# Inside-Out: STL

## How to use it wisely?

Jadwiga Pokorska

TietoEvry

26.02.2020

# Presentation plan

1. Introduction
   - Time complexity
   - Vector
   - Sort?
2. Balanced trees
   - BST
   - AVL / Red-Black Tree
3. Hash tables
   - Hash table of integers

# Time complexity

## Purpose

Time complexity is a tool to measure the efficiency of our algorithm.

# Time complexity

## Purpose

Time complexity is a tool to measure the efficiency of our algorithm.

Usually defined with *big-O* notation:

- $O(N)$,
- $O(N^2)$,
- $O(\log N)$,
- $O(N \cdot \log N)$,
- $O(\sqrt{N})$,

# Task: find not paired item

**Task**

Given an array of integers, find the only number that does **not** have a pair.

# Task: find not paired item

## Task

Given an array of integers, find the only number that does **not** have a pair.

## Example

For array $[2, 3, 7, 7, 2, 3, 2]$ the answer is 2.

# Task: find not paired item

```
int f(const vector<int>& t) {
    for (int selected_item : t) {
        int cnt = 0;
        for (int item : t)
            if (item == selected_item)
                cnt++;
        if (cnt % 2 == 1)
            return selected_item;
    }
    throw "All items have pairs!";
}
```

What is the time complexity?

# Task: find not paired item

```
int f(const vector<int>& t) {
    for (int selected_item : t) {
        int cnt = 0;
        for (int item : t)
            if (item == selected_item)
                cnt++;
        if (cnt % 2 == 1)
            return selected_item;
    }
    throw "All items have pairs!";
}
```

What is the time complexity? $O(N^2)$

# Task: find not paired item

```
int f(const vector<int>& t) {
    std::sort(t.begin(), t.end());
    int cnt = 0, prev = -1;
    for (int i = 0; i < t.size(); ++i) {
        if (prev == t[i]) cnt++;
        else {
            if (cnt % 2 == 1) return prev;
            prev = t[i];
            cnt = 0;
        }
    }
    if (cnt % 2 == 1) return prev;
    throw "All items have pairs!";
}
```

What is the time complexity?

# Task: find not paired item

```
int f(const vector<int>& t) {
    std::sort(t.begin(), t.end());
    int cnt = 0, prev = -1;
    for (int i = 0; i < t.size(); ++i) {
        if (prev == t[i]) cnt++;
        else {
            if (cnt % 2 == 1) return prev;
            prev = t[i];
            cnt = 0;
        }
    }
    if (cnt % 2 == 1) return prev;
    throw "All items have pairs!";
}
```

What is the time complexity? $O(N \cdot \log N)$

# Task: find not paired item

```
int f(const vector<int>& t) {
    int result = 0;
    for (int item : t)
        result ^= item;
    if (result > 0)
        return result;
    throw "All items have pairs!";
}
```

What is the time complexity?

# Task: find not paired item

```
int f(const vector<int>& t) {
    int result = 0;
    for (int item : t)
        result ^= item;
    if (result > 0)
        return result;
    throw "All items have pairs!";
}
```

What is the time complexity? $O(N)$

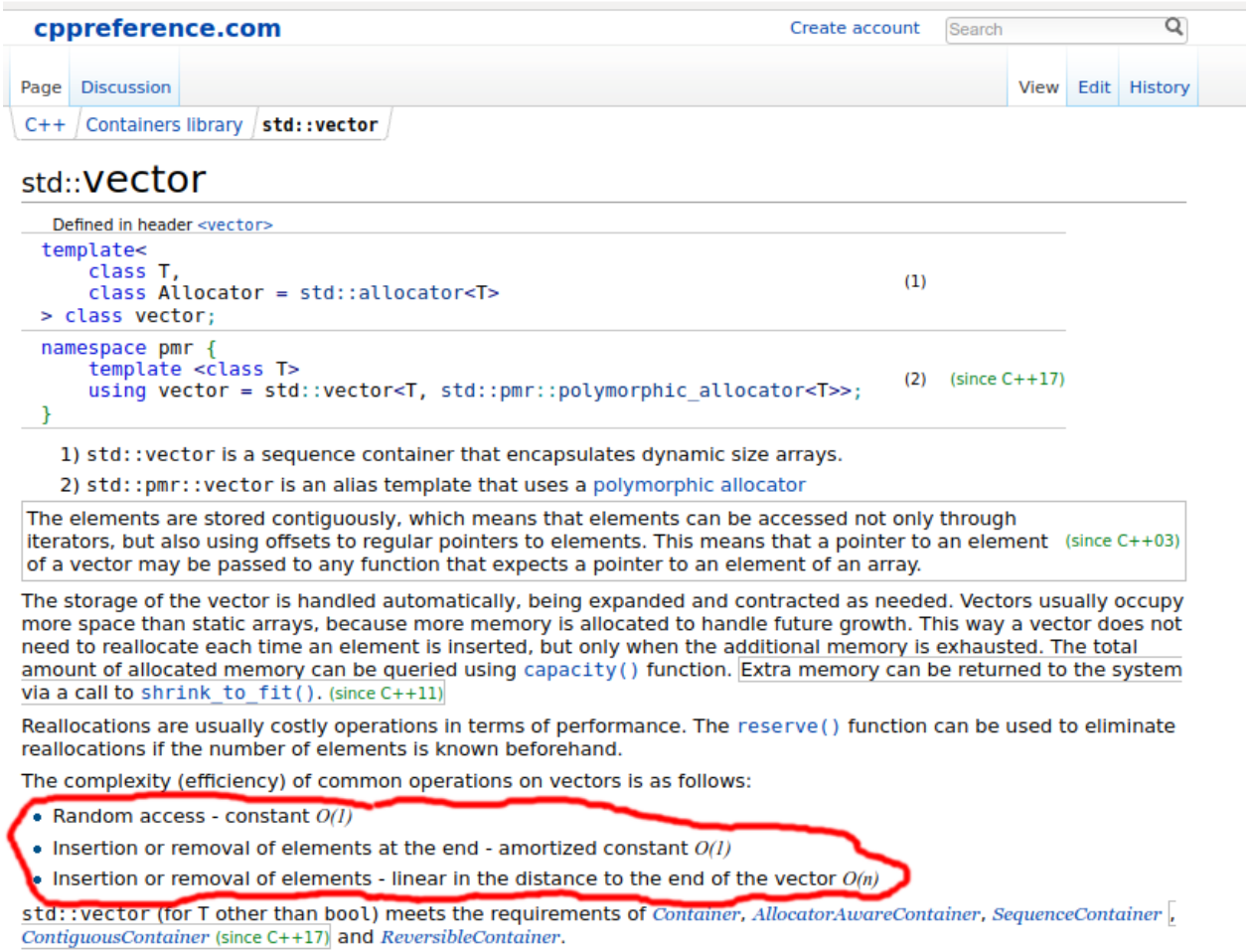
# Vector - time complexity

Where do I find the information about the time complexity?

# Vector - time complexity

Where do I find the information about the time complexity?

Documentation!



cppreference.com Create account Search

Page Discussion View Edit History

C++ Containers library std::vector

## 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. (since C++03)

The storage of the vector is handled automatically, being expanded and contracted 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(). (since C++11)

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 *O(1)*
- Insertion or removal of elements at the end - amortized constant *O(1)*
- Insertion or removal of elements - linear in the distance to the end of the vector *O(n)*

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

# Vector - time complexity



## std::vector<T,Allocator>::erase

```
iterator erase( iterator pos );                                  (1)  (until C++11)
iterator erase( const_iterator pos );                                 (since C++11)
iterator erase( iterator first, iterator last );                 (2)  (until C++11)
iterator erase( const_iterator first, const_iterator last );          (since C++11)
```

Erases the specified elements from the container.

1) Removes the element at pos.
2) Removes the elements in the range `[first, last)`.

Invalidates iterators and references at or after the point of the erase, including the `end()` iterator.

The iterator pos must be valid and dereferenceable. Thus the `end()` iterator (which is valid, but is not dereferencable) cannot be used as a value for pos.

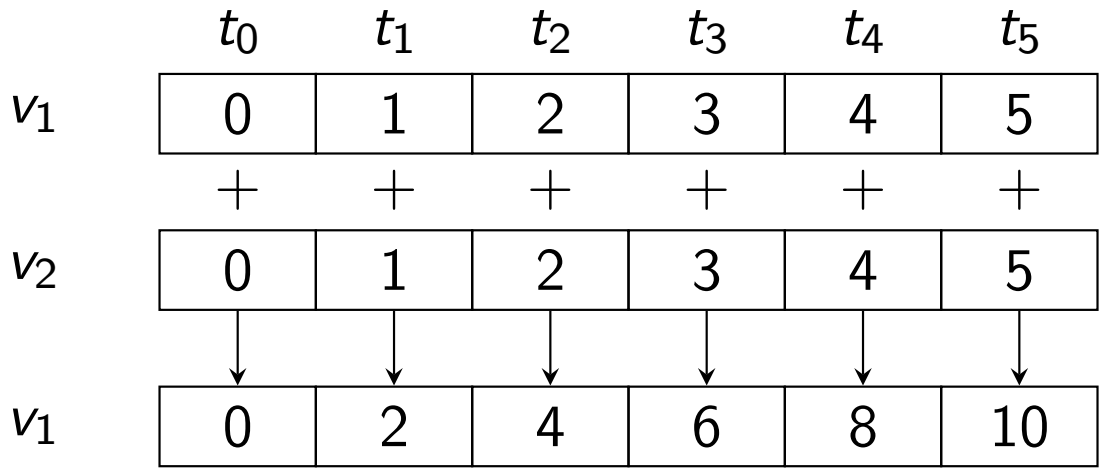
The iterator `first` does not need to be dereferenceable if `first==last`: erasing an empty range is a no-op.

### Complexity

Linear: the number of calls to the destructor of T is the same as the number of elements erased, the assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements

# Vector - internal implementation

TODO: make piktures showing inserting elements (with resizing and copying).



TODO.

TODO.

What is it?