CSC 212: Data Structures and Abstractions

Dynamic Arrays

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

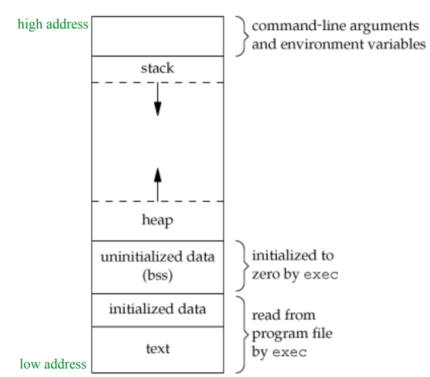
C/C++ memory model

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 << "address of arg\t" << &arg << std::endl;
    std::cout << "address of i\t" << &i << std::endl;
}

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

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

    delete [] A;

    return 0;
}
```

```
./prog | sort -k 4
```

```
address of foo  0x105499fc0
address of main 0x10549a0f0
address of min  0x10549af48
address of pi   0x10549c000
address of sum  0x10549c004
value of A      0x7fd29f705e90
address of i    0x7ff7baa66438
address of arg  0x7ff7baa6643c
address of A    0x7ff7baa66460
```

NOTE: leading zeros are ignored (64-bit addresses)

Can you tell what are the memory locations grouped by different colors?

What happens if you run the program multiple times?

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

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 **grow or shrink in size** 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.

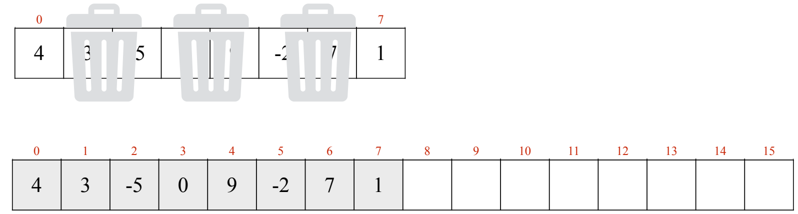
Dynamic array class in C++

```
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		

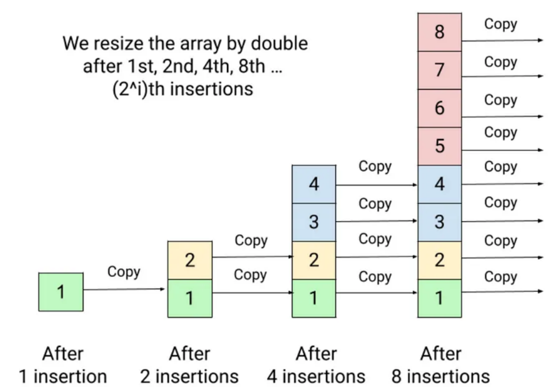
$$T(n) = n + \sum_{i=0}^{n-1} 2i$$

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

$$= \Theta(n^2)$$ ← 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.

Repeated doubling



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

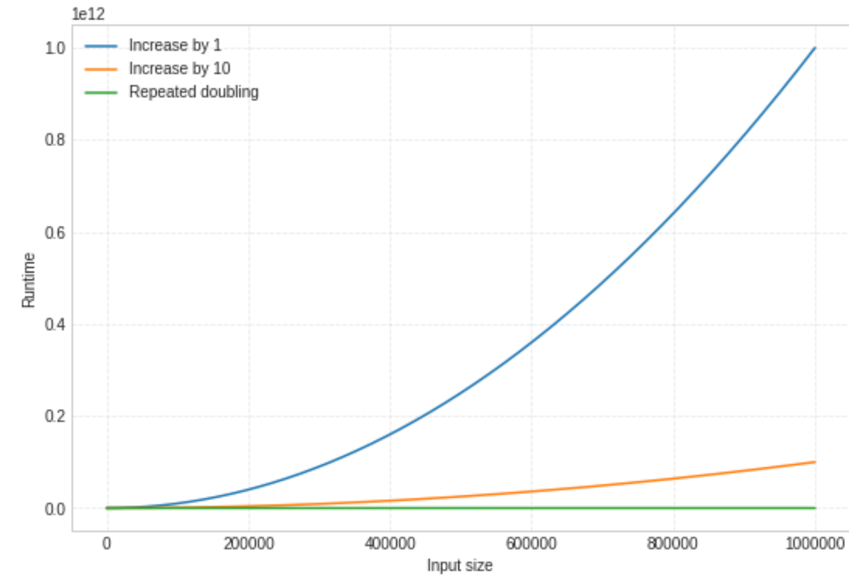
Grow by factor

- 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		

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

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.



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

Live coding …

- Complete the implementation of the `DynamicArray` class:

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

    public:
        DynamicArray(float growth_rate);            // constructor
        ~DynamicArray();                            // destructor
        void push_back(int val);                    // add an element to the end
        void pop_back();                            // remove the last element
        const int& operator[](size_t idx) const;    // read-only access at a specific index
        int& operator[](size_t idx);                // access at a specific index (can modify)
        void insert(int val, size_t idx);           // insert an element at a specific index
        void erase(int uidx);                       // remove an element at a specific index
        void resize(int ulen);                      // change the capacity of the array
        size_t get_size();                          // return the number of elements
        size_t get_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
};
```

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

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

Information taken from `claude.ai` (to be confirmed)

bottom line: growth factors range from ~1.2 to ~2 depending on language used

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			

Dynamic arrays in the STL (C++)

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;                                                   (1)
```

```
namespace pmr {
    template< class T >
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;   (2) (since C++17)
}
```

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator.

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

The storage of the vector is handled automatically, being expanded as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using `capacity()` function. Extra memory can be returned to the system via a call to `shrink_to_fit()`[1].

Reallocations are usually costly operations in terms of performance. The `reserve()` function can be used to eliminate reallocations if the number of elements is known beforehand.

The complexity (efficiency) of common operations on vectors is as follows:

- Random access - constant $\mathcal{O}(1)$.
- Insertion or removal of elements at the end - amortized constant $\mathcal{O}(1)$.
- Insertion or removal of elements - linear in the distance to the end of the vector $\mathcal{O}(n)$.

`std::vector` (for T other than `bool`) meets the requirements of *Container*, *AllocatorAwareContainer* (since C++11), *SequenceContainer*, *ContiguousContainer* (since C++17) and *ReversibleContainer*.

Member functions

(constructor)	constructs the `vector` (public member function)
(destructor)	destructs the `vector` (public member function)
operator=	assigns values to the container (public member function)
assign	assigns values to the container (public member function)
assign_range (C++23)	assigns a range of values to the container (public member function)
get_allocator	returns the associated allocator (public member function)
Element access	
at	access specified element with bounds checking (public member function)
operator[]	access specified element (public member function)
front	access the first element (public member function)
back	access the last element (public member function)
data	direct access to the underlying contiguous storage (public member function)
Iterators	
begin **cbegin** (C++11)	returns an iterator to the beginning (public member function)
end **cend** (C++11)	returns an iterator to the end (public member function)
rbegin **crbegin** (C++11)	returns a reverse iterator to the beginning (public member function)
rend **crend** (C++11)	returns a reverse iterator to the end (public member function)

Capacity

empty	checks whether the container is empty (public member function)
size	returns the number of elements (public member function)
max_size	returns the maximum possible number of elements (public member function)
reserve	reserves storage (public member function)
capacity	returns the number of elements that can be held in currently allocated storage (public member function)
shrink_to_fit (DR*)	reduces memory usage by freeing unused memory (public member function)

Modifiers

clear	clears the contents (public member function)
insert	inserts elements (public member function)
insert_range (C++23)	inserts a range of elements (public member function)
emplace (C++11)	constructs element in-place (public member function)
erase	erases elements (public member function)
push_back	adds an element to the end (public member function)
emplace_back (C++11)	constructs an element in-place at the end (public member function)
append_range (C++23)	adds a range of elements to the end (public member function)
pop_back	removes the last element (public member function)
resize	changes the number of elements stored (public member function)
swap	swaps the contents (public member function)

std::vector

```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';
}
```