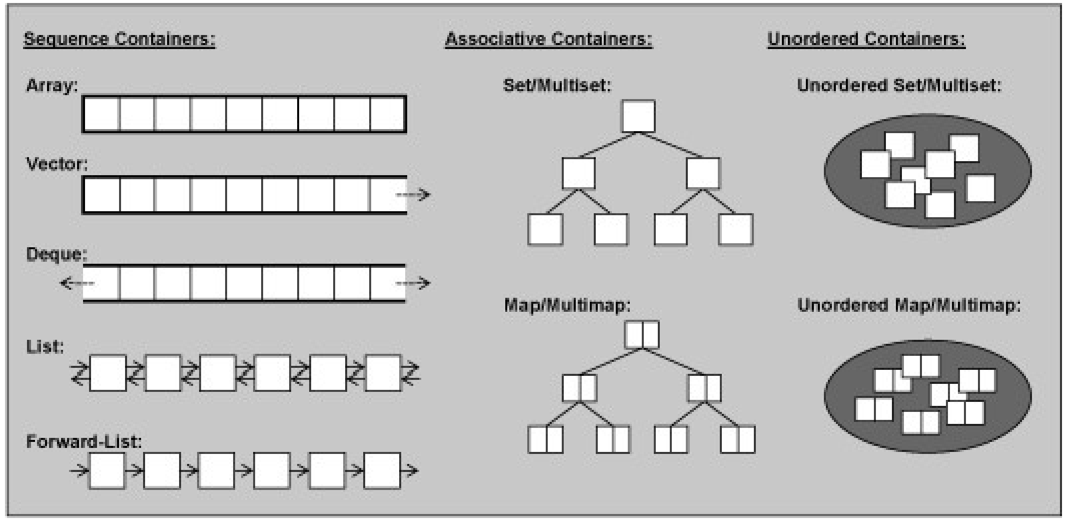
CPSC 131 – Data Structures

Array & Vector Abstract Data Types



Professor T. L. Bettens
Spring 2023

Array Abstract Data Type

Definitions:

Capacity - max number of elements that can be stored
Size - another name for Capacity – an array's size does not (can not) change

Fixed Capacity Array

- Capacity is constant
  - Set at container definition at design (compile) time

Two flavors

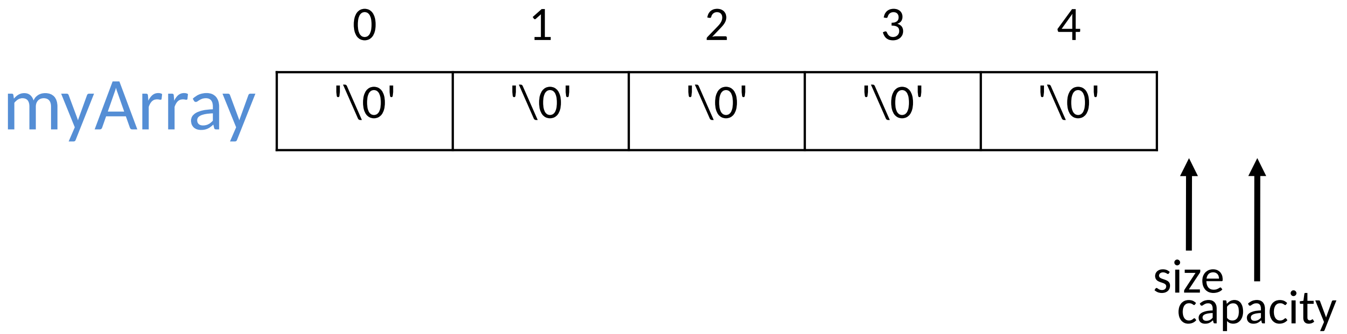
- Standard arrays
  - Smart wrapper around native array
  - std::array from the <array> library
  - Ex: `std::array<Student, 10> myArray;`
- Native arrays
  - aka C-Style or raw array
  - AVOID using these
  - Ex: `Student myArray[10];`

Array Abstract Data Type
The Abstraction - What can I do to an Array

std::array

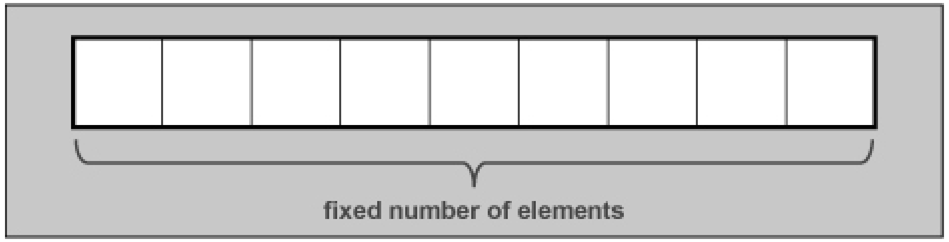
- Construct, destruct, assign
- Copy, compare
- Iterate
- Access elements
  - at, operator[], front, back
- Query
  - empty, size
- Operations
  - ----

size = 5
capacity = 5



myArray

0	1	2	3	4
'\0'	'\0'	'\0'	'\0'	'\0'



Vector Abstract Data Type

Definitions:

Capacity	- max number of elements that can be stored
Size	- number of elements that are stored

Fixed Capacity Vector

- Capacity is constant
  - Set at container construction during runtime, or
  - Set at container definition at design (compile) time

Extendable Capacity Vector

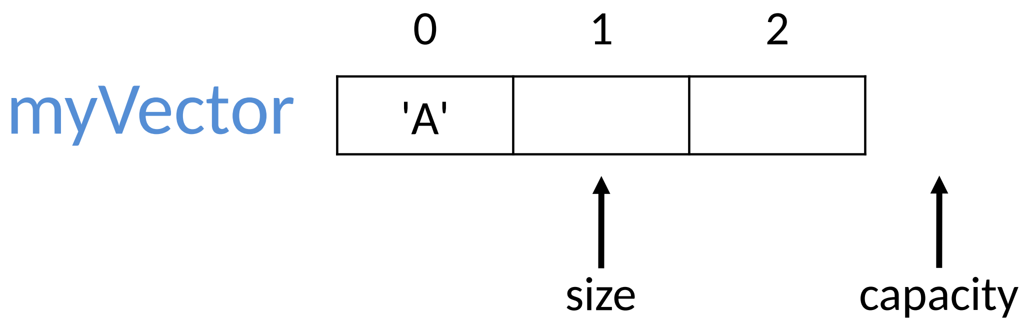
- Capacity is dynamic and changes during runtime
  - Initialized at container construction during runtime
  - Grows and shrinks during runtime

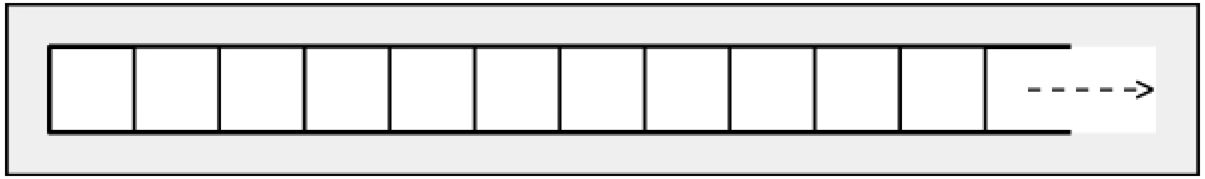
Vector Abstract Data Type
The Abstraction - What can I do to a Vector

Vector

- Construct, destruct, assign
- Copy, compare
- Iterate
- Access elements
  - at, operator[], front, back
- Query
  - empty, size, capacity
- Operations
  - insert, erase, clear, push_back, pop_back

size = 1
capacity = 3





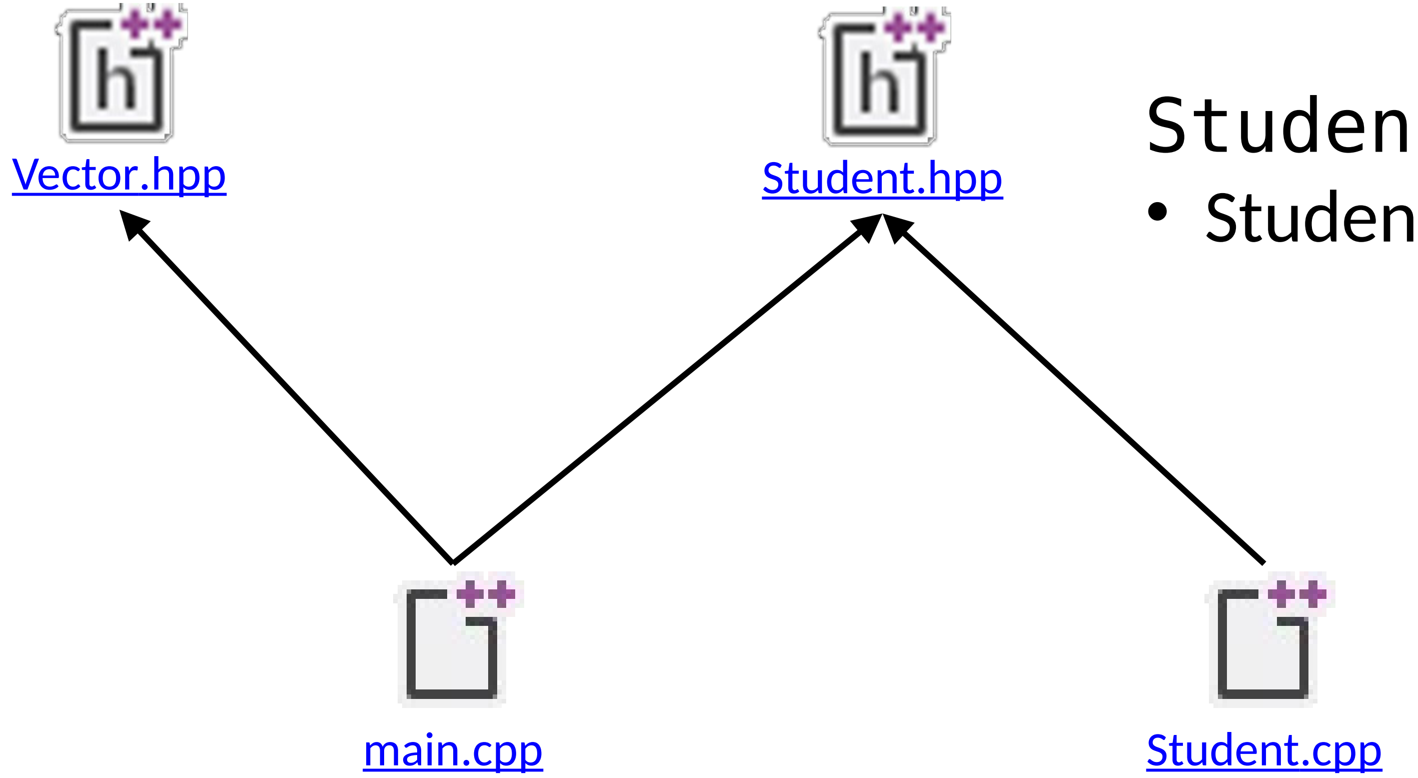
Array ^{vs} Vector – What’s the difference?

Arrays	Vectors
Capacity is constant	Two flavors, Fixed and Extendable Capacity
Size is constant	Size changes
Capacity and Size are always the same	Capacity and Size usually differ
Every cell always contains an element	Some cells do not contain an element
No insert and erase operations	Elements can be inserted and erased
Two template parameters	One template parameter

Array vs Vector – What’s the same?

Arrays and Vectors
Consecutive locations in memory
Indexed the same way
operator[] may over index avoidance is client’s responsibility
Comparison, assignment, initialization

Vector Implementation Example



`main.cpp` includes both

- Vector.hpp and Student.hpp

`Student.cpp` includes only

- Student.hpp

In Class Sketching Activity

Using the Implementation Examples, for each of the major operations of Arrays and Vectors, step through the code and sketch the resulting structure

Analysis of the Vector Abstract Data Type
Complexity Analysis (1)

Function	Analysis – std::array<T, S>	Analysis – std::vector<T> (Extendable Vector)
at()	O(1) Elements directly indexable	same
size()	O(1) Always returns S, as in std::array<T, S>	O(1) Returns the number of elements held
empty()	O(1)	same
clear()	Not available std::array<T, S> will always have S elements	O(n) All elements are destroyed and size set to zero *O(1) if only size set to zero, as in zyBook*

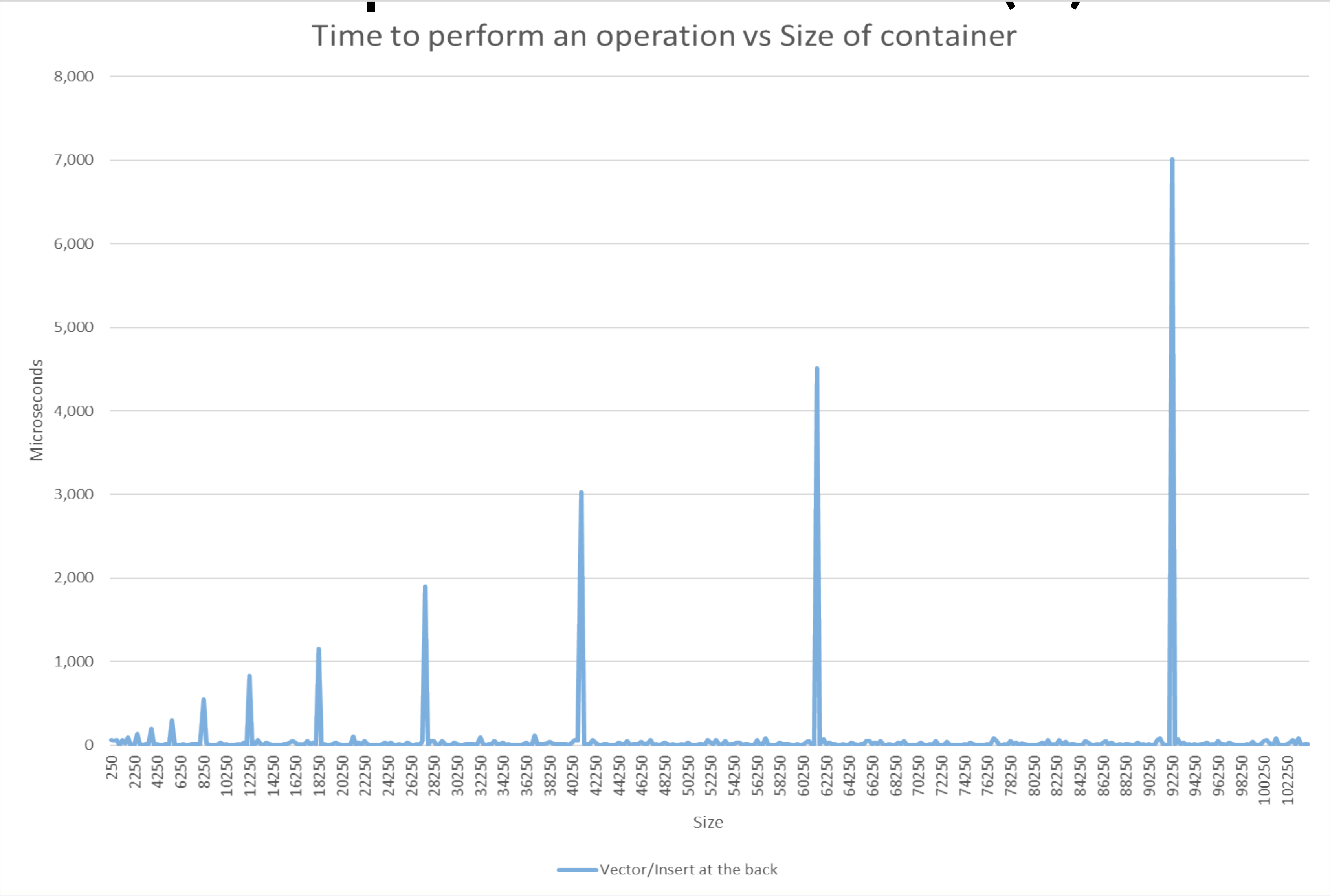
Analysis of the Vector Abstract Data Type
Complexity Analysis (2)

Function	Analysis – std::array<T, S>	Analysis – std::vector<T> (Extendable Vector)
push_back()	Not available std::array<T, S> will always have S elements	O(n) amortized to O(1) Special case of insert()
erase()	Not available std::array<T, S> will always have S elements	O(n) Have to "close the gap" which means N copies (worst case, N/2 copies average case)
splice	Not available	Not available

Efficiency Class Example: Amortized O(1)

<u>vector.push_back(data)</u>

- The “norm” is constant time
- But every now and then consumes linear time
- The interval between spikes doubles each time
- The severity of the spike increases each time



Analysis of the Vector Abstract Data Type
Complexity Analysis (3)

Function	Analysis – std::array<T, S>	Analysis – std::vector<T> (Extendable Vector)
insert()	Not available std::array<T,S> will always have S elements	O(n) (worst case) If space is not available, • get more space and copy N elements • Destroy N elements "Open a gap" which means N copies
default construction	O(n) container is never empty	O(1) creates an empty container
Equality $C_1 == C_2$	O(n)	same

Analysis of the Vector Abstract Data Type
Complexity Analysis (4)

Function	Analysis – std::array<T, S>	Analysis – std::vector<T> (Extendable Vector)
push_front	Not available std::array<T, S> will always have S elements	Not available
resize	Not available std::array<T, S> will always have S elements	O(n)
find	O(n) linear search from begin() to end() *(i.e. a[0] to a[size()-1])*	same

Analysis of the Vector Abstract Data Type
Complexity Analysis (4)

Function	Analysis – std::array<T, S>	Analysis – std::vector<T> (Extendable Vector)
Visit every element **e.g. print()**	O(n) Visiting every node from begin() to end()	same
Visit in reverse **e.g. print_reverse()**	O(n) Visiting every node from rbegin() to rend() Direction doesn’t matter	same