CSC 212: Data Structures and Abstractions

06: Dynamic Arrays

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

THINK BIG WE DO℠

Dynamic arrays

C-style arrays

- Contiguous sequence of elements of identical type
  - ✓ random access: `base_address + index * sizeof(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

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;
}
```

Array insertions

- Insert at end (a.k.a. append or push_back)
  - add element after the last current element
  - time complexity: Θ(1) — if space available
  - why fast? no shift necessary, just place element at next index
    - e.g., [10, 20, 30] → append 40 → [10, 20, 30, 40]
- Insert at front (a.k.a. prepend or push_front)
  - add element at index 0
  - time complexity: Θ(n) — always linear time
  - why slower? must shift all existing elements one position right
    - e.g., [10, 20, 30] → prepend 40 → [40, 10, 20, 30]
- Insert at middle
  - add element at any arbitrary index
  - time complexity: Θ(n) — worst-case linear time
  - why slower? must shift all elements after insertion point
    - e.g., [10, 20, 40] → insert 50 at index 1 → [10, 50, 20, 40]

Array deletions

- Delete at end (a.k.a. pop_back)
  - remove element from the last position
  - time complexity: Θ(1) — constant time
  - why fast?: no shift necessary, just remove the last element
    - e.g., [10, 20, 30, 40] → pop_back → [10, 20, 30]
- Delete at front (a.k.a. pop_front)
  - remove element at index 0
  - time complexity: Θ(n) — always linear time
  - why slower? must shift all remaining elements one position left
    - e.g., [40, 10, 20, 30] → pop_front → [10, 20, 30]
- Delete at middle
  - remove element at any arbitrary index
  - time complexity: Θ(n) — worst-case linear time
  - why slower? must shift all elements after deletion point left
  - e.g., [10, 50, 20, 40] → delete at index 1 → [10, 20, 40]

Dynamic (growing) arrays

- Limitations of C-style arrays
  - size <u>must be known at compile time</u>
    - alternatively, use dynamic memory allocation
  - once created, array size does not change (inflexible)
- Dynamic arrays
  - can **<u>grow or shrink in size</u>** during run-time
    - essential for many applications, for example, a server keeping track of a queue of requests
  - **<u>combine</u>** 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.

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

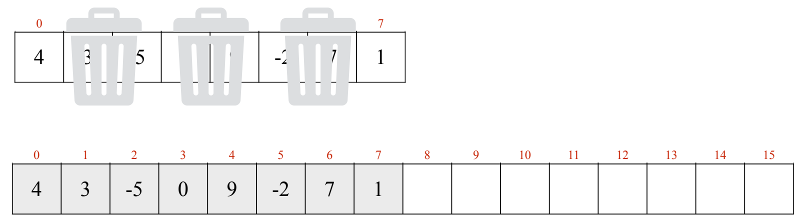
Designing a dynamic array class in C++

```
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.

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

Grow by one

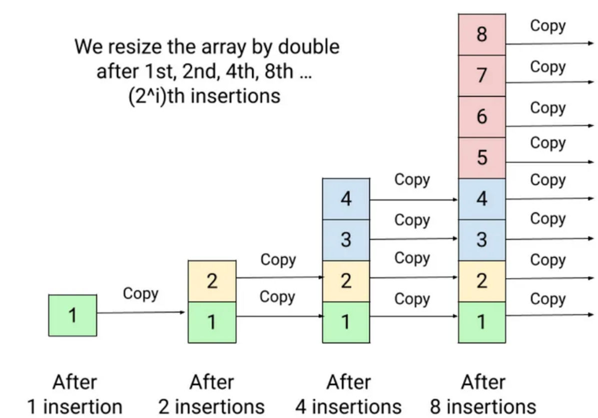
- When array is full, grow capacity to: `capacity + 1`
  - starting from an empty array, count number of array accesses (**reads and writes**) for appending n elements (ignore cost of allocating/deallocating memory)

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

$$T(n) = n + \sum_{i=0}^{n-1} 2i$$
$$= n + 2\sum_{i=0}^{n-1} i$$
$$= n + 2\left(\frac{n(n-1)}{2}\right)$$
$$= \Theta(n^2) \longleftarrow \text{cost of adding } n \text{ elements}$$

Inserting into an array of size n costs $\Theta(n)$. Performing n insertions from empty costs $\Theta(n^2)$ in total, which means the amortized cost per insertion is $\Theta(n)$.

Repeated doubling



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

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

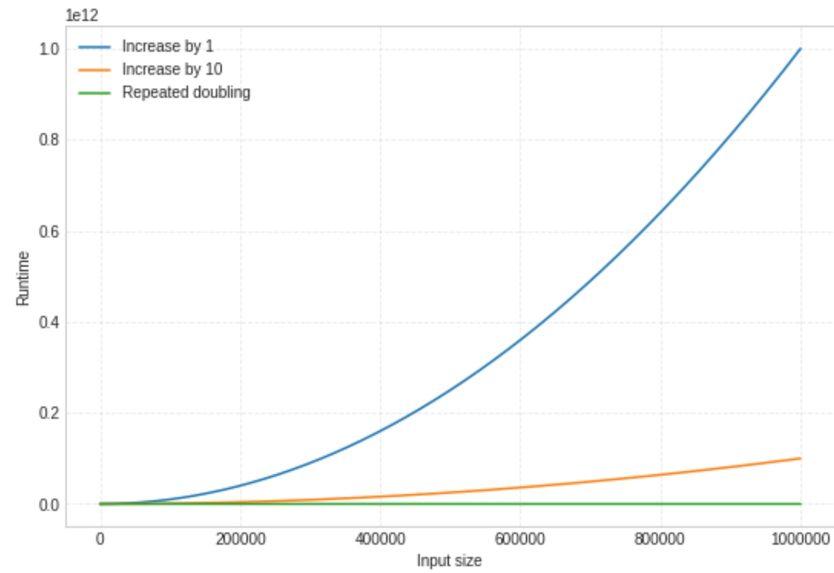
$$T(n) = n + 2\sum_{i=0}^{\log n - 1} 2^i$$

$$= n + 2\left(\frac{2^{\log n} - 1}{2 - 1}\right)$$

$$= n + 2(n-1)$$

$$= \Theta(n) \leftarrow \text{cost of adding n elements}$$

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



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

Growth factors by language

- C++ (`std::vector`)
  - grow by 1.5 (MS Visual C++) or 2.0 (g++/clang)
- Java (`ArrayList`)
  - grow by 1.5
- Python (`List`)
  - grow by ~1.125
- Rust (`std::vec::Vec`)
  - grow by 2

Growth factors typically range from ~1.12 to ~2 depending on language/compiler used

https://en.wikipedia.org/wiki/Dynamic_array

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 idx			
Remove 1 element from index idx			
Read element from index idx			
Write (update) element at index idx			

Aside: references in C++

non-`const` References

```cpp
int   i = 2;
int& ri = i;  // reference to i
```

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

```cpp
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

```cpp
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;         // ✗ COMPILER ERROR: reference must be initialized
double& r3 = i;  // ✗ COMPILER ERROR: types must agree
```

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