CSC 212: Data Structures and Abstractions

Dynamic Arrays

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

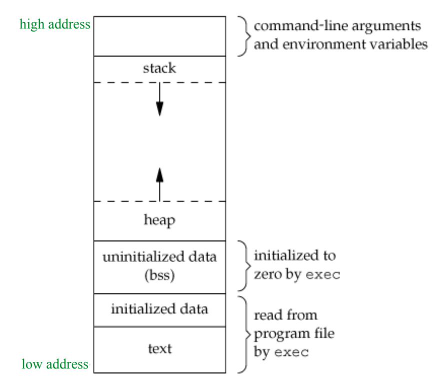
C/C++ memory model

Memory model

- What is the C/C++ memory model?
  - a formal specification that defines how programs interact with memory at a high level, ensuring safe and predictable behavior
    - implementation details are delegated to the compiler and CPU architecture
  - the memory model establishes a "contract" between programmer and compiler
- Memory Layout
  - memory is divided into multiple segments
  - each segment serves a specific purpose and has different properties

Memory layout

- Text Segment (code)
  - contains **instructions** generated by the compiler
  - marked as read-only to prevent accidental modification
- Data Segment (global/static variables)
  - contains multiple subsections (e.g. initialized data, uninitialized data, constant data)
  - size determined at compilation, addresses resolved during linking
- Heap
  - dedicated to dynamic memory allocation
  - requires explicit management by the programmer
- Stack (function calls, local variables)
  - implements last-in-first-out (LIFO) for function calls and local variables
  - no explicit deallocation required



```
#include <iostream>

// global variable
float pi = 3.1416;
// constant global variable
const int min = 100;
// uninitialized global variable
int sum;

void foo(int arg) {
    // local variable
    int i = 1;
    std::cout << "arg:\t" << &arg << std::endl;
    std::cout << "i:\t" << &i << std::endl;
}

int main() {
    // heap variable
    int *array = new int[10];

    std::cout << "pi:\t" << &pi << std::endl;
    std::cout << "min:\t" << &min << std::endl;
    std::cout << "sum:\t" << &sum << std::endl;
    std::cout << "array:\t" << array << std::endl;
    std::cout << "main:\t" << (void*) &main << std::endl;
    std::cout << "foo:\t" << (void*) &foo << std::endl;
    foo(5);

    delete [] array;

    return 0;
}
```

```
./prog | sort -k 2
```

```
foo:    0x102802f14
main:   0x102803064
min:    0x102803f04
pi:     0x102808000
sum:    0x102808004
array:  0x14c605e00
i:      0x16d5ff3b8
arg:    0x16d5ff3bc
```

leading zeros are ignored (64-bit addresses)

Dynamic arrays

C-style arrays

- <u>Contiguous</u> sequence of elements of <u>identical type</u>
  - 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

Dynamic arrays

- Limitations of C-style arrays
  - size must be known at compile time or use dynamic memory allocation — once created the array size does not change
  - provide $\Theta(1)$ read/write cost, but inflexible
- Dynamic arrays
  - can **<u>grow or shrink in size</u>** during runtime
    - 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.

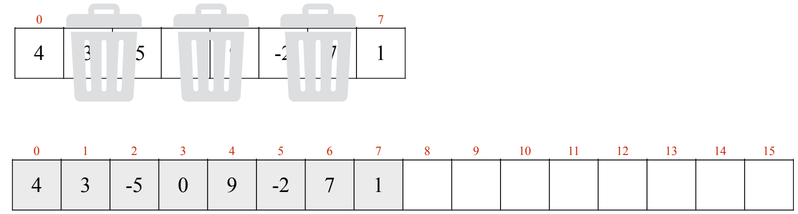
Dynamic array class in C++

```
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, new capacity: `current + 1`
  - starting from an empty array, count number of array accesses **(reads and writes)** for adding n elements (ignore cost of allocating/deallocating memory)

n	copy	append
1	2 x 0	1
2	2 x 1	1
3	2 x 2	1
4		
5		
6		
n-1		
n		

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

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

$$= \Theta(n^2)$$

cost of adding n elements

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.

Grow by factor

- When array is full, new capacity: `current * factor`
  - called repeated doubling, when `factor == 2`
  - starting from an empty array, count number of array accesses **(reads and writes)** for adding n elements — assume n is a power of 2 (ignore cost of allocating/deallocating memory)

n	copy	append
1	0	1
2	2 * 1	1
3	2 * 2	1
4	0	1
5	2 * 4	1
6	0	1
7	0	1
8	0	1
9	2 * 8	1
10	0	1
n-1		
n		

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

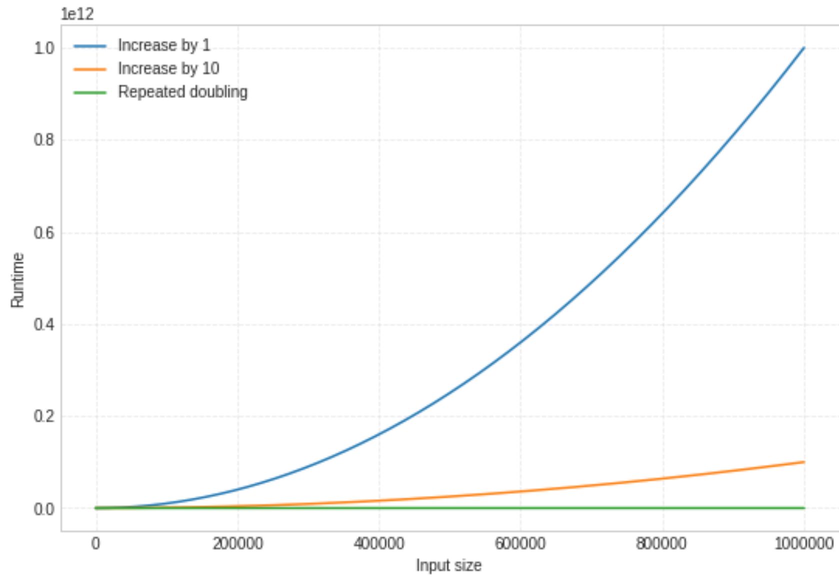
$$= n + 2\left(2^{\log n} - 1\right)$$

$$= n + 2n - 2$$

$$= \Theta(n)$$

cost of adding n elements

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.



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 times the current capacity
  - 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

`std::vector`

`std::vector is a sequence container that encapsulates dynamic arrays`

Quick Start

`#include <vector>`	
`std::vector<int> v {2, 4, 5};`	2 4 5
`v.push_back(6);`	2 4 5 6
`v.pop_back();`	2 4 5
`v[1] = 3;`	2 3 5
`v.resize(5, 0);`	2 3 5 0 0
`cout << v[2];`	prints 5
`for (int x : v)`	
`  cout << x << ' '`	prints 2 3 5 0 0

https://hackingcpp.com/cpp/std/vector.html

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			