

CSC 212: Data Structures and Abstractions

06: Dynamic Arrays

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025



Dynamic arrays

C-style arrays

- Contiguous sequence of elements of identical type

✓ random access: $\text{base_address} + \text{index} * \text{sizeof}(\text{type})$

0	1	2	3		n-1
A[0]	A[1]	A[2]	A[3]	...	A[n-1]

array name: A

array length: n

- Statically allocated arrays

✓ allocated in the stack (fixed-length), size known at compile time

- Dynamic allocated arrays

✓ allocated in the heap (fixed-length), size may be determined at runtime

3

Practice

- Where are each of these variables allocated? (stack vs heap)
- Can the arrays change size during program execution?

```
void sort(int *arr, int size) {
    int i, j, temp;
    // sorting logic here
    // ...
}

int main() {
    int array[100];
    int *ptr;
    // ...
    ptr = new int[100];
    //...
    sort(ptr, 100);
    sort(array, 100);
    //...
    delete[] ptr;
    return 0;
}
```

4

Dynamic (growing) arrays

- Limitations of C-style arrays
 - ✓ size must be known at compile time
 - alternatively, use dynamic memory allocation
 - ✓ once created, array size does not change (inflexible)
- Dynamic arrays
 - ✓ can **grow or shrink in size** during run-time
 - essential for many applications, for example, a server keeping track of a queue of requests
 - ✓ **combine** the flexibility of dynamic memory allocation with the efficiency of fixed-length arrays
 - ✓ e.g. **std::vector** in C++, **ArrayList** in Java, **List** in Python, **Array** in JavaScript, **List** in C#, **Vec** in Rust, etc.

5

std::vector from C++ STL

```
#include <iostream>
#include <vector>

int main()
{
    // create a vector containing integers
    std::vector<int> v = {8, 4, 5, 9};

    // add two more integers to vector
    v.push_back(6);
    v.push_back(9);

    // overwrite element at position 2
    v[2] = -1;

    // print out the vector
    for (int n : v)
        std::cout << n << ' ';
    std::cout << '\n';
}
```

<https://en.cppreference.com/w/cpp/container/vector>

6

Designing a dynamic array class in C++

```
class DynamicArray {
private:
    int *arr;           // pointer to the (internal) array
    int capacity;       // total number of elements that can be stored
    int size;           // number of elements currently stored

public:
    DynamicArray();     // constructor
    ~DynamicArray();    // destructor
    void push_back(int val); // add an element to the end
    void pop_back();    // remove the last element
    const int& operator[](int idx) const; // read-only access at a specific index
    int& operator[](int idx); // access at a specific index (can modify)
    void insert(int val, int idx); // insert an element at a specific index
    void erase(int idx); // remove an element at a specific index
    void resize(int len); // change the capacity of the array
    int size();           // return the number of elements
    int capacity();       // return the capacity
    bool empty();         // check if the array is empty
    void clear();         // remove all elements, maintaining the capacity

    // additional methods can be added here
};
```

A class definition specifies the **data members** and **member functions** of the class. The data members are the attributes of the class, and the member functions are the operations that can be performed on the data members. The class definition is a blueprint for creating objects of the class.

7

Resizing dynamic arrays

• Grow

- ✓ when the array is full (size == capacity), **allocate a new array** with increased capacity, **copy elements** from old to new array, **deallocate old array**



• Shrink

- ✓ **optional optimization**, used when the number of elements is "significantly" less than the capacity, allocate a new array with decreased capacity, copy the elements from old to new array, and deallocate the old array

8

Grow by one

- When array is full, grow capacity to: **capacity + 1**
- starting from an empty array, **count number of array accesses (reads and writes)** for appending n elements (ignore cost of allocating/deallocating memory)

element	cost copy	cost append
1	2 x 0	1
2	2 x 1	1
3	2 x 2	1
4	2 x 3	1
5		
6		
n-1	2 x (n-2)	1
n	2 x (n-1)	1

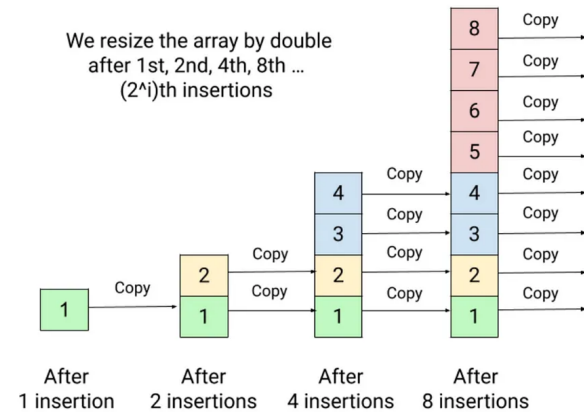
read and write
write

$$\begin{aligned}
 T(n) &= n + \sum_{i=0}^{n-1} 2i \\
 &= n + 2 \sum_{i=0}^{n-1} i \\
 &= n + 2 \left(\frac{n(n-1)}{2} \right) \\
 &= \Theta(n^2) \quad \leftarrow \text{cost of adding } n \text{ elements}
 \end{aligned}$$

The amortized cost of inserting an element is $\Theta(n)$, meaning that any sequence of n insertions takes at most $\Theta(n^2)$ time in total.

9

Repeated doubling



<https://itsfuad.medium.com/how-dynamic-arrays-actually-work-bff5bb5749bb>

10

Grow by factor

- When array is full, grow capacity to: **capacity * factor**
- called **repeated doubling** when factor == 2
- starting from an empty array, **count number of array accesses (reads and writes)** for appending n elements (ignore cost of allocating/deallocating memory)

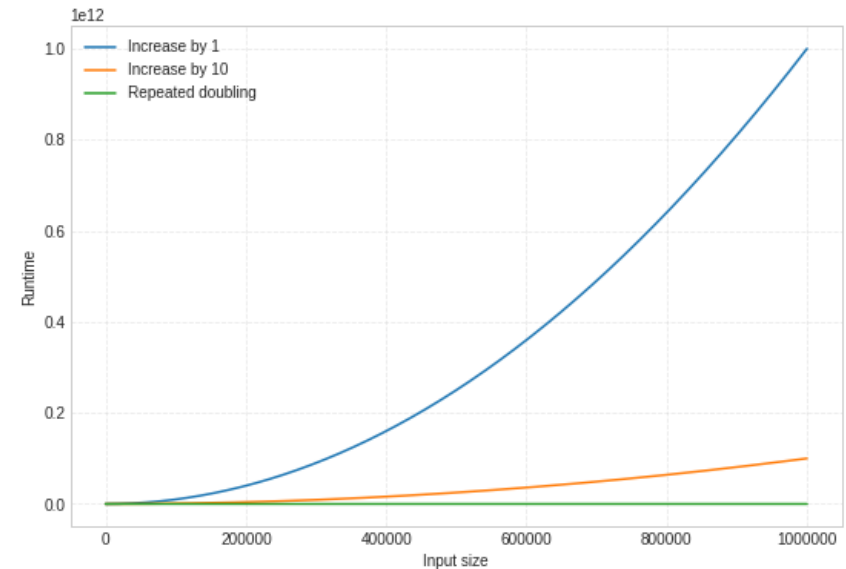
element	cost copy	cost append
1	2 x 0	1
2	2 x 1	1
3	2 x 2	1
4	—	1
5	2 x 4	1
6	—	1
7	—	1
8	—	1
9	2 x 8	1
10	—	1
n-1	—	1
n	—	1

read and write
write

$$\begin{aligned}
 T(n) &= n + 2 \sum_{i=0}^{\log n - 1} 2^i \\
 &= n + 2 \left(\frac{2^{\log n} - 1}{2 - 1} \right) \\
 &= n + 2(n - 1) \\
 &= \Theta(n) \quad \leftarrow \text{cost of adding } n \text{ elements}
 \end{aligned}$$

The amortized cost of inserting an element is $\Theta(1)$, meaning that any sequence of n insertions takes at most $\Theta(n)$ time in total.

11



12

Shrinking the array

- May half the capacity when array is **one-half** full
 - ✓ **worst-case** when the array is full and we alternate between adding and removing elements
 - each alternating operation would require resizing the array
- **More efficient resizing**
 - ✓ half the capacity when the array is **one-quarter** full
- **In practice ...**
 - ✓ most standard implementations do not automatically shrink capacity
 - avoids performance penalties from frequent resizing
 - ✓ instead, they provide explicit operations like `shrink_to_fit()` that allow the programmer to request size reduction when deemed necessary

13

Growth factors by language

- **C++ (`std::vector`)**
 - ✓ grow by 1.5 or 2.0 times the current capacity (depending on compiler)
 - ✓ shrink when the array is one-quarter full
- **Java (`ArrayList`)**
 - ✓ grow by 1.5 of the current capacity
 - ✓ shrink when the array is one-half full
- **Python (`List`)**
 - ✓ grow by ~ 1.125 times the current capacity
 - ✓ shrink when the array is one-quarter full
- **Rust (`std::vec::Vec`)**
 - ✓ grow by 2 times the current capacity
 - ✓ shrink when the array is one-half full

Information taken from `claude.ai` (some confirmed)

bottom line: growth factors range from ~ 1.12 to ~ 2 depending on language used

14

Practice

- Complete the following table with rates of growth using Θ notation
 - ✓ assume we implement a dynamic array with repeated doubling and no shrinking

Operation	Best case	Average case	Worst case
Append 1 element			
Remove 1 element from the end			
Insert 1 element at index <code>idx</code>			
Remove 1 element from index <code>idx</code>			
Read element from index <code>idx</code>			
Write (update) element at index <code>idx</code>			

15

Aside: references in C++

```

non-const References
int i = 2;
int& ri = i; // reference to i

ri and i refer to the same object / memory location:

cout << i << '\n'; // 2
cout << ri << '\n'; // 2

i = 5;
cout << i << '\n'; // 5
cout << ri << '\n'; // 5

ri = 88;
cout << i << '\n'; // 88
cout << ri << '\n'; // 88

• references cannot be "null", i.e., they must always refer to an object
• a reference must always refer to the same memory location
• reference type must agree with the type of the referenced object

int i = 2;
int k = 3;
int& ri = i; // reference to i
ri = k; // assigns value of k to i (target of ri)
int& r2; // X COMPILER ERROR: reference must be initialized
double& r3 = i; // X COMPILER ERROR: types must agree
    
```

<https://hackingcpp.com/cpp/lang/references.html>

16