CSC 212: Data Structures and Abstractions

05: Dynamic Arrays

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

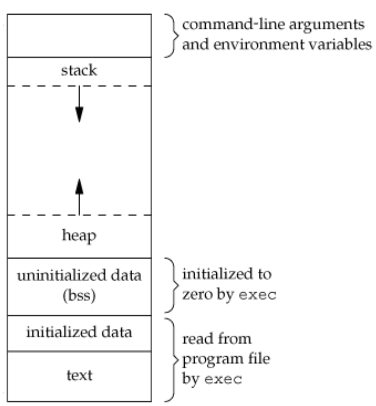
Spring 2026

Administrativia

- Midterm preparation
  - ✓ 2 problem sets
  - ✓ textbook
  - ✓ lecture materials
- Programming non-negotiables
  - ✓ basic data types
  - ✓ classes and objects
  - ✓ pointers
  - ✓ dynamic memory allocation
  - ✓ templates (*)

Memory layout

- What is a **process**?
  - ✓ an **instance** of a running program
  - ✓ has its own memory space, isolated from other processes
  - ✓ managed by the OS, which schedules execution and allocates resources
- Memory in a process (each process has its own memory layout)
  - ✓ Text Segment (code)
    - contains compiled instructions
    - marked read-only
  - ✓ Data (global/static variables)
    - includes initialized, uninitialized (BSS), and constant data
    - size fixed at compile-time; addresses resolved at linking
  - ✓ Heap
    - for dynamic memory allocation (`new`/`malloc`)
    - grows upward (low => high addresses)
  - ✓ Stack
    - holds local variables and function call frames
    - grows downward (high => low addresses)



Static vs Dynamic memory

- Stack memory (automatic allocation)
  - ✓ size known at compile-time, automatic allocation, scope-based lifetime
- Dynamic memory (**heap**)
  - ✓ size determined at runtime, manual or RAII-managed allocation, lifetime controlled by programmer
- Why dynamic memory?
  - ✓ variable-sized data (user input, large arrays)
  - ✓ flexible data structures (linked lists, trees, graphs)
- Critical Rules
  - ✓ every `new` must have exactly one matching `delete`
  - ✓ deleting the same pointer twice => *undefined behavior*
  - ✓ accessing deleted memory => *undefined behavior*

```
undefined behavior:
anything can happen:
crashes, core dumps,
segmentation faults,
corrupted memory, or
silently wrong results.
```

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;
}
```

- **Who owns ptr?**
  - The code that calls `new` / `new[]` is responsible for calling `delete` / `delete[]`
- **What happens if `delete[]` is forgotten?**
  - A memory leak. The allocated memory is never released during the program's lifetime.
- **Why `std::vector`?**
  - It provides automatic, safe memory management

Dynamic arrays

Run this code

```
import time

n = 100000

start = time.time()
array = []
for i in range(n):
    array.append('s')
print(time.time() - start)

start = time.time()
array = []
for i in range(n):
    array = array + ['s']
print(time.time() - start)
```

How are lists implemented in CPython?

CPython's lists are really variable-length arrays. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure. This makes indexing a list an operation whose cost is independent of the size of the list or the value of the index. When items are appended or inserted, the array of references is resized; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

`CPython is the reference implementation of the Python programming language (primarily written in C)`

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]

- Stack-allocated arrays
  - ✓ fixed size for the lifetime of the array
  - ✓ cannot be resized after allocation
  - ✓ size often known at compile time
- Dynamic allocated arrays
  - ✓ allocated in the heap (fixed-length), size may be determined at runtime

Common array operations

- Insert at end (a.k.a. append or push_back)
  - ✓ add element after the last current element
  - ✓ time complexity: $\Theta(1)$ — 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: $\Theta(n)$ — 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 index
  - ✓ add element at any arbitrary index
  - ✓ time complexity: $\Theta(n)$ — 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: $\Theta(1)$ — 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: $\Theta(n)$ — 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 index
  - ✓ remove element at any arbitrary index
  - ✓ time complexity: $\Theta(n)$ — 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

```cpp
#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

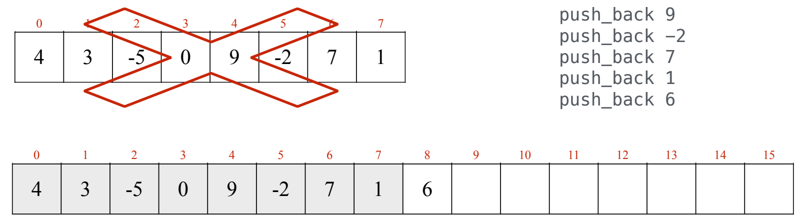
Designing a dynamic array class in C++

```cpp
#include <cstddef>
// ---------------------------------------------------------------------------
// This simplified implementation uses int. A production-quality dynamic array
// would use C++ templates to support arbitrary types, as std::vector<T> does.
// It also would implement the Rule of Three/Five for proper memory management.
// ---------------------------------------------------------------------------
class DynamicArray {
    private:
        int *m_arr;                          // pointer to the (internal) array
        size_t m_capacity;                   // total number of elements that can be stored
        size_t m_size;                       // number of elements currently stored

    public:
        DynamicArray();                      // constructor (allocates memory)
        ~DynamicArray();                     // destructor (frees memory)
        void push_back(int val);             // add an element to the end
        void pop_back();                     // remove the last element
        void insert(int val, size_t idx);    // insert an element at a specific index
        void erase(size_t idx);              // remove an element at a specific index
        void resize(size_t len);             // change the capacity of the array
        size_t size() const;                 // return the number of elements
        size_t capacity() const;             // return the capacity
        bool empty() const;                  // 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




```
push_back 9
push_back −2
push_back 7
push_back 1
push_back 6
```

- 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

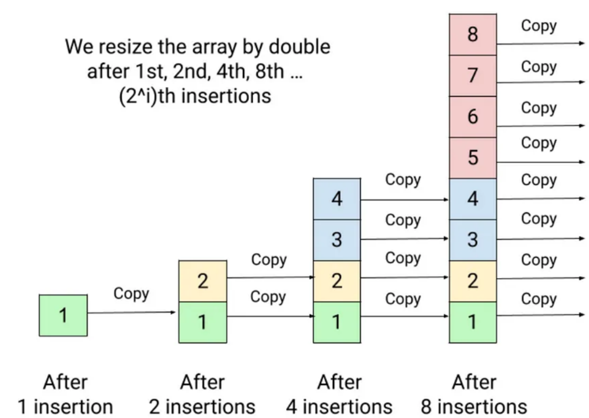
- When array is full, grow capacity to: `capacity + 1`
  - starting from an empty array, count number of array accesses (**reads and writes**) for appending n elements (ignore cost of allocating/deallocating memory)

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 cost per insertion is $\Theta(n)$ — inefficient.

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

$$T(n) = n + 2\sum_{i=0}^{\log_2 n - 1} 2^i$$

assume n is a power of 2

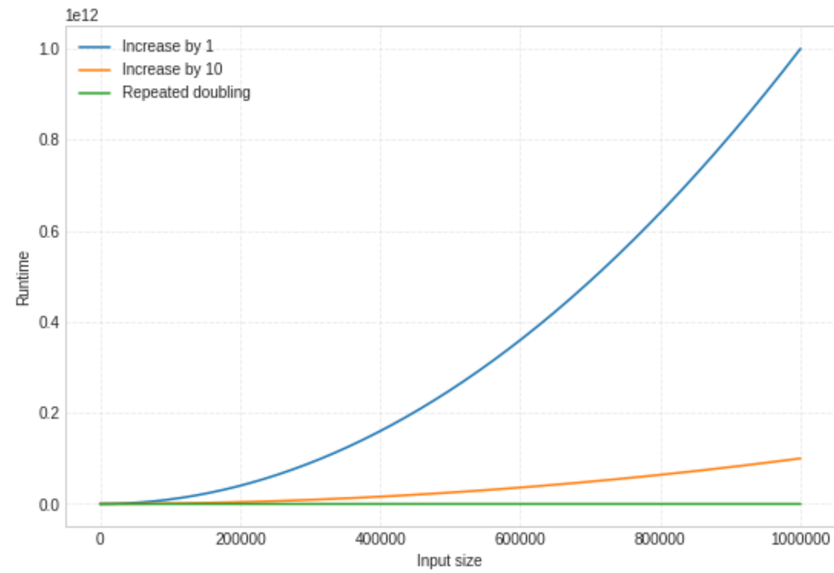
$$= n + 2\left(\frac{2^{\log_2 n} - 1}{2 - 1}\right)$$

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

$$= \Theta(n)$$

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`)
  - ✓ implementation-defined (not in the C++ standard)
  - ✓ grow by 1.5 (MS Visual C++) or 2.0 (g++/clang)
- Java (`ArrayList`)
  - ✓ grow by 1.5 in OpenJDK
- Python (`List`)
  - ✓ grow by ~1.125
- Rust (`std::vec::Vec`)
  - ✓ typically grow by 2

Factor	Memory overhead	Copy frequency
Large (e.g., 2)	Up to 50% wasted	Fewer resizes
Small (e.g., 1.25)	Less waste	More resizes

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			

Rule of three/five

- Shallow Copies
  - ✓ default **copy operations** copy pointer values, not the data they point to, resulting in two objects sharing the same heap memory
  - ✓ when one object is destroyed, the other holds a dangling pointer, leading to double-free errors and undefined behavior
- Rule of Three
  - ✓ if a class requires a custom implementation of any one of these, it likely requires all three:
    - destructor: releases dynamically allocated resources
    - copy constructor: creates a new object as a deep copy of another
    - copy assignment operator: replaces contents with a deep copy of another
- Rule of Five
  - ✓ extends Rule of Three with move semantics for efficiency:
    - move constructor: transfers ownership from a temporary (rvalue)
    - move assignment operator: transfers ownership, replacing current contents
  - ✓ move operations "steal" resources instead of copying, leaving source in a valid but unspecified state