.

Relation between Array and pointer

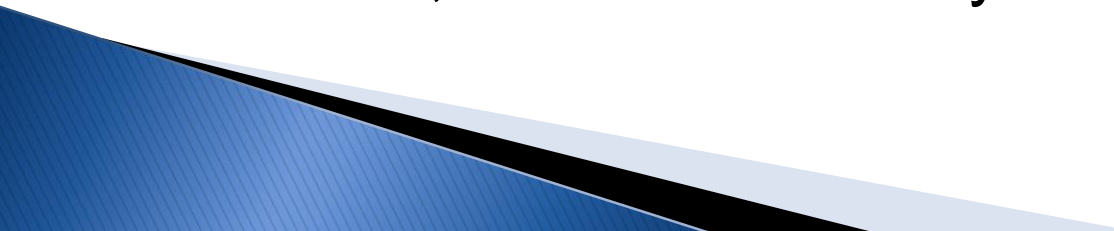
- ▶ Value of an array is in fact pointer to first element.
- ▶ Can use in programs, name of array as pointer to first element.
- ▶ **Note** : Array name is not a variable, so assignment to it illegal.

STATIC ARRAY

- **Disadvantages**

- 1) Constant size.
- 2) Large free sequential block to accommodate large arrays.
- 3) Buffer Overflow (no Bounds Checking) .

BUFFER OVERFLOW

- ▶ What is buffer overflow or buffer overrun ?
 - ▶ It is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory.
 - ▶ Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates.
 - ▶ This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security.
- 

BUFFER OVERFLOW

Buffer Overflow (no Bounds Checking)

- Because
 - Arrays are fixed size.
 - C and C++ provide no built-in protection against accessing or overwriting data in any part of memory.
 - Programmers forget to check bounds, or simply assume nothing can go wrong.
 - **Why no bounds checking on array indexes?**
 - Because it affects :
 - Runtime performance.

BUFFER OVERFLOW

- ▶ Solution 1 - Check array bounds by programmer

```
int a[1000];      // Declare an array of 1000 ints  
int n = 0;        // number of values in a.  
  
...  
while (n < 1000 && cin >> a[n]) {  
    n++;  
}
```

- ▶ Solution 2 - vectors - the correct solution

STATIC ARRAY

Solution for disadvantage of static array

- ▶ ***Dynamic allocation*** - Solution for fixed size restriction by using *new* operator.
- ▶ ***Vectors*** - Solution for buffer overflow, fixed size, unknown maximum and current size, ...

DYNAMIC MEMORY ALLOCATION

- ▶ **Dynamic memory allocation** is the allocation of memory storage for use in during the runtime of that program.
- ▶ By using *new* operator, you can allocate memory dynamically without having to know the size you should declare.
- ▶ A dynamic allocation exists until it is explicitly released, either by the programmer or by a *garbage collector* implementation

DYNAMIC MEMORY ALLOCATION

new and delete

- ▶ For dynamic memory allocation we use the *new* and *delete* keywords
- ▶ *new* operator, dynamically allocates memory on the *heap*.
- ▶ *new* attempts to allocate enough memory on the heap for the new data. If successful, it initializes the memory and returns the address to the newly allocated and initialised memory.
- ▶ Using *new* operator, it returns a pointer to the beginning of the new block of memory allocated.

DYNAMIC MEMORY ALLOCATION

- ▶ It is not possible to directly reallocate memory allocated with *new []*.
- ▶ To extend or reduce the size of a block :
 - Allocate a new block of adequate size,
 - Copy over the old memory, and
 - Delete the old block.

DYNAMIC MEMORY ALLOCATION

- ▶ Once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory by using *delete* operator.
- ▶ *delete* returns memory allocated by *new* back to the heap.
- ▶ A call to *delete* must be made for every call to *new* to avoid a *memory leak*.
 - **Note** : C++ programmers are responsible for memory management.

DYNAMIC MEMORY ALLOCATION

- ▶ After calling *delete*, the memory object pointed to is invalid and should no longer be used.
- ▶ Many programmers assign 0 (**null pointer**) to pointers after using delete to help minimize programming errors.

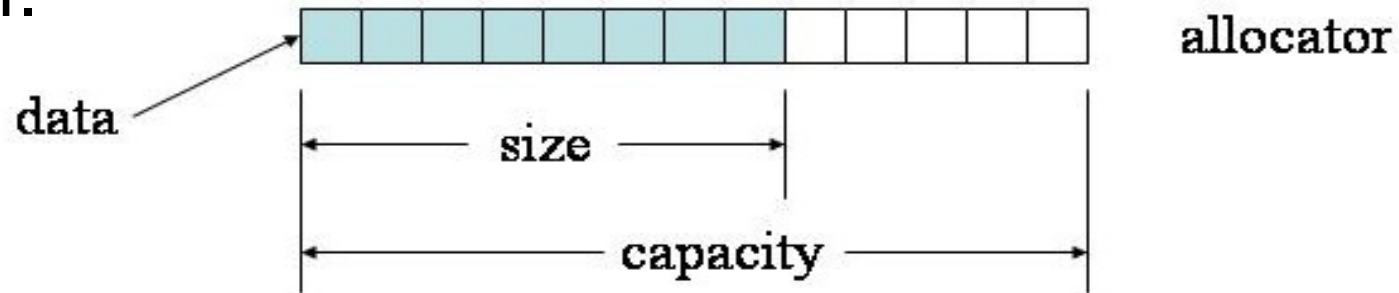
```
int size = 10;  
int *p_var = 0; // new pointer declared  
p_var = new int [size]; // memory dynamically allocated  
/* ..... other code ..... */  
delete [] p_var; // memory freed up  
p_var = 0; // pointer changed to 0
```

DYNAMIC ARRAY (VECTORS)

- ▶ Vectors are a kind of sequence containers.
- ▶ Vector is an expandable array.
- ▶ Elements stored in contiguous storage locations,
- ▶ Elements can be accessed not only using iterators but also using offsets on regular pointers to elements.

STL - VECTORS

- ▶ Data members holding the capacity and size of the vector.




- ▶ The size of the vector refers to the actual number of elements, while the capacity refers to the size of the internal array.
- ▶ The first "size" of elements are constructed (initialized) and the last "capacity - size" elements are uninitialized.

STL - VECTORS

- ▶ Vector containers are implemented as dynamic arrays; Just as regular arrays.
- ▶ Unlike regular arrays, storage in vectors is handled automatically, allowing it to be expanded and contracted as needed.
- ▶ An STL vector always allocates memory for its data on the *heap*. Heap allocation is slower than stack allocation.

DYNAMIC ARRAY (VECTORS)

- ▶ **Vectors are good at:**
 - ▶ Accessing individual elements by their position index (constant time).
 - ▶ Iterating over the elements in any order (linear time).
 - ▶ Add and remove elements from its end (constant amortized time).
- 

VECTORS

- ▶ Vectors provide a standard set of functions for accessing elements, **adding** elements to the end or anywhere, **deleting** elements and finding how many elements are stored.
- ▶ Example.

**Thank you for your
attention...**

► References

 <http://www.cplusplus.com/doc/tutorial/>

 <http://www.fredosaurus.com/notes-cpp/>

 <http://programming.im.ncnu.edu.tw/>

 [https://www.securecoding.cert.org/confluence/dis
play/cplusplus/](https://www.securecoding.cert.org/confluence/display/cplusplus/)

 <http://en.wikipedia.org/wiki/>