# STL Algorithms

## Contents

## 1. Why we need to know STL Algorithms?

- STL algorithms can make code more expressive.
- Raising levels of abstraction. Which means, they allow us to express **what we want to do** as supposed to **how we want to do**.
- Can be sometimes spectacular.
- Avoid common mistakes
    - Not checking out of bound.
    - Empty loops.
    - Complexity.
- Used by lots of people.
- A common vocabulary.
- Works on all the versions of compiler.

## 2. Heaps

- We use heap data structure to get the max or min value of a collection in constant time.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


int main() {

    vector<int> numbers = {20, 30, 40, 25, 15};

    // To build a heap.
    make_heap(begin(numbers),end(numbers));

    // We use heap to get the max or min value of a collection in constant time.
    cout << "Max Element: " << numbers[0] << endl;

    // To add an element to heap.
    //    We add the element to the end.
    //    Then we reorder the elements.
    numbers.push_back(99);
    push_heap(begin(numbers),end(numbers));
    cout << "Max Element: " << numbers[0] << endl;

    // To delete the max element form heap.
    //    We swap the max element with the end.
    //    Rearrange the heap.
    //    Then we delete the last element.
    pop_heap(begin(numbers),end(numbers));
```

```
    numbers.pop_back();

    // To sort the elements...
    sort_heap(begin(numbers),end(numbers));
    cout << "The heap elements after sorting are:";
    for (const int &x : numbers)
       cout << x << " ";

    return 0;
}
```

## 3. Sorting

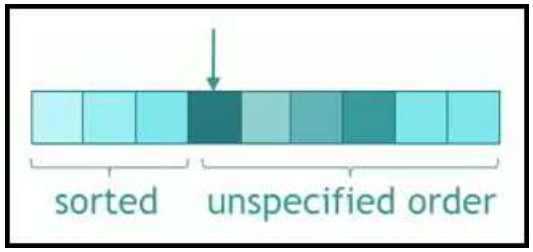
### a. std::sort

- It takes 3 parameters,
  - `begin()`
  - `end()`
  - Optional `Comparator`.
- By default, the `sort()` function sorts the elements in ascending order.
- Time Complexity: `O(N log N)`
- Space Complexity: It may use `O(log N)`

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  vector<int> numbers = {20, 30, 40, 25, 15};
                                          // Sort Descending!!!
  std::sort(begin(numbers), end(numbers) /*, greater<int>() */);

  cout << "The elements after sorting are: ";
    for (const int &x : numbers)
       cout << x << " ";

  return 0;
}
// The elements after sorting are: 15 20 25 30 40
```

### b. std::partial_sort

- It takes 4 parameters,
  - `begin()`
  - `middle` (Example: `begin() + 4`)
  - `end()`
  - Optional `comparator`.
- It rearranges the elements in the range `[first, last)`, in such a way that the elements before middle are sorted in **ascending** order, whereas the elements after middle are left without any specific order.



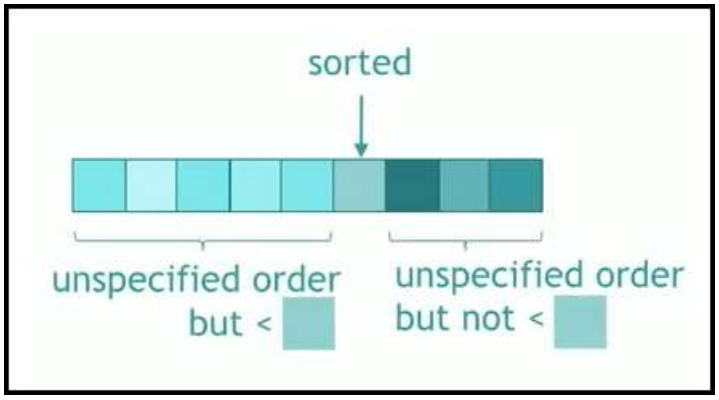

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  vector<int> numbers = {4, 2, 8, 1, 3, 6, 0, 9, 7};
              // Begin         Middle
  partial_sort(begin(numbers), begin(numbers) + 4,
              // End       Comparator
              end(numbers)/*, greater<int>() */);

  cout << "The elements after partial sorting are: ";
    for (const int &x : numbers)
       cout << x << " ";

  return 0;
}
// The elements after partial sorting are: 0 1 2 3 8 6 4 9 7
```

- To find the largest element.

```
std::partial_sort(begin(numbers), begin(numbers) + 1,
                  end(numbers), greater<int>());
// Will print the largest number in the collection.
cout << numbers[0] << endl;
```

### c. std::nth_element

- This rearranges the list in such a way such that the element at the $n^{th}$ position is the one which should be at that position if we sort the list.
- It does not sort the list, just that all the elements, which precede the $n^{th}$ element are not greater than it, and all the elements which succeed it are not less than it.




- It takes 4 parameters,
  - `begin()`
  - $n^{th}$ (Example: `begin() + 4`)
  - `end()`
  - Optional `comparator`.

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> numbers = {4, 2, 8, 1, 3, 6, 0, 999, 7};

  auto m = numbers.begin() + 7;
  std::nth_element(begin(numbers), m, end(numbers));

  cout << "The elements after nth_element sorting are: ";
  for (const int &x : numbers)
    cout << x << " ";

  return 0;
}
// The elements after nth_element sorting are: 3 2 0 1 7 6 4 8 999
```

- Where can we apply `std::nth_element()`?
  - It can be used if we want to find the first n smallest numbers, but they may or may not be ordered.

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int v[8] = { 3, 7, 8, 4, 6, 5, 2, 1 }, i;
  cout << "The array is: ";
  for (i = 0; i < 8; i++)
    cout << v[i] << " ";
  cout << '\n';
  std::nth_element(v, v + 4, v + 8);
  cout << "The 4 smallest elements of array are: ";
  for (i = 0; i < 4; i++)
    cout << v[i] << " ";

  return 0;
}
// The array is: 3 7 8 4 6 5 2 1
// The 4 smallest elements of array are: 4 1 2 3
```

  - We can also find first n largest numbers, by just changing the Binary Function passed as argument in `std::nth_element`.

```
#include <algorithm>
#include <iostream>
using namespace std;
int main() {
  int v[8] = { 3, 7, 8, 4, 6, 5, 2, 1 }, i;
  cout << "The array is: ";
  for (i = 0; i < 8; i++)
    cout << v[i] << " ";
  cout << '\n';
  std::nth_element(v, v + 4, v + 8, greater<int>());
  cout << "The 4 largest elements of array are: ";
  for (i = 0; i < 4; i++)
    cout << v[i] << " ";
  return 0;
}
// The array is: 3 7 8 4 6 5 2 1
// The 4 largest elements of array are: 6 7 8 5
```

### d. std::sort_heap

- We have seen this while discussing the 'Heaps'.

### e. std::inplace_merge

- Merges two **consecutive sorted ranges**: `[first, middle)` and `[middle, last)`, putting the result into the combined sorted range `[first, last)`.

```
#include <iostream>     // std::cout
#include <algorithm>    // std::inplace_merge, std::sort, std::copy
#include <vector>       // std::vector

int main() {
  int first[] = { 5,10,15,20,25 };    // Unsorted Array!!!
  int second[] = { 50,40,30,20,10 };  // Unsorted Array!!!
  std::vector<int> v(10);             // To store the sorted vector.
  std::vector<int>::iterator it;

  std::sort(first, first + 5);        // Sort the first array.
  std::sort(second, second + 5);      // Sort the second array.

  it = std::copy(first, first + 5, v.begin());  // Copy the first array to vector.
  std::copy(second, second + 5, it);            // Copy the second array to vector.

  // Sort the merged array. The first half is sorted.
  // The second half is also sorted.
  std::inplace_merge(v.begin(), v.begin() + 5, v.end());

  std::cout << "The resulting vector contains:";
  for (it = v.begin(); it != v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
// The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
```

## 4. Partition

- Partition refers to act of dividing elements of containers depending upon a given condition.
- Partition operations:
    1. `partition(begin, end, condition)`
    2. `is_partitioned(begin, end, condition)`
    3. `stable_partition(begin, end, condition)`
    4. `partition_point(begin, end, condition)`

### a. std::partition

- This function is used to partition the elements on basis of condition mentioned in its arguments.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
  vector<int> n = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};

  cout << "Elements of the vector are before PARTITION: ";
  for(auto& i: n) cout << i << " ";
  cout << endl;
  // Partition the array as even and odd numbers.
  partition(begin(n), end(n), [](int x){ return ((x & 1) == 0); });

  cout << "Elements of the vector are after PARTITION: ";
  for(auto& i: n) cout << i << " ";
  return 0;
}
// Elements of the vector are before PARTITION: 9 2 3 8 1 7 5 6 4 0
// Elements of the vector are after PARTITION: 0 2 4 8 6 7 5 1 3 9
```

### b. std::is_partitioned

- This function returns boolean `true` if container is partitioned else returns `false`.

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
int main() {
  vector<int> n = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  // Partition the array as even and odd numbers.
  partition(begin(n), end(n), [](int x) { return ((x & 1) == 0); });
  bool isPartitioned =
      is_partitioned(begin(n), end(n), [](int x) { return ((x & 1) == 0); });
```

```
  if (isPartitioned)
    cout << "The array is partitioned as even and odd numbers!!!\n";
  else
    cout << "The array is not partitioned!!!\n";

  return 0;
}
// The array is partitioned as even and odd numbers!!!
```

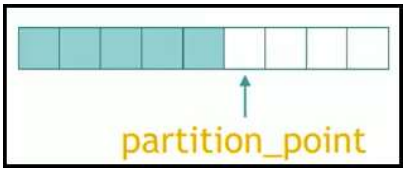
### c. std::stable_partition

- This function is used to partition the elements on basis of condition mentioned in its arguments in such a way that the relative order of the elements is preserved.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
  vector<int> n = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};

  cout << "Elements of the vector are before STABLE PARTITION: ";
  for(auto& i: n) cout << i << " ";
  cout << endl;
  // Partition the array as even and odd numbers.
  stable_partition(begin(n), end(n), [](int x){ return ((x & 1) == 0); });

  cout << "Elements of the vector are after STABLE PARTITION: ";
  for(auto& i: n) cout << i << " ";
  return 0;
}
// Elements of the vector are before STABLE PARTITION: 9 2 3 8 1 7 5 6 4 0
// Elements of the vector are after STABLE PARTITION: 2 8 6 4 0 9 3 1 7 5
```

### d. std::partition_point

- This function returns an iterator pointing to the partition point of container i.e. the first element in the partitioned range [beg, end) for which condition is not true.
- The container should already be partitioned for this function to work.



```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
  vector<int> n = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};

  cout << "Elements of the vector are before PARTITION: ";
  for(auto& i: n) cout << i << " ";
  cout << endl;
  // Partition the array as even and odd numbers.
  partition(begin(n), end(n), [](int x){ return ((x & 1) == 0); });

  // Get the PARTITION point.
  auto it = partition_point(begin(n), end(n), [](int x){  return ((x & 1) == 0); });
  std::vector<int> odd, even;

  // Copy even numbers to 'even' vector.
  even.assign(begin(n), it);
  // Copy odd numbers to 'odd' vector.
  odd.assign(it, end(n));

  cout << "Even elements of the vector are: ";
  for(auto& i: even) cout << i << " ";
  cout << '\n';
  cout << "Odd elements of the vector are: ";
  for(auto& i: odd) cout << i << " ";
  cout << '\n';
  return 0;
}
// Elements of the vector are before PARTITION: 9 2 3 8 1 7 5 6 4 0
// Even elements of the vector are: 0 2 4 8 6
// Odd elements of the vector are: 7 5 1 3 9
```

## 5. Permutations Related

### a. std::rotate

- Rotates the order of the elements in the range `[first, last)`, in such a way that the element pointed by middle becomes the new first element.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
  std::vector<int> myvector(9); // Size is 9.

  cout << "Original Vector   : ";
  // Fill some values!
  for (size_t i{0}; i < myvector.size(); i++) {
    myvector[i] = i+1;
    cout << myvector[i] << " ";
  }

  cout << "\nRotated LEFT Once : ";
  std::rotate(myvector.begin(),myvector.begin()+1,myvector.end());
  for(const auto& i: myvector)
    cout << i << " ";

  cout << "\nRotated RIGHT Once: ";
  std::rotate(myvector.begin(),myvector.end()-1,myvector.end());
  for(const auto& i: myvector)
    cout << i << " ";

  return 0;
}
// Original Vector   : 1 2 3 4 5 6 7 8 9
// Rotated LEFT Once : 2 3 4 5 6 7 8 9 1
// Rotated RIGHT Once: 1 2 3 4 5 6 7 8 9
```

### b. std::shuffle

- It randomly re-arranges elements in range [first, last).
- It takes 3 parameters…
  - `begin()`
  - `end()`
  - The function object that generates random number.
- The function swaps the value of each element with some other randomly picked element. When provided, the function gen determines which element is picked in every case. Otherwise, the function uses some unspecified source of randomness.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <random>
using namespace std;
int main () {
  std::vector<int> myvector(9); // Size is 9.

  cout << "Original Vector : ";
  // Fill some values!
  for (size_t i{0}; i < myvector.size(); i++) {
    myvector[i] = i+1;
    cout << myvector[i] << " ";
  }

  cout << "\nShuffled        : ";
  auto rng = std::default_random_engine{};
  std::shuffle(std::begin(myvector), std::end(myvector), rng);

  for(const auto& i: myvector)
    cout << i << " ";

  cout << "\nOriginal Vector : ";
  // Fill values again!
  for (size_t i{0}; i < myvector.size(); i++) {
    myvector[i] = i+1;
    cout << myvector[i] << " ";
  }
  std::random_shuffle(std::begin(myvector), std::end(myvector));
  cout << "\nRandom Shuffled : ";
  for(const auto& i: myvector)
    cout << i << " ";
  return 0;
}
// Original Vector : 1 2 3 4 5 6 7 8 9
// Shuffled        : 4 1 5 8 6 2 9 7 3
// Original Vector : 1 2 3 4 5 6 7 8 9
// Random Shuffled : 9 2 7 3 1 6 8 4 5
```

- What is the difference between shuffle and `random_shuffle`?
  - `std::random_shuffle` uses `std::rand()` function to randomize the items, while the `std::shuffle` uses URBG (UniformRandomBitGenerator) which is a better random generator.

### c. std::next_permutation

- Rearranges the elements in the range `[first, last)` into the next lexicographically greater permutation.

### d. std::prev_permutation

- It is used to rearranges the elements in the range `[first, last)` into the previous lexicographically ordered permutation.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;
int main () {
  std::vector<int> myvector = { 1, 2, 3, 4};

  cout << "Original Vector    : ";
  for(const auto& i: myvector)
    cout << i << " ";

  cout << "\nNext Greatest      : ";
  next_permutation(begin(myvector), end(myvector));
  for(const auto& i: myvector)
    cout << i << " ";

  cout << "\nPrevious Greatest  : ";
  prev_permutation(begin(myvector), end(myvector));
  for(const auto& i: myvector)
    cout << i << " ";
  return 0;
}
// Original Vector    : 1 2 3 4
// Next Greatest      : 1 2 4 3
// Previous Greatest  : 1 2 3 4
```

### e. std::reverse

- Reverses the order of the elements in the range `[first, last)`.

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;
int main () {
  std::vector<int> myvector = { 1, 2, 3, 4};
```

```
  cout << "Original Vector    : ";
  for(const auto& i: myvector)
    cout << i << " ";

  cout << "\nReversed Vector    : ";
  reverse(begin(myvector), end(myvector));
  for(const auto& i: myvector)
    cout << i << " ";

  return 0;
}
// Original Vector    : 1 2 3 4
// Reversed Vector    : 4 3 2 1
```

## 6. Can Combine with Other Algorithms

### std::stable_*

- When 'stable' is tacked onto an algorithm, it does what an algorithm does but keeps the relative order.
- We have…
    1. `std::stable_sort`
    2. `std::stable_partition`

#### a. std::stable_sort

- `stable_sort()` is used to sort the elements in the range `[first, last)` in ascending order.
- It is like `std::sort`, but `stable_sort()` keeps the relative order of elements with equivalent values.
- When to Prefer `stable_sort` over `sort()`?
    - Sometimes we want to make sure that **order of equal elements is same in sorted array as it was in original array**.
    - Example:
        - Consider sorting students by marks, if two students have the same marks, we may want to keep them in the same order as they appear input.
        - Consider we have time intervals sorted by ending time and we want to sort by start time. Also, if two start times are same, we want to keep them sorted by the end time. This is not guaranteed by `sort()`.

```cpp
struct Interval { int start; int end; };

// Custom Comparator To Sort 'Interval' Objects.
// Sorting based on 'start' time.
bool compareStartTime(Interval a, Interval b) {
    return (a.start < b.start);
}
// Sorting based on 'end' time.
bool compareEndTime(Interval a, Interval b) {
    return (a.end < b.end);
}

int main () {

  vector<Interval> intervals = { { 1, 3 }, { 2, 4 }, { 1, 7 }, { 2, 19 },
                                 { 2, 7 }, { 3, 6 }, { 2, 8 }, { 5, 15 } };
  cout << "Original Vector: ";
  for(const auto& i: intervals){
    cout << "{" << i.start << ", " << i.end << "} ";
  }

  sort(intervals.begin(), intervals.end(), compareEndTime);
  cout << "\nstd::sort based on End Time: ";
  for(const auto& i: intervals){
    cout << "{" << i.start << ", " << i.end << "} ";
  }

  stable_sort(intervals.begin(), intervals.end(), compareStartTime);
  cout << "\nstd::stable_sort based on Start Time: ";
  for(const auto& i: intervals){
    cout << "{" << i.start << ", " << i.end << "} ";
  }

  return 0;
}
```

Order of equal elements is same in sorted array as it was in original array.

```
// Original Vector                    : {1, 3} {2, 4} {1, 7} {2, 19} {2, 7} {3, 6} {2, 8} {5, 15}
// std::sort based on End Time        : {1, 3} {2, 4} {3, 6} {1, 7} {2, 7} {2, 8} {5, 15} {2, 19}
// std::stable_sort based on Start Time: {1, 3} {1, 7} {2, 4} {2, 7} {2, 8} {2, 19} {3, 6} {5, 15}
```

### b. std::stable_partition

- We have seen this in 'Partition' section.

### std::is_*

- When '`is`' is tacked onto an algorithm, it checks for the predicate on the collection and returns a boolean.
- We have…
    1. `std::is_sorted`
    2. `std::is_partitioned`
    3. `std::is_heap`

### a. std::is_sorted

- Checks if the elements in range `[first, last)` are sorted in ascending order.
- It also takes 'comparator' as an optional parameter.

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main () {
  string s = "AbCdefGHIJklm";

  if(is_sorted(begin(s), end(s)))
    cout << "String " << s << " is sorted!" << endl;
  else
    cout << "String " << s << " is not sorted!" << endl;

  cout << "Checking if string is sorted or not using comparator..." << endl;

  if(is_sorted(begin(s), end(s), [](char a, char b) { return tolower(a) <= tolower(b); }))
    cout << "String " << s << " is sorted!" << endl;
  else
    cout << "String " << s << "  is not sorted!" << endl;

  return 0;
}
// String AbCdefGHIJklm is not sorted!
// Checking if string is sorted or not using comparator...
// String AbCdefGHIJklm is sorted!
```

### b. std::is_partitioned

- We have seen this in 'Partition' section.

### c. std::is_heap

- Checks if the elements in range `[first, last)` are a max heap.
- It returns `true` when given ranges of elements forms Max Heap, else it returns `false`.

```
int main () {
  vector<int> myArray = { 7, 8, 5, 4, 6, 2, 1, 9 };
  cout << "Original array: ";
  for(const auto& i : myArray)
    cout << i << " ";

  if(!is_heap(begin(myArray), end(myArray))){
    make_heap(begin(myArray), end(myArray));
    cout << "\nAfter making the heap: ";
    for(const auto& i : myArray)
      cout << i << " ";
  }
  return 0;
}
// Original array: 7 8 5 4 6 2 1 9
// After making the heap: 9 8 5 7 6 2 1 4
```

### d. std::is_permutation

- Returns `true` if the 2 collections contain the same elements but not necessarily in the same order.

```
int main(void) {
  vector V1{9, 3, 4, 7, 6, 8, 5, 1, 2};
  vector V2{8, 6, 7, 2, 5, 3, 4, 9, 1};

  if (is_permutation(begin(V1), end(V1), begin(V2))) {
    cout << "V2 is a permutation of V1.\n";
  } else {
    cout << "V2 is not a permutation of V1.\n";
  }
  return 0;
}
// V2 is a permutation of V1.
```

## std::is_*_until

- When '`is_*_until`' is tacked onto an algorithm, returns an iterator to the first position where the predicate doesn't hold anymore.
- We have…
    1. `std::is_sorted_until`
    2. `std::is_heap_until`

### a. std::is_sorted_until

- `is_sorted_until` is used to find out the first unsorted element in the range `[first, last)`.
- It returns an iterator to the first unsorted element in the range, so all the elements in between `[first, iterator)` are sorted.
- It can also be used to count the total no. of sorted elements in the range.

```
int main () {
  vector<int> myArray = { 1, 2, 3, 4, 7, 10, 8, 9, 5, 6 };
  cout << "Original array: ";
  for(const auto& i : myArray)
    cout << i << " ";

  auto itr = is_sorted_until(begin(myArray), end(myArray));

  cout << "\nThere are " << (itr - myArray.begin()) << " sorted elements.";
  cout << "\nSorted Elements in the original array are: ";
  for_each(begin(myArray), itr, [](int x) {cout << x << " ";});
  return 0;
}
// Original array: 1 2 3 4 7 10 8 9 5 6
// There are 6 sorted elements.
// Sorted Elements in the original array are: 1 2 3 4 7 10
```

### b. std::is_heap_until

- `is_heap_until` returns an iterator to the first element in the range `[first,last)` which is not in a valid position if the range is considered a heap.

```
int main() {
  std::vector<int> myArray{2, 6, 9, 3, 8, 4, 5, 1, 7};

  // Collection sorted in reverse order a valid heap.
  std::sort(myArray.begin(), myArray.end());
  std::reverse(myArray.begin(), myArray.end());

  auto last = is_heap_until(myArray.begin(), myArray.end());
  cout << "The " << (last - myArray.begin())
       << " first elements are a valid heap: ";
  for_each(begin(myArray), last, [](int i) { cout << i << " "; });

  std::cout << '\n';
  return 0;
}
// The 9 first elements are a valid heap: 9 8 7 6 5 4 3 2 1
```

### std::*_if

- When `'if'` is tacked onto an algorithm, it creates new algorithm that do the same thing but it take a predicate to generate the output.
- We have…
    - count_if
    - find_if
    - find_if_not
    - remove_if
    - remove_copy_if
    - replace_if
    - replace_copy_if
    - copy_if

#### a. std::count_if

- Returns number of elements in the range `[first, last)` for which the predicate returns `true`.

```
int main() {
  std::vector<int> myArray{2, 6, 1, 3, 1, 4, 5, 1, 7};

  cout<< "There are " <<
      count_if(begin(myArray), end(myArray),
      [](int i){return !(i & 1);}) << " even elements!\n";

  std::cout << '\n';

  return 0;
}
// There are 3 even elements!
```

#### b. std::find_if

- Returns an iterator to the first element in the range `[first, last)` that satisfies specific criteria.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  auto it = find_if(vec.begin(), vec.end(), [](int i){ return !(i&1);});
  cout << "\nFirst even number in the collection is: " << *it << "\n";
  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// First even number in the collection is: 2
```

### c. std::find_if_not

- Returns an iterator to the first element in the range `[first, last)` for which predicate returns `false`.
- If no such element is found, the function returns last.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  auto it = find_if_not(vec.begin(), vec.end(), [](int i){ return !(i&1);});
  cout << "\nFirst odd number in the collection is: " << *it << "\n";

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// First odd number in the collection is: 9
```

### d. std::remove_if

- Removes all elements from the range `[first, last)` for which the predicate returns true and returns a past-the-end iterator for the new end of the range.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  auto it = remove_if(vec.begin(), vec.end(), [](int i){ return !(i&1);});
  cout << "\nAfter removing even numbers: ";
  for_each(vec.begin(), it, [](int i){ cout << i << " ";});

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// After removing even numbers: 9 3 1 7 5
```

### e. std::remove_copy_if

- Copies elements from the range `[first, last)`, to another range beginning at `d_first`, omitting the elements which satisfy specific criteria.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  remove_copy_if(vec.begin(), vec.end(), dest.begin(), [](int i){ return !(i&1);});
  cout << "\nDestination Vector: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// Destination Vector: 9 3 1 7 5 0 0 0 0 0
```

### f. std::replace_if

- Assigns new_value to all the elements in the range `[first,last)` for which pred returns `true`.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  replace_if(vec.begin(), vec.end(), [](int i){ return !(i&1);}, -1);
  cout << "\nAfter replace_if the original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// After replace_if the original Vector: 9 -1 3 -1 1 7 5 -1 -1 -1
```

### g. std::replace_copy_if

- Copies the elements in the range [first,last) to the range beginning at result, replacing those for which pred returns true by new_value.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  replace_copy_if(vec.begin(), vec.end(), dest.begin(), [](int i){ return !(i&1);}, -1);
  cout << "\nDestination Vector: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// Destination Vector: 9 -1 3 -1 1 7 5 -1 -1 -1
```

### h. std::copy_if

- Copies the elements in the range [first,last) for which pred returns true to the range beginning at result.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  copy_if(vec.begin(), vec.end(), dest.begin(), [](int i){ return !(i&1);});
  cout << "\nDestination Vector: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// Destination Vector: 2 8 6 4 0 0 0 0 0 0
```

### std::*_copy

- When 'copy' is tacked onto an algorithm, it creates new algorithm that do the same thing but in a new collection.
- These algorithms leave the collection untouched. These algorithms take an output iterator and produce the output on this iterator.

#### a. remove_copy

- Copies elements from the range [first, last), to another range beginning at d_first, omitting the elements which satisfy specific criteria.

```
int main()
{
  vector vec {1,2,3,3,9,3,3,3,1,9,10};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  remove_copy(vec.begin(), vec.end(), dest.begin(), 3);
  cout << "\nDestination Vector after std::remove_copy: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Source Vector: 1 2 3 3 9 3 3 3 1 9 10
// Destination Vector after std::remove_copy: 1 2 9 1 9 10 0 0 0 0 0
```

#### b. unique_copy

- Copies the elements in the range [first, last) to the range beginning at result, except **consecutive duplicates**.

```
int main()
{
  vector vec {1,2,3,3,9,3,3,3,1,9,10};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  unique_copy(vec.begin(), vec.end(), dest.begin());
  cout << "\nDestination Vector after std::unique_copy: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Source Vector: 1 2 3 3 9 3 3 3 1 9 10
// Destination Vector after std::unique_copy: 1 2 3 9 3 1 9 10 0 0 0
```

### c. reverse_copy

- Copies the elements in the range `[first, last)` to the range beginning at result, but in reverse order.

```
int main()
{
  vector vec {1,2,3,3,9,3,3,3,1,9,10};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  reverse_copy(vec.begin(), vec.end(), dest.begin());
  cout << "\nDestination Vector after std::reverse_copy: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Source Vector: 1 2 3 3 9 3 3 3 1 9 10
// Destination Vector after std::reverse_copy: 10 9 1 3 3 3 9 3 3 2 1
```

### d. rotate_copy

- Copies the elements in the range `[first, last)` to the range beginning at result, but rotating the order of the elements in such a way that the element pointed by middle becomes the first element in the resulting range.

```
int main()
{
  vector vec {1,2,3,4,5,6,7,8,9,10};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  rotate_copy(vec.begin(), vec.begin() + 5, vec.end() ,dest.begin());
  cout << "\nDestination Vector after std::rotate_copy: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Source Vector: 1 2 3 4 5 6 7 8 9 10
// Destination Vector after std::rotate_copy: 6 7 8 9 10 1 2 3 4 5
```

### e. replace_copy

- It copies the elements in the range `[first, last)` to the range beginning at result, replacing the appearances of `old_value` by `new_value`.

```
int main()
{
  vector vec {1,2,3,4,1,1,1,1,9,10};
  vector<int> dest(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  replace_copy(vec.begin(), vec.end(), dest.begin(), 1, -1);
  cout << "\nDestination Vector after std::replace_copy: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Source Vector: 1 2 3 4 1 1 1 1 9 10
// Destination Vector after std::replace_copy: -1 2 3 4 -1 -1 -1 -1 9 10
```

### f. partition_copy

- Copies the elements in the range `[first, last)` for which pred returns true into the range pointed by `result_true`, and those for which it does not into the range pointed by `result_false`.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  vector<int> dest1(vec.size());
  vector<int> dest2(vec.size());
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  partition_copy(vec.begin(), vec.end(), dest1.begin(), dest2.begin(),
  [](int x){ return ((x & 1) == 0); });
  cout << "\nOutput of partition_copy: \n";
  cout << "True Array: ";
  for_each(dest1.begin(), dest1.end(), [](int i){ cout << i << " ";});
  cout << "\nFalse Array: ";
  for_each(dest2.begin(), dest2.end(), [](int i){ cout << i << " ";});

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// Output of partition_copy:
// True Array: 2 8 6 4 0 0 0 0 0 0
// False Array: 9 3 1 7 5 0 0 0 0 0
```

### g. partial_sort_copy

- Sorts some of the elements in the range `[first, last)` in ascending order, storing the result in the range `[d_first, d_last)`.

```
int main()
{
  vector<int> vec = {9, 2, 3, 8, 1, 7, 5, 6, 4, 0};
  vector<int> dest = {-1,-1,-1};
  cout << "Source Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  partial_sort_copy(vec.begin(), vec.end(), dest.begin(), dest.end());
  cout << "\nOutput of partial_sort_copy: ";
  for_each(dest.begin(), dest.end(), [](int i){ cout << i << " ";});

  return 0;
}
// Source Vector: 9 2 3 8 1 7 5 6 4 0
// Output of partial_sort_copy: 0 1 2
```

### std::*_n

- When '_n' is tacked onto an algorithm, it creates new algorithm that do the same thing but on a specified size.
- We have…
    - copy_n
    - fill_n
    - generate_n
    - search_n
    - for_each_n
    - uninitialized_copy_n
    - uninitialized_fill_n
    - uninitialized_move_n
    - uninitialized_default_construct_n
    - uninitialized_value_construct_n
    - destroy_n

#### a. std::copy_n

- Copies exactly count values from the range beginning at first to the range beginning at result.

```
int main()
{
    vector<int> vec {1,2,3,4,5,6,7,8,9};
    vector<int> dest(5);
    copy_n(vec.begin(), 5, dest.begin());
    cout << "Copied 5 vales form source to destination: ";
    for_each(dest.begin(),dest.end(),[](int i) { cout << i << " ";});
    return 0;
}
// Copied 5 vales form source to destination: 1 2 3 4 5
```

#### b. std::fill_n

- This function is used to fill values up to first n positions from a starting position.

```
int main()
{
    vector<int> vec {1,2,3,4,5,6,7,8,9};

    fill_n(vec.begin(), 5, 99);
    cout << "Fill first 5 places with 99: ";
    for_each(vec.begin(),vec.end(),[](int i) { cout << i << " ";});
    return 0;
}
// Fill first 5 places with 99: 99 99 99 99 99 6 7 8 9
```

### c. std::generate_n

- This is used to generate numbers based upon a generator function, and then, it assigns those values to first 'count' number of elements in the container.

```
int main()
{
    vector<int> vec {1,2,3,4,5,6,7,8,9};
    static int i = 10;
    generate_n(vec.begin(), 5, [&]() { return ++i;});
    cout << "Fill first 5 places with value generated by lambda: ";
    for_each(vec.begin(),vec.end(),[](int i) { cout << i << " ";});
    return 0;
}
// Fill first 5 places with value generated by lambda: 11 12 13 14 15 6 7 8 9
```

### d. std::search_n

- This function is used to search whether a given element satisfies a predicate a given no. of times **consecutively** with the container elements.

```
int main()
{
    char sequence[] = "1001010100010101001010101";
    cout << std::boolalpha;
    auto it = search_n(std::begin(sequence), std::end(sequence),3,'0');
    if(it!=std::end(sequence)){
      cout << "There are 3 consecutive 0s in the sequence at "
      << it-begin(sequence) << ".\n";
    }
    else {
    cout << "No 3 consecutive 0s in the sequence.\n";
    }
    return 0;
}
// There are 3 consecutive 0s in the sequence at 8.
```

### e. std::for_each_n

- Applies the given function object f to the result of dereferencing every iterator in the range [first, first + n), in order.

```
int main()
{
   vector<int> vec {1,2,3,4,5,6,7,8,9};
   cout << "Printing first 5 elements of vector: ";
   for_each_n(begin(vec), 5, [](int i) {cout << i << " ";});
   return 0;
}
// Printing first 5 elements of vector: 1 2 3 4 5
```

f. std::uninitialized_copy_n
g. std::uninitialized_fill_n
h. std::uninitialized_move_n
i. std::uninitialized_default_construct_n
j. std::uninitialized_value_construct_n
k. std::destroy_n

## 7. The Queries

- The algorithms which are categorized under 'The Queries' are used for querying a collection.
- They can be further categorized for better understanding purpose.

### 1. Numeric Algorithms

#### a. std::count

- `count()` returns the number of elements in the range `[first, last)` that compare equal to val.

```
int main() {
  std::vector<int> myArray{2, 6, 1, 3, 1, 4, 5, 1, 7};
  int searchValue = 1;

  cout<< "The " << searchValue << " appears " <<
      count(begin(myArray), end(myArray), searchValue) << " times!\n";

  std::cout << '\n';

  return 0;
}
// The 1 appears 3 times!
```

#### b. std::accumulate

- Computes the sum of the given value `'init'` and the elements in the range `[first, last)`.
- It is the part of standard library since C++98. It provides a way to fold a binary operation (such as addition) over an iterator range, resulting in a single value.
- Present under numeric header.

```
#include <numeric>

int main() {
  // CTAD C++17.
  vector v{1, 2, 3, 4, 5, 6};
  int init = 0;
  auto sum = accumulate(begin(v), end(v), /*Initial Value*/ init);
  cout << "Sum of all the elements of array: " << sum << endl;
  init = 1;
  auto product = accumulate(begin(v), end(v), /*Initial Value*/ init
                 , [](int x, int y) { return x * y; });
  cout << "Product of all the elements of array: " << product << endl;
  return 0;
}
// Sum of all the elements of array: 21
// Product of all the elements of array: 720
```

### c. std::reduce

- `std::reduce` was added in C++17 and looks remarkably similar to `std::accumulate`.
- Allowing parallelism is the main reason for addition of `std::reduce`.
- We need to make sure operation that we want to use with `std::reduce` is both associative and commutative.

```
int main() {
  // CTAD C++17.
  vector v{1, 2, 3, 4, 5, 6};

  // We don't need to specify the initial values.
  // It will consider the first element of collection as the initial value.
  auto x = reduce(begin(v),end(v));
  // In C++17, we can use {} instead of <>().
  auto y = reduce(begin(v),end(v), 0, plus{});
  auto z = reduce(begin(v),end(v), 1, multiplies<>());

  cout << "Using std::reduce, X: " << x << endl;
  cout << "Using std::reduce, Y: " << y << endl;
  cout << "Using std::reduce, Z: " << z << endl;
  return 0;
}
// Using std::reduce, X: 21
// Using std::reduce, Y: 21
// Using std::reduce, Z: 720
```

### d. std::transform_reduce

- The `std::transform_reduce` takes a 'function' and applies this function to the elements of the collection before calling the `reduce` on the collection.

```
int main() {

auto mice = vector<string>{"Mickey","Minnie","Jerry"};
auto num_chars = std::transform_reduce(begin(mice), end(mice),
                  size_t{0},                                    // Initial Value
                  [](size_t a, size_t b) { return a + b; },     // Reduce
                  [](const std::string& m) { return m.size(); } // Transform
                  );
  cout << "Total number of characters: " << num_chars << endl;
  return 0;
}
// Total number of characters: 17
```

### e. std::partial_sum

- Computes the partial sums (prefix) of the elements in the subranges of the range `[first, last)` and writes them to the another/same range.

```
int main() {
  vector myArray {1, 2, 3, 4, 5};
  vector<int> pSum(myArray.size());
  vector<int> pProduct(myArray.size());
  // pSum will be having prefix sum.
  partial_sum(begin(myArray), end(myArray), begin(pSum));
  cout << "Prefix Sum: ";
  for_each(begin(pSum), end(pSum), [](int x) {cout << x << " ";});
  // pProduct will be having prefix product.
  partial_sum(begin(myArray), end(myArray), begin(pSum), [](int x, int y){ return x*y;});
  cout << "\nPrefix Product: ";
  for_each(begin(pSum), end(pSum), [](int x) {cout << x << " ";});
  return 0;
}
// Prefix Sum: 1 3 6 10 15
// Prefix Product: 1 2 6 24 120
```

### f. std::inclusive_scan

- Computes an inclusive prefix sum operation for the range `[first, last)`, using `init` as the initial value (if provided), and writes the results to the range beginning at `d_first`.
- "`inclusive`" means that the `i-th` input element is included in the `i-th` sum.
- `std::inclusive_scan` was added in C++17 as part of the parallel STD, while `std::partial_sum` existed before.
- The summation operations may be performed in arbitrary order. The behaviour is nondeterministic if binary_op is not associative. This is because of parallelism.

```
int main() {
  vector myArray {1, 2, 3, 4, 5};
  vector<int> pSum1(myArray.size());
  vector<int> pSum2(myArray.size());
  // pSum will be having prefix sum.
  partial_sum(begin(myArray), end(myArray), begin(pSum1));
  cout << "Prefix Sum using partial_sum: ";
  for_each(begin(pSum1), end(pSum1), [](int x) {cout << x << " ";});
  cout << "\nPrefix Sum using inclusive_scan : ";
  inclusive_scan(begin(myArray), end(myArray), begin(pSum2));
  for_each(begin(pSum2), end(pSum2), [](int x) {cout << x << " ";});
  return 0;
}
// Prefix Sum using partial_sum: 1 3 6 10 15
// Prefix Sum using inclusive_scan : 1 3 6 10 15
```

### g. std::exclusive_scan

- Computes an exclusive prefix sum operation using binary_op for the range `[first, last)`, using `init` as the initial value, and writes the results to the range beginning at `d_first`.
- "`exclusive`" means that the `i-th` input element is not included in the `i-th` sum.

```
int main() {
  vector myArray {1, 2, 3, 4, 5};
  vector<int> pSum1(myArray.size());
  vector<int> pSum2(myArray.size());
  // pSum will be having prefix sum.
  inclusive_scan(begin(myArray), end(myArray), begin(pSum1));
  cout << "Prefix Sum using inclusive_scan: ";
  for_each(begin(pSum1), end(pSum1), [](int x) {cout << x << " ";});
  cout << "\nPrefix Sum using exclusive_scan : ";
  // Sum of all elements before it and not including the current one.
  exclusive_scan(begin(myArray), end(myArray), begin(pSum2), 0);
  for_each(begin(pSum2), end(pSum2), [](int x) {cout << x << " ";});
  return 0;
}
// Prefix Sum using inclusive_scan: 1 3 6 10 15
// Prefix Sum using exclusive_scan : 0 1 3 6 10
```

### h. std::transform_inclusive_scan

- The `std::transform_inclusive_scan` takes a '`function`' and applies this function to the elements of the collection before calling the `inclusive_scan` on the collection.
- "`inclusive`" means that the `i-th` input element is included in the `i-th` sum.

```
int main() {
  vector myArray {1, 2, 3, 4, 5};
  vector<int> pSum(myArray.size());
  cout <<"Original Array: ";
  for_each(myArray.begin(), myArray.end(),[](int i){cout << i << " ";});
  transform_inclusive_scan(begin(myArray), end(myArray), begin(pSum),
    // + Operation applied.
    [](int x, int y){return x+y;},
    // Transformation - If even, return 0, else return the element itself.
    [](int i) { if(i%2 == 0) return 0; return i; },
    100 // Initial Value
    );
  cout << "\nAfter transform_inclusive_scan:";
  for_each(pSum.begin(), pSum.end(),[](int i){cout << i << " ";});
  return 0;
}
// Original Array: 1 2 3 4 5
// After transform_inclusive_scan:101 101 104 104 109
```

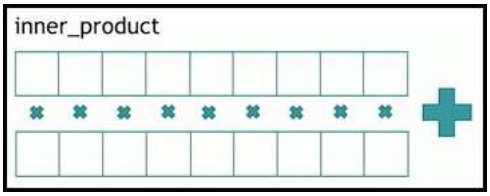
### i. std::transform_exclusive_scan

- The `std::transform_exclusive_scan` takes a `'function'` and applies this function to the elements of the collection before calling the `exclusive_scan` on the collection.
- `" exclusive"` means that the `i-th` input element is excluded in the `i-th` sum.

```
int main() {
  vector myArray {1, 2, 3, 4, 5};
  vector<int> pSum(myArray.size());

  cout <<"Original Array: ";
  for_each(myArray.begin(), myArray.end(),[](int i){cout << i << " ";});
  transform_inclusive_scan(begin(myArray), end(myArray), begin(pSum),
    // + Operation applied.
    [](int x, int y){return x+y;},
    // Transformation - If even, return 0, else return the element itself.
    [](int i) { if(i%2 == 0) return 0; return i; },
    100 // Initial Value
    );
  cout << "\nAfter transform_inclusive_scan:";
  for_each(pSum.begin(), pSum.end(),[](int i){cout << i << " ";});

  transform_exclusive_scan(begin(myArray), end(myArray), begin(pSum),
    100, // Initial Value
    [](int x, int y){return x+y;}, // + Operation applied.
    // Transformation - If even, return 0, else return the element itself.
    [](int i) { if(i%2 == 0) return 0; return i; }
    );

  cout << "\nAfter transform_exclusive_scan:";
  for_each(pSum.begin(), pSum.end(),[](int i){cout << i << " ";});
  return 0;
}
// Original Array: 1 2 3 4 5
// After transform_inclusive_scan:101 101 104 104 109
// After transform_exclusive_scan:100 101 101 104 104
```

### j. std::inner_product

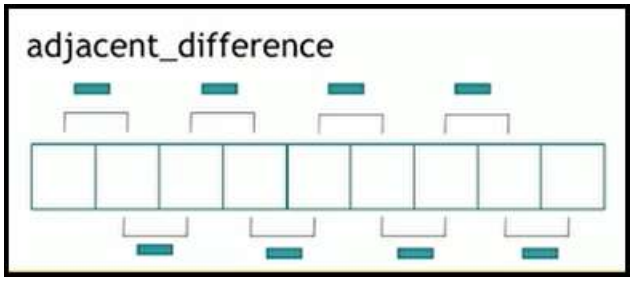
- Compute cumulative inner product of range.
- Returns the result of accumulating `init` with the inner products of the pairs formed by the elements of two ranges starting at `first1` and `first2`.



```
int main() {
  vector a {1, 2, 3, 4, 5};
  vector b {5, 4, 3, 2, 1};
  // (1*5) + (2*4) + (3*3) + (4*2) + (5*1)
  // 5 + 8 + 9 + 8 + 5 = 35 + (Initial Value = 0)
  auto ip = inner_product(begin(a), end(a), begin(b), 0 /*Initial Value*/);
  cout << "Inner product: " << ip << endl;
  return 0;
}
// Inner product: 35
```

### k. std::adjacent_difference

- Compute adjacent difference of range.
- Computes the differences between the second and the first of each adjacent pair of elements of the range `[first, last)` and writes them to the range beginning at `d_first + 1`.




```
int main() {
  vector a{ 2, 1, 3, 5, 9, 11, 12 };
  vector<int> aDiff(a.size());

  // aDiff[0] = a[0]                = 2
  // aDiff[1] = a[1] - a[0] = 1 - 2 = -1
  // aDiff[2] = a[2] - a[1] = 3 - 1 = 2
  adjacent_difference(a.begin(), a.end(), aDiff.begin());
  cout << "Adjacent Difference: ";
  for_each(aDiff.begin(), aDiff.end(),[](int i){cout << i << " ";});

  return 0;
}
// Adjacent Difference: 2 -1 2 2 4 2 1
```

### l. std::sample

- Selects n elements from the sequence `[first; last)` and writes those selected elements into the output iterator.
- If n is greater than the number of elements in the sequence, selects last-first elements.

```
int main() {

  vector<int> numbers(100);
  int i = 0;
  // Fill the Vector.
  for_each(begin(numbers), end(numbers), [&](int& x) { x = ++i; });

  vector<int> samples(5);
  sample(begin(numbers), end(numbers), begin(samples),
   5, std::mt19937{std::random_device{}()});
  cout << "5 Ramdomly picked samples: ";
  for_each(begin(samples), end(samples),[](int& x) {cout << x << " ";});

  return 0;
}
// 5 Ramdomly picked samples: 11 38 40 55 95
```

## 2. Querying property

- Apart from querying a value, we can query property.
- We have…
    - std::all_of
    - std::any_of
    - std::none_of

### a. std::all_of

- It checks for a given property on every element and returns `true` when each element in range satisfies specified property, else returns `false`.
- It returns `true` if the range is empty.

```
int main() {

  vector evenNumbers { 2, 4, 6, 8, 10, 12, 14, 16 };
  bool areEven = all_of(begin(evenNumbers), end(evenNumbers),
  [](const int& i) { return !(i&1);});

  if(areEven)
    cout << "All numbers are even!!!\n";

  return 0;
}
// All numbers are even!!!
```

### b. std::any_of

- Returns `true` if at least one element satisfies the property else returns false.

```
int main() {

  vector evenNumbers { 2, 4, 6, 8, 10, 12, 14, 16 };
  vector numbers { 2, 4, 6, 8, 111, 12, 14, 16 };
  bool anyOddFound = any_of(begin(evenNumbers), end(evenNumbers),
  [](const int& i) { return (i&1);});
  if(anyOddFound)
    cout << "One of the elements in the 'evenNumbers' collection is ODD!!!\n";
  else
    cout << "None of the elements in the 'evenNumbers' collection is ODD!!!\n";

  anyOddFound = any_of(begin(numbers), end(numbers),
  [](const int& i) { return (i&1);});
  if(anyOddFound)
    cout << "One of the elements in the 'numbers' collection is ODD!!!\n";
  else
    cout << "None of the elements in the 'numbers' collection is ODD!!!\n";
  return 0;
}
// None of the elements in the 'evenNumbers' collection is ODD!!!
// One of the elements in the 'numbers' collection is ODD!!!
```

### c. std::none_of

- `std::none_of` function returns `true` if certain condition returns `false` for all the elements in the range `[first, last)` or if the range is empty, and `false` otherwise.

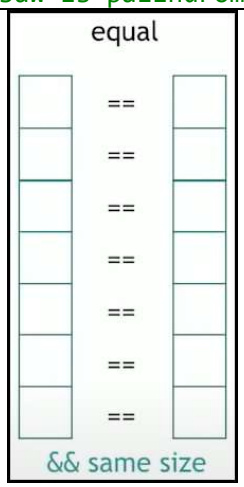
```
int main() {

  vector evenNumbers { 2, 4, 6, 8, 10, 12, 14, 16 };

  bool anyOddFound = none_of(begin(evenNumbers), end(evenNumbers),
  [](const int& i) { return (i&1);});
  if(anyOddFound)
    cout << "One of the elements in the 'evenNumbers' collection is ODD!!!\n";
  else
    cout << "None of the elements in the 'evenNumbers' collection is ODD!!!\n";

  return 0;
}
```

## 3. Querying property on 2 ranges

### a. std::equal

- `std::equal` helps to compares the elements within the range `[first_1,last_1)` with those within range beginning at `first_2`.

```
int main() {

  string s1 = "Was it a car or a cat I saw";
  // Remove spaces
  s1.erase(remove(begin(s1), end(s1), ' '), end(s1));
  bool isPalindrome =
      equal(begin(s1), begin(s1) + s1.size() / 2, rbegin(s1),
            [](char c1, char c2) { return (tolower(c1) == tolower(c2)); });

  if (isPalindrome)
    cout << "The string " << s1 << " is palindrome!!!\n";
  else
    cout << "The string " << s1 << " is not palindrome!!!\n";

  return 0;
}
// The string WasitacaroracatIsaw is palindrome!!!
```




### b. std::lexicographical_compare

- Checks if the first range is lexicographically less than the second range.
- Comparing strings can be generally used in dictionary, where we need to place words in lexicographical order. That is what `std::lexicographical_compare` does.

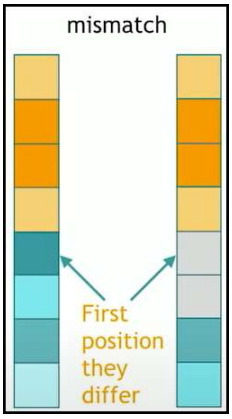
```
Example:
If we consider 2 strings:
  string S1 = "abcdefgh"; // Length is 8
  string S2 = "abyz";     // Length is 4
```

Even though the length of S1 (Length = 8) is greater than length of S2 (Length = 4), the string S1 is smaller and comes first if we were to put this in a dictionary.

```
int main() {
  string S1 = "abcdefgh"; // Length is 8
  string S2 = "abyz";     // Length is 4
  bool isS1SmallerThanS2 =
      lexicographical_compare(begin(S1), end(S1), begin(S2), end(S2));
  if (isS1SmallerThanS2)
    cout << "String S1: " << S1
         << " comes first in the dictionary When compared to S2: " << S2
         << ".\n";
  else
    cout << "String S2: " << S2
         << " comes first in the dictionary When compared to S1: " << S1
         << ".\n";
  return 0;
}
// String S1: abcdefgh comes first in the dictionary When compared to S2: abyz.
```

### c. std::mismatch

- Compares the elements in the Range1 with those in the Range2 and returns the first element of both sequences that does not match.
- The function returns a pair of iterators to the first element in each range that does not match.




```
int main()
{
  vector<int> vector1= {10, 20, 30, 40, 50, 60, 77, 80, 100};
  vector<int> vector2= {10, 20, 30, 40, 50, 60, 70, 80, 100};

  pair< vector<int>::iterator,  vector<int>::iterator > mismatch_pair;
  mismatch_pair = mismatch(vector1.begin(), vector1.end(), vector2.begin());

  cout << "The first mismatch pair from vector1 and vector2 is: ";
  cout << *mismatch_pair.first << " " << *mismatch_pair.second << endl;

  return 0;
}
// The first mismatch pair from vector1 and vector2 is: 77 70
```

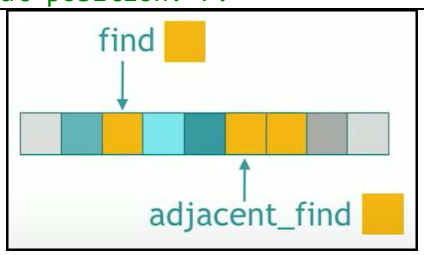
## 4. Searching a value

### a. Unsorted Range

#### *i. std::find*

- Returns an iterator to the first element in the range `[first,last)` that compares equal to val.
- If no such element is found, the function returns last.

```
int main() {
  vector v1{8, 5, 9, 2, 7, 1, 3, 10};
  auto searchVal = 10;
  auto it = std::find(v1.begin(), v1.end(), searchVal);
  if (it != v1.end()) {
    std::cout << "Element " << searchVal << " found at position: ";
    std::cout << it - v1.begin() << ".\n";
  }
  else {
    cout << "Element " << searchVal << " not found!\n";
  }
  return 0;
}
// Element 10 found at position: 7.
```




#### *ii. std::adjacent_find*

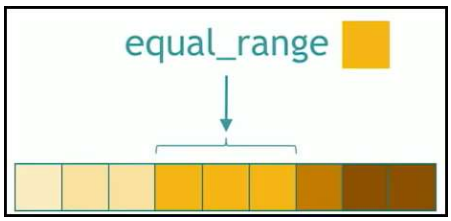
- Searches the range `[first, last)` for **two consecutive equal elements**, and returns an iterator to the first of these two elements, or last if no such pair is found.

```
int main() {
  vector v1{8, 5, 9, 2, 7, 1, 1, 3, 10};

  auto it = adjacent_find(v1.begin(), v1.end());
  if (it != v1.end()) {
    std::cout << "Consecutive " << *it << "'s found at position: ";
    std::cout << it - v1.begin() << ".\n";
  }
  else {
    cout << "No consecutive elements were found!\n";
  }
  return 0;
}
// Consecutive 1's found at position: 5.
```

### b. Sorted Range

#### *i. std::equal_range*

- `std::equal_range` is used to find the sub-range within a given range `[first, last)` that has all the elements equivalent to a given `value`. It returns the initial and the final bound of such a sub-range.
- The elements in the range shall already be **sorted** according to this same criterion (operator< or comp), or at least partitioned with respect to val.




```
int main() {
  vector v1{8, 5, 9, 2, 20, 23, 19, 19, 3, 19, 29, 19, 10};
  auto searchVal = 19;
  sort(v1.begin(), v1.end());
  auto pairOfIt = equal_range(v1.begin(), v1.end(), searchVal);
  cout << "The " << searchVal << " found in between " <<
  pairOfIt.first - v1.begin() << " and " << pairOfIt.second - v1.begin()
  << "." << endl;
  return 0;
}
// The 19 found in between 6 and 10.
```

#### *ii. std::lower_bound*

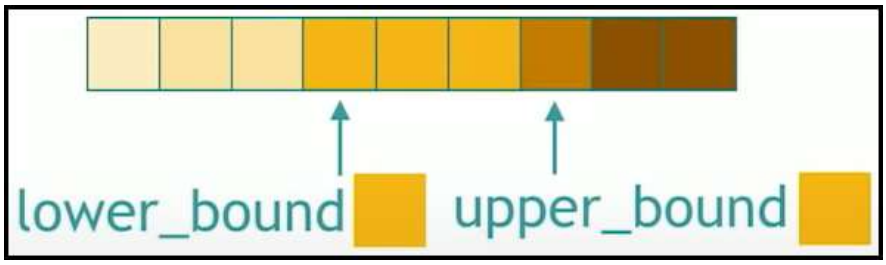
- Returns an iterator pointing to the first element in the range `[first, last)` which has a value not less than `val`. This means that the function returns the index of the next smallest number just greater than or equal to that number.
- The elements in the range shall already be **sorted** or at least partitioned with respect to `val`.

```
int main() {
  vector v1{8, 5, 9, 2, 20, 23, 19, 19, 3, 19, 29, 19, 10};
  auto searchVal = 19;
  sort(v1.begin(), v1.end());
  auto it = lower_bound(v1.begin(), v1.end(), searchVal);
  cout << "Array after sorting: ";
  for_each(v1.begin(), v1.end(), [](int x) { cout << x << " "; });
  cout <<'\n';
  cout << "To insert " << searchVal << " the lower bound is at: "
       << it - v1.begin() << '\n';
  return 0;
}
// Array after sorting: 2 3 5 8 9 10 19 19 19 19 20 23 29
// To insert 19 the lower bound is at: 6
```

### iii. *std::upper_bound*

- Returns an iterator pointing to the first element in the range `[first, last)` that is greater than `value`, or last if no such element is found.
- The elements in the range shall already be sorted or at least partitioned with respect to `val`.

```
int main() {
  vector v1{8, 5, 9, 2, 20, 23, 19, 19, 3, 19, 29, 19, 10};
  auto searchVal = 19;
  sort(v1.begin(), v1.end());
  auto it = upper_bound(v1.begin(), v1.end(), searchVal);
  cout << "Array after sorting: ";
  for_each(v1.begin(), v1.end(), [](int x) { cout << x << " "; });
  cout <<'\n';
  cout << "To insert " << searchVal << " the upper bound is at: "
       << it - v1.begin() << '\n';
  return 0;
}
// Array after sorting: 2 3 5 8 9 10 19 19 19 19 20 23 29
// To insert 19 the upper bound is at: 10
```




### iv. *std::binary_search*

- Returns `true` if any element in the range `[first, last)` is equivalent to `val`, and `false` otherwise.
- The elements in the range shall already be sorted or at least partitioned with respect to `val`.

```
using namespace std;

struct Interval {
  int start;
  int end;
  bool operator < (Interval const& other) const {
    return start < other.start;
  }
};
```

```
// Custom Comparator To Sort 'Interval' Objects.
// Sorting based on 'start' time.
bool compareStartTime(Interval a, Interval b) {
  return a.start < b.start;
}

int main () {

  vector<Interval> intervals = { { 1, 3 }, { 2, 4 }, { 1, 7 }, { 2, 19 },
                                 { 2, 7 }, { 3, 6 }, { 2, 8 }, { 5, 15 } };
  cout << "Original Vector: ";
  for(const auto& i: intervals){
    cout << "{" << i.start << ", " << i.end << "} ";
  }

  sort(intervals.begin(), intervals.end(), compareStartTime);
  cout << "\nstd::sort based on Start Time: ";
  for(const auto& i: intervals){
    cout << "{" << i.start << ", " << i.end << "} ";
  }
  Interval searchVal { 2, 19 };
  bool found = binary_search(begin(intervals), end(intervals), searchVal);

  if(found)
    cout <<"\nThe element { 2, 19 } found in the intervals array.\n";
  else
    cout <<"\nThe element { 2, 19 } was not found in the intervals array.\n";

  return 0;
}
// Original Vector: {1, 3} {2, 4} {1, 7} {2, 19} {2, 7} {3, 6} {2, 8} {5, 15}
// std::sort based on Start Time: {1, 3} {1, 7} {2, 4} {2, 19} {2, 7} {2, 8}
{3, 6} {5, 15}
// The element { 2, 19 } found in the intervals array.
```

## 5. Searching a range

- Till now we were looking for a value in a collection. We can also search for range in a collection.
- We have…
  - std::search
  - std::find_end
  - std::find_first_of

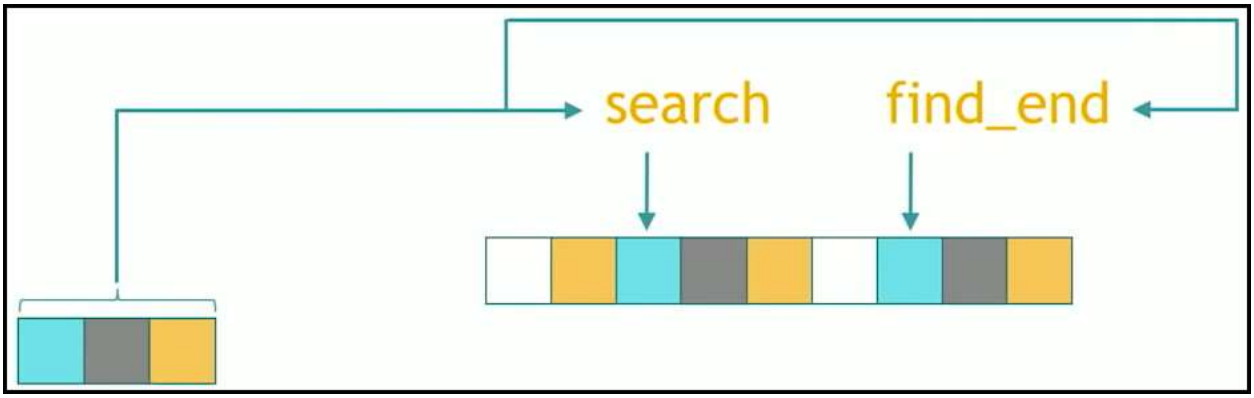
### a. std::search

- Used to find out the presence of a subsequence satisfying a condition with respect to another sequence.
- Searches the range `[first1, last1)` for the first occurrence of the sequence defined by `[first2, last2)`, and returns an iterator to its first element, or `last1` if no occurrences are found.

```
int main () {
  vector<int> vec;
  for (int i=1; i<10; i++) vec.push_back(i*10);

  vector<int> searchSeq = { 40, 50, 60 };

  auto it = search(begin(vec), end(vec), begin(searchSeq), end(searchSeq));
  if(it!= end(searchSeq))
    cout << "Found the search sequence in vector!\n";
  else
    cout <<"Search sequence not found!\n";

  return 0;
}
// Found the search sequence in vector!
```

### b. std::find_end

- `std::find_end` is used to find the last occurrence of a sub-sequence inside a container.
- It searches the range `[first1, last1)` for the last occurrence of the sequence defined by `[first2, last2)`, and returns an iterator to its first element, or `last1` if no occurrences are found.

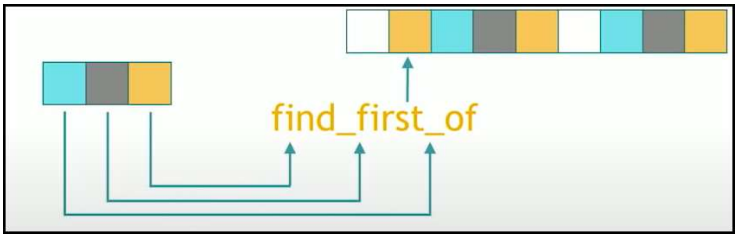


```
int main () {
  vector<int> vec {10, 20, 30, 40, 50, 60, 70, 80, 90, 40, 50, 60};
  vector<int> searchSeq = { 40, 50, 60 };
  auto it = search(begin(vec), end(vec), begin(searchSeq), end(searchSeq));
  if(it!= end(searchSeq))
    cout << "Found the search sequence in vector at " << it - begin(vec) << "!\n";
  else
    cout <<"Search sequence not found!\n";
  it = find_end(begin(vec), end(vec), begin(searchSeq), end(searchSeq));
  if(it!= end(searchSeq))
    cout << "Found the search sequence in vector at " << it - begin(vec) << "!\n";
  else
    cout <<"Search sequence not found!\n";

  return 0;
}
// Found the search sequence in vector at 3!
// Found the search sequence in vector at 9!
```

c. std::find_first_of

- Used to compare elements between two containers. It compares all the elements in a range `[first1, last1)` with the elements in the range `[first2, last2)`, and if any of the elements present in the second range is found in the first one, then it returns an iterator to that element.
- In case there is no match, then iterator pointing to `last1` is returned.

```
int main () {
  vector<int> vec {10, 20, 90, 40, 50, 60, 70, 80, 30, 40, 50, 60};
  vector<int> searchSeq = { 30, 50, 60 };

  auto it = find_first_of(begin(vec), end(vec), begin(searchSeq), end(searchSeq));
  if(it!= end(vec))
    cout << "Found the one of the search element in vector at "
    << it - begin(vec) << " which is " << *it << "!\n";
  else
    cout <<"Search sequence not found!\n";
  return 0;
}
// Found the one of the search element in vector at 4 which is 50!
```



## 6. Searching a relative value

- We can search for a relative value such as max element or min element or both in a collection.
- We have…
    - std::max_element
    - std::min_element
    - std::minmax_element

### a. std::max_element

- Returns an iterator pointing to the element with the largest value in the range `[first,last)`.
- We can also find the max element in the sub-section of the list.
- If more than one element satisfies the condition of being the largest, the iterator returned points to the first of such elements.

```
int main () {
  vector<int> vec {10, 20, 90, 40, 50, 60, 70, 80, 30, 90, 50, 60};

  auto it = max_element(begin(vec), end(vec));
  cout << "The largest element in the vector is " << *it <<
  " and is at the position " << it - vec.begin() << ".\n";

  return 0;
}
// The largest element in the vector is 90 and is at the position 2.
```

### b. std::min_element

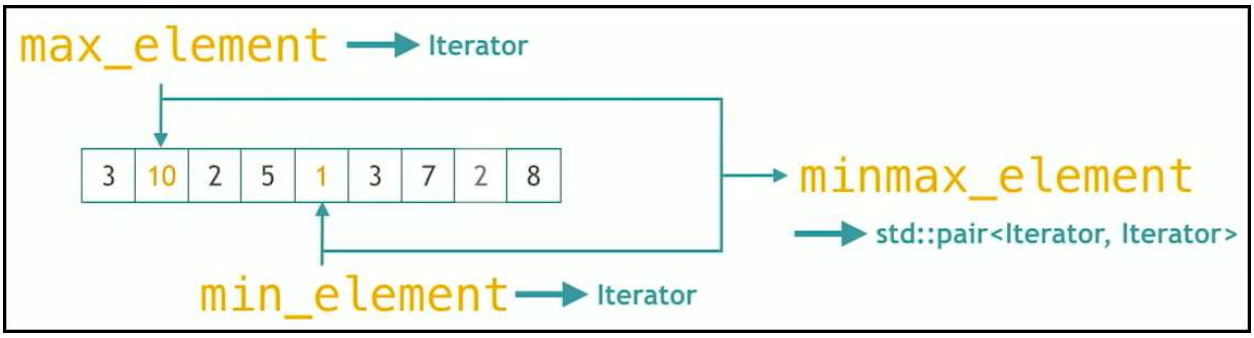
- Returns an iterator pointing to the element with the smallest value in the range `[first,last)`.
- We can also find the min element in the sub-section of the list.
- If more than one element satisfies the condition of being the largest, the iterator returned points to the first of such elements.

```
int main () {
  vector<int> vec {80, 20, 90, 40, 50, 60, 10, 80, 30, 90, 50, 10};

  auto it = min_element(begin(vec), end(vec));
  cout << "The smallest element in the vector is " << *it <<
  " and is at the position " << it - vec.begin() << ".\n";

  return 0;
}
// The smallest element in the vector is 10 and is at the position 6.
```

### c. std::minmax_element

- Finds the smallest and greatest element in the range [first, last).
- This function returns **pair pointer**, whose **1st element points** to the position of **minimum** element in the range and **2nd element points** to the position of **maximum** element in the range.
- If there are more than 1 minimum numbers, then the **1st pointer points** to **first** occurring element. If there are more than 1 maximum numbers, then the **2nd element points** to **last** occurring element.

```
int main () {
  vector<int> vec {80, 20, 90, 40, 50, 60, 10, 80, 30, 90, 50, 10};

  auto itPair = minmax_element(begin(vec), end(vec));
  cout << "The smallest and largest element in the vector is " << *(itPair.first) <<
  " and " << *(itPair.second) << ".\n";
  cout << "The smallest element is at the position: " << itPair.first - vec.begin()
<< ".\n";
  cout << "The largest element is at the position: " << itPair.second - vec.begin()
<< ".\n";

  return 0;
}
// The smallest and largest element in the vector is 10 and 90.
// The smallest element is at the position: 6.
// The largest element is at the position: 9.
```



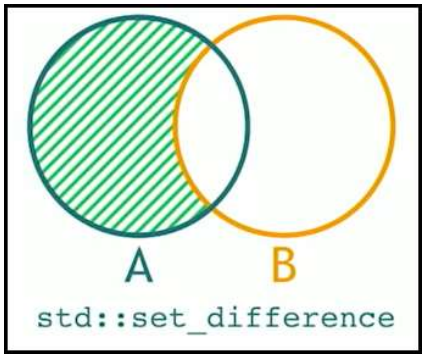
## 8. Algorithms on Sets

*Set*

- Sets are containers that store **unique elements** following a specific order. In C++ any sorted collection is also considered as `set`.
- The `std::set` is sorted and it is a `set`. The sorted `vector` is also considered as set.
- We have set algorithms which can be applied on any range.
    - std::set_difference
    - std::set_intersection
    - std::set_union
    - std::set_symmetric_difference
    - std::includes
    - std::merge

### a. std::set_difference

- Copies the elements from the **sorted** range `[first1, last1)` which are not found in the **sorted** range `[first2, last2)` to the range beginning at `d_first`.

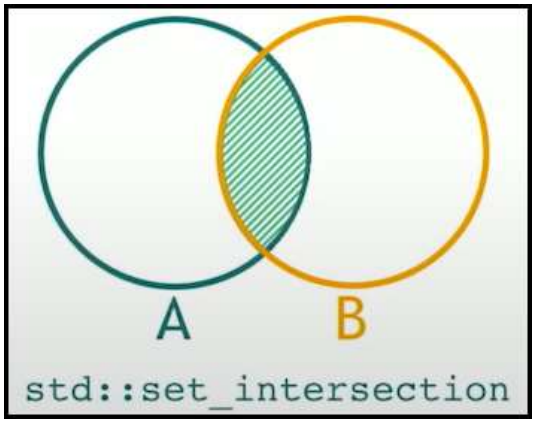



```
int main() {
  vector vec1{80, 20, 90, 40, 50, 60, 10, 80, 30, 90, 50, 10};
  vector vec2{80, 20, 90, 40, 60, 80};
  vector<int> destination(6);
  // set_difference operates on sorted collections.
  sort(vec1.begin(), vec1.end());sort(vec2.begin(), vec2.end());
  set_difference(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(),
                 destination.begin());
  cout << "vec1 elements which are not present in vec2 are: ";
  for_each(destination.begin(), destination.end(),
           [](int i) { cout << i << " "; });
  return 0;
}
// vec1 elements which are not present in vec2 are: 10 10 30 50 50 90
```

### b. std::set_intersection

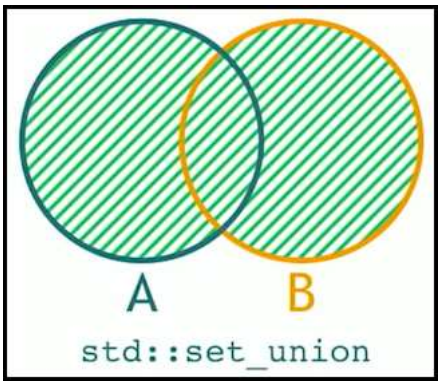
- The intersection of two sets is formed only by the elements that are present in both sets.
- The elements copied by the function come always from the first range, in the same order.
- The elements in the both the ranges shall already be ordered.

```
int main() {
  vector vec1{80, 20, 90, 40, 50, 60, 10, 80, 30, 90, 50, 10};
  vector vec2{80, 20, 90, 40, 60, 80};
  vector<int> destination(6);
  // set_intersection operates on sorted collections.
  sort(vec1.begin(), vec1.end());
  sort(vec2.begin(), vec2.end());
  set_intersection(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(),
                   destination.begin());
  cout << "Common elements which are present in both vectors are: ";
  for_each(destination.begin(), destination.end(),
           [](int i) { cout << i << " "; });

  return 0;
}
// Common elements which are present in both vectors are: 20 40 60 80 80 90
```




### c. std::set_union

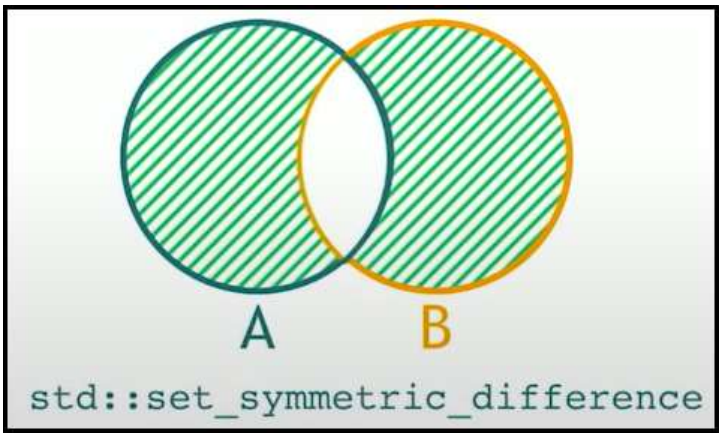
- Constructs a sorted union beginning at `d_first` consisting of the set of elements present in one or both sorted ranges `[first1, last1)` and `[first2, last2)`.
- If some element is found m times in `[first1, last1)` and n times in `[first2, last2)`, then all m elements will be copied from `[first1, last1)` to `d_first`, preserving order, and then exactly `std::max(n-m, 0)` elements will be copied from `[first2, last2)` to `d_first`, also preserving order.




```
int main() {
  vector vec1{80, 20, 90, 40, 50, 60, 10, 30};
  vector vec2{80, 20, 90, 40, 60};
  vector<int> destination(vec1.size()+vec2.size());
  // set_union operates on sorted collections.
  sort(vec1.begin(), vec1.end());
  sort(vec2.begin(), vec2.end());
  set_union(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(),
                destination.begin());
  cout << "Set Union: ";
  for_each(destination.begin(), destination.end(),
           [](int i) { cout << i << " "; });
  return 0;
}
// Set Union: 10 20 30 40 50 60 80 90 0 0 0 0 0
```

### d. std::set_symmetric_difference

- Computes symmetric difference of two sorted ranges: the elements that are found in either of the ranges, but not in both are copied to the range beginning at `d_first`.
- The resulting range is also sorted.



```
int main() {
  vector vec1{80, 20, 90, 40, 50, 60, 10, 30};
  vector vec2{80, 20, 90, 40, 60, 75};
  vector<int> destination(5);
  // set_symmetric_difference operates on sorted collections.
  sort(vec1.begin(), vec1.end());
  sort(vec2.begin(), vec2.end());
  set_symmetric_difference(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(),
                destination.begin());
  cout << "Set Symmetric Difference: ";
  for_each(destination.begin(), destination.end(),
           [](int i) { cout << i << " "; });
  return 0;
}
// Set Symmetric Difference: 10 30 50 75 0
```

### e. std::includes

- Returns `true` if the sorted range `[first2, last2)` is a subsequence of the sorted range `[first1, last1)`.


```
int main() {
  vector vec1{80, 20, 90, 40, 50, 60, 10, 30};
  vector vec2{80, 20, 90, 60, 10};
  // includes operates on sorted collections.
  sort(vec1.begin(), vec1.end());
  sort(vec2.begin(), vec2.end());

  auto isSubsequence = includes(vec1.begin(), vec1.end(), vec2.begin(), vec2.end());
  if(isSubsequence)
    cout << "vec2 is a sub-sequence of vec1.\n";
  else
   cout << "vec2 is not a sub-sequence of vec1.\n";
  return 0;
}
// vec2 is a sub-sequence of vec1.
```

### f. std::merge

- Combines the elements in the sorted ranges [first1, last1) and [first2, last2), into a new range with all its elements sorted.

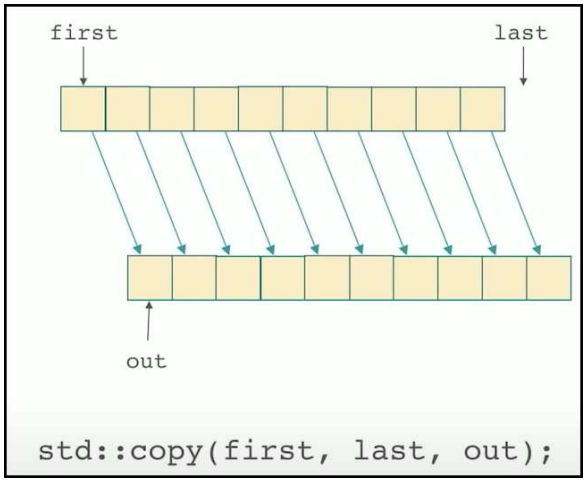
```
int main() {
  vector vec1{80, 20, 90, 40, 50, 60, 10, 30};
  vector vec2{80, 20, 90, 60, 10};
  vector<int> destination(vec1.size() + vec2.size());
  // merge  operates on sorted collections.
  sort(vec1.begin(), vec1.end());
  sort(vec2.begin(), vec2.end());
  merge(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(),
        destination.begin());

  cout << "Merge of vec1 and vec2:\n";
  for_each(destination.begin(), destination.end(),
           [](int x) { cout << x << " "; });

  return 0;
}
// Merge of vec1 and vec2:
// 10 10 20 20 30 40 50 60 60 80 80 90 90
```

## 9. Movers

- These STL algorithms moves things around.
- We have…
    - std::copy
    - std::move
    - std::swap_ranges
    - std::copy_backward
    - std::move_backward

### a. std::copy

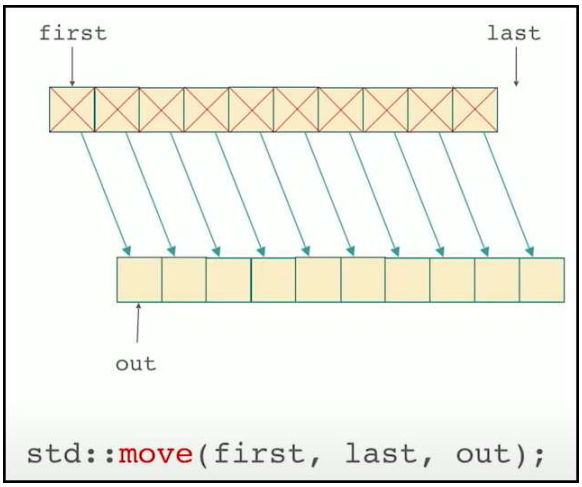
- Copies the elements in the range, defined by `[first, last)`, to another range beginning at `d_first`.




```
int main() {
  vector vec{80, 20, 90, 40, 50, 60, 10, 30};
  vector<int> destination(vec.size());
  copy(vec.begin(), vec.end(), destination.begin());
  cout << "After copy, the destination is :\n";
  for_each(destination.begin(), destination.end(),
           [](int x) { cout << x << " "; });

  return 0;
}
// After copy, the destination is :
// 80 20 90 40 50 60 10 30
```

### b. std::move

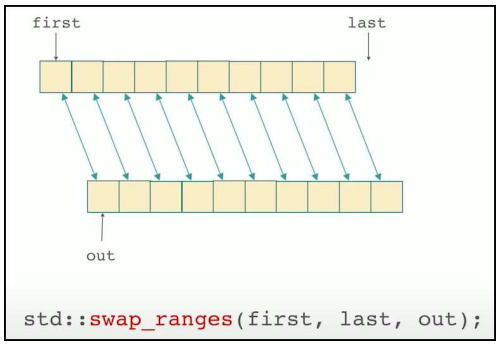
- Moves the elements in the range `[first, last]` into the range beginning at result.
- The value of the elements in the `[first, last]` is transferred to the elements pointed by result.
- After the call, the elements in the range `[first, last]` are left in an unspecified but valid state.




```
int main() {
  vector vec{80, 20, 90, 40, 50, 60, 10, 30};
  vector<int> destination(vec.size());
  move(vec.begin(), vec.end(), destination.begin());
  cout << "After move, the destination is :\n";
  for_each(destination.begin(), destination.end(),
           [](int x) { cout << x << " "; });

  return 0;
}
// After move, the destination is :
// 80 20 90 40 50 60 10 30
```

### c. std::swap_ranges

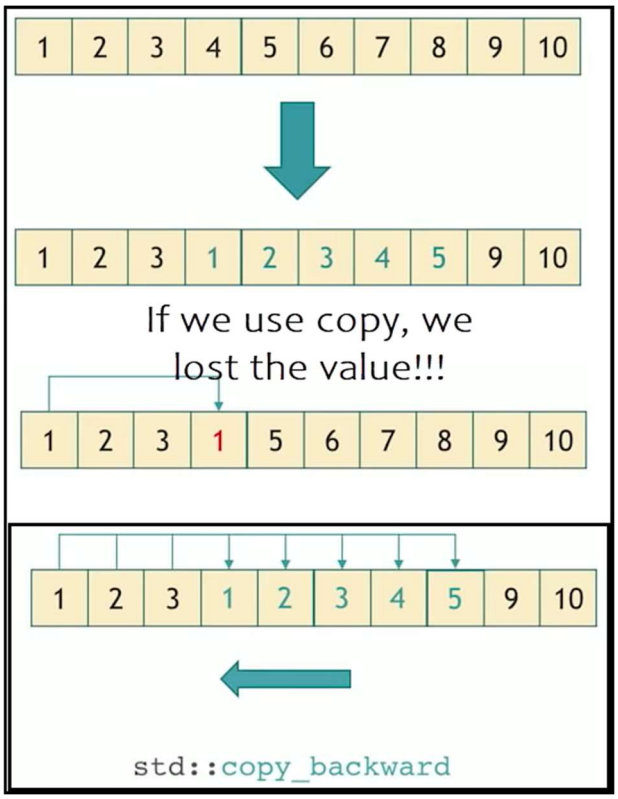
- It simply exchanges the values of each of the elements in the range `[first1, last1)` with those of their respective elements in the range beginning at `first2`.



```cpp
auto print = [](auto comment, auto const& seq) {
    std::cout << comment;
    for (const auto& e : seq) { std::cout << e << ' '; }
    std::cout << '\n';
};

int main() {
    std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};
    std::list<char> l = {'1', '2', '3', '4', '5'};
    print("Before swap_ranges:\n" "Vector: ", v);
    print("List: ", l);
    std::swap_ranges(v.begin(), v.begin()+3, l.begin());
    print("After swap_ranges:\n" "Vector: ", v);
    print("List: ", l);
    return 0;
}
// Before swap_ranges:
// Vector: a b c d e
// List: 1 2 3 4 5
// After swap_ranges:
// Vector: 1 2 3 d e
// List: a b c 4 5
```

### d. std::copy_backward

- Copies the elements from the range, defined by `[first, last)`, to another range ending at `d_last`. The elements are copied in reverse order (the last element is copied first), but their relative order is preserved.



```
int main()
{
  vector vec {1,2,3,4,5,6,7,8,9,10};
  // Expected values in vec: 1,2,3,1,2,3,4,5,9,10
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  copy_backward(vec.begin(), vec.begin()+5, vec.end() - 2);
  cout << "\nAfter copy_backward: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 4 5 6 7 8 9 10
// After copy_backward: 1 2 3 1 2 3 4 5 9 10
```

### e. std::move_backward

- Moves the elements from the range [`first`, `last`), to another range ending at `d_last`. The elements are moved in reverse order (the last element is moved first), but their relative order is preserved.

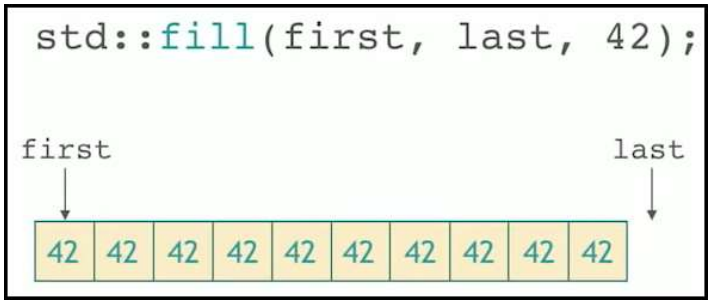
```
int main()
{
  vector vec {1,2,3,4,5,6,7,8,9,10};
  // Expected values in vec: 1,2,3,1,2,3,4,5,9,10
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  move_backward(vec.begin(), vec.begin()+5, vec.end() - 2);
  cout << "\nAfter move_backward: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 4 5 6 7 8 9 10
// After move_backward: 1 2 3 1 2 3 4 5 9 10
```

## 10. Value Modifiers

- These algorithms change the values inside of a collection.
- We have…
  - std::fill
  - std::generate
  - std::iota
  - std::replace

### a. std::fill

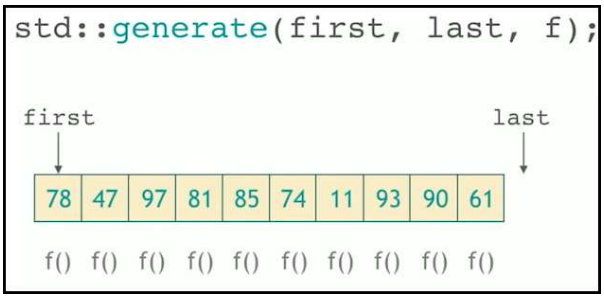
- Assigns the given `value` to the elements in the range `[first, last)`.




```
int main()
{
  vector vec {1,2,3,4,5,6,7,8,9,10};
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});

  fill(vec.begin(), vec.end(), -1);
  cout << "\nAfter std::fill: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 4 5 6 7 8 9 10
// After std::fill: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

### b. std::generate

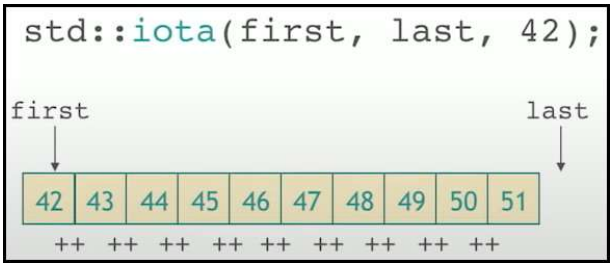
- Assigns each element in range `[first, last)` a value generated by the given function object g.



```
int main()
{
  vector vec {1,2,3,4,5,6,7,8,9,10};
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  static int i = 100;
  generate(vec.begin(), vec.end(), [&](){return ++i;});
  cout << "\nAfter std::generate: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 4 5 6 7 8 9 10
// After std::generate: 101 102 103 104 105 106 107 108 109 110
```

### c. std::iota

- Fills the range [first, last) with sequentially increasing values, starting with value and repetitively evaluating ++value.




```
int main()
{
  vector vec {1,2,3,4,5,6,7,8,9,10};
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  iota(vec.begin(), vec.end(), 11);
  cout << "\nAfter std::iota: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 4 5 6 7 8 9 10
// After std::iota: 11 12 13 14 15 16 17 18 19 20
```

### d. std::replace

- Replaces all elements satisfying specific criteria with `new_value` in the range `[first, last)`.

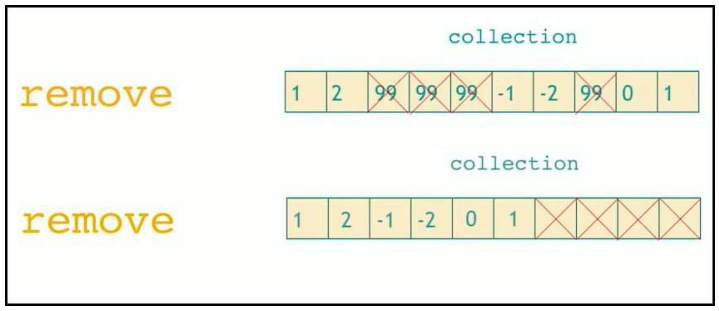
```
int main()
{
  vector vec {1,2,3,3,5,3,3,3,9,10};
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  replace(vec.begin(), vec.end(), 3, -1);
  cout << "\nAfter std::replace: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 3 5 3 3 3 9 10
// After std::replace: 1 2 -1 -1 5 -1 -1 -1 9 10
```

# 11. Changing the Structure

- These algorithms when applied, changes the collections.
- We have…
  - std::remove
  - std::unique

## a. std::remove

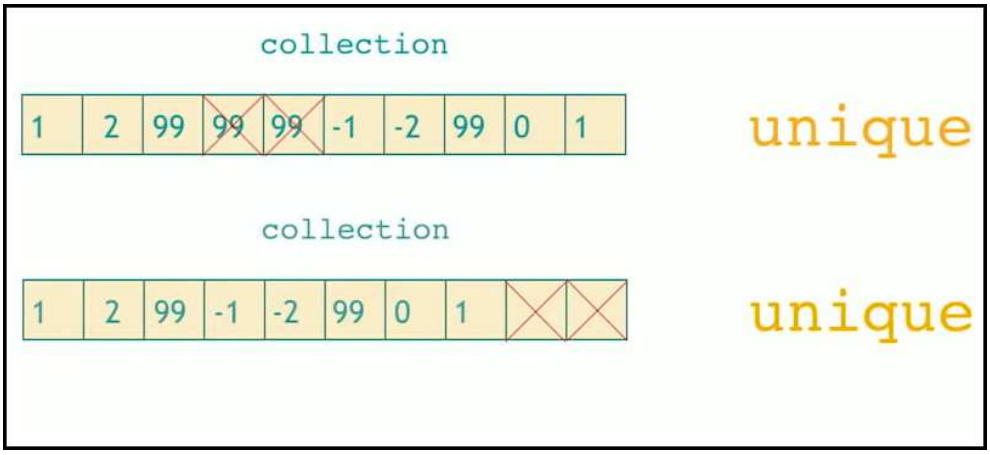
- Removes all elements satisfying specific criteria from the range `[first, last)` and returns a `past-the-end` iterator for the new end of the range.
  - Removes all elements that are equal to value.
  - If we have given a predicate, removes all elements for which the predicate returns true.




```
int main()
{
  vector vec {1,2,3,3,5,3,3,3,9,10};
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  auto it  = remove(vec.begin(), vec.end(), 3);
  cout << "\nAfter std::remove: ";
  for_each(vec.begin(), it, [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 3 5 3 3 3 9 10
//After std::remove: 1 2 5 9 10
```

## b. std::unique

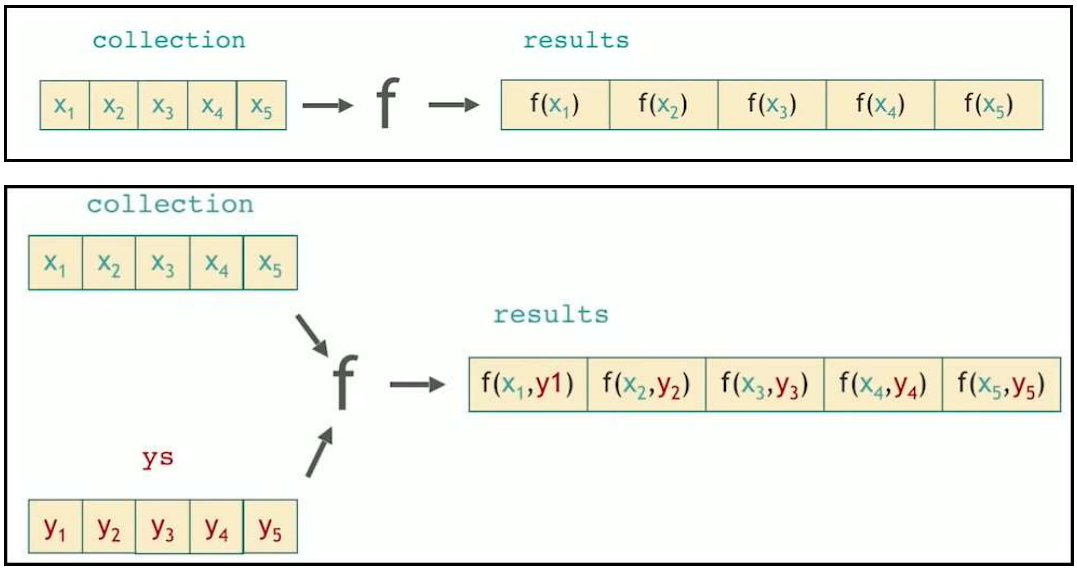
- `std::unique` is used to remove duplicates of any element present consecutively in a range`[first, last)`.
- It does not delete all the duplicate elements, but it removes duplicate by just replacing those elements by the next element present in the sequence which is not duplicate to the current element being replaced. All the elements which are replaced are left in an unspecified state.
- Another interesting feature of this function is that it does not changes the size of the container after deleting the elements, it just returns a pointer pointing to the new end of the container, and based upon that we have to resize the container, or remove the garbage elements.




```
int main()
{
  vector vec {1,2,3,3,9,3,3,3,1,9,10};
  cout << "Original Vector: ";
  for_each(vec.begin(), vec.end(), [](int i){ cout << i << " ";});
  auto it  = unique(vec.begin(), vec.end());
  cout << "\nAfter std::unique: ";
  for_each(vec.begin(), it, [](int i){ cout << i << " ";});
  return 0;
}
// Original Vector: 1 2 3 3 9 3 3 3 1 9 10
// After std::unique: 1 2 3 9 3 1 9 10
```

## 12. Transform

- Transform applies a function to the elements of the collection and produces the output.




Example 1:

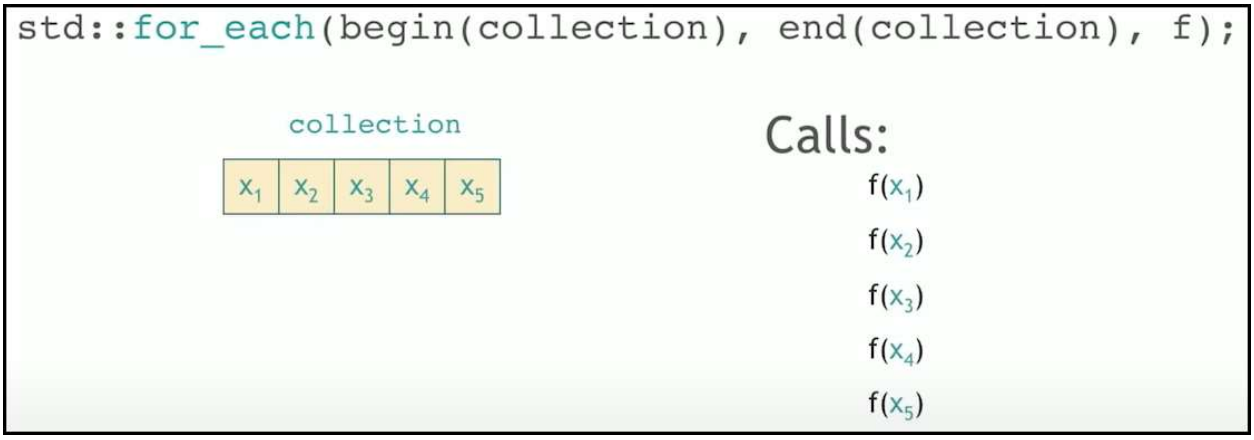
```
int main()
{
  string s = "hello world";
  string d(s.length(), 'X');
  cout << "Source String: " << s << endl;
  transform(s.begin(), s.end(), d.begin(), [](char i){ return toupper(i);});
  cout << "Destination String: " << d << endl;
  return 0;
}
// Source String: hello world
// Destination String: HELLO WORLD
```

Example 2:

```
int main()
{
  vector<int> a {1,2,3,4,5,6,7,8,9};
  vector<int> b {9,8,7,6,5,4,3,2,1};
  vector<int> c(9);
  transform(a.begin(), a.end(),b.begin(),c.begin(), [](int x, int y) { return x + y;});
  cout << "Sum of individual elements of both vectors: ";
  for_each(c.begin(),c.end(),[](int x) { cout << x << " "; });
  return 0;
}
// Sum of individual elements of both vectors: 10 10 10 10 10 10 10 10 10
```

## 13. For_Each

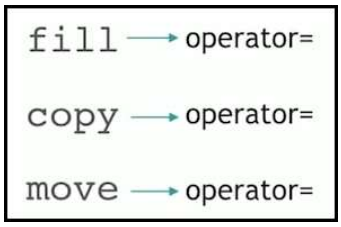
- Applies function to each of the elements in the range `[first,last)`.
- `std::for_each` ignores the `return` value of the function and guarantees order of execution.
- `std::transform` assigns the `return` value to the iterator, and does not guarantee the order of execution.

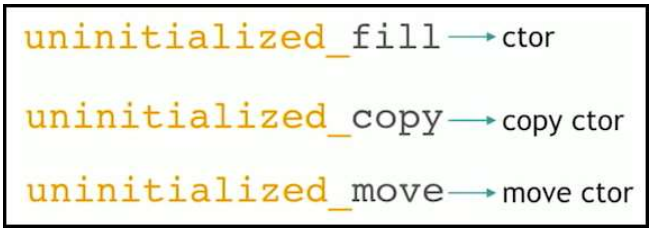



```
void printer(int i) {
        cout << i << " ";
}
int main() {
    int mynumbers[] = { 1, 2, 3, 4 };
    vector<int> v(mynumbers, mynumbers + 4);

    // No effect as returned value of function negate is ignored.
    for_each(v.begin(), v.end(), negate<int>());
    for_each(v.begin(), v.end(), printer);      // Guarantees order

    cout << endl;
    // Negates elements correctly.
    transform(v.begin(), v.end(), v.begin(), negate<int>());
    for_each(v.begin(), v.end(), printer);
    return 0;
}
// 1 2 3 4
// -1 -2 -3 -4
```

## 14. Raw Memory

- The below algorithms uses operator= for the operational purpose.
  - std::fill
  - std::copy
  - std::move
- This means that the object has been constructed.
- But in some rare cases we have the objects that are not constructed yet. We have chunks of memory got from somewhere and we would like to construct the objects.
- For these situations, we use...
  - uninitialized_fill
  - uninitialized_copy
  - uninitialized_move
- These algorithms use respective constructors.




### a. std::uninitialized_fill

- `std::uninitialized_fill` does essentially the same thing as `std::fill`, but it takes a range of memory that has been allocated but not initialized (for instance with operator new, malloc, or a custom memory allocator).
- This algorithm performs the initialization of each element with the passed value, which means that it calls its constructor taking a value of this type.

```
struct MyClass{
  int i;
  explicit MyClass(int ii):i(ii){
    cout << "Constructor\n";
  }
};

int main() {
    MyClass* ptr;
    std::size_t sz;
    std::tie(ptr, sz) = std::get_temporary_buffer<MyClass>(2);
    std::uninitialized_fill(ptr, ptr+sz, 1);
    for (MyClass* x= ptr; x != ptr+sz; ++x) {
      cout << (*x).i <<"\n";
    }
    std::return_temporary_buffer(ptr);
}
// Constructor
// Constructor
// 1
// 1
```

### b. std::uninitialized_copy

- Let's say we have allocated some memory on the heap via malloc and have a pointer T* p to it. We end up with uninitialized storage because all malloc does is mark a location of the size we asked for as allocated (new on the other hand actually constructs objects and thus makes the allocated region initialized storage).
- Since the memory location starting from p does not have a valid object of type T sitting there, we cannot do this…

```
T a;
*p = a;
```

- Since there is no object of type T at p to invoke the assignment operator on. Instead, we will have a construct an object of type T at location p using placement new:

```
T a;
new (p) T{a};
```

- `std::uninitialized_copy` simply implements the range version of the above code snippet when dealing with a range that you want to copy over to uninitialized storage.

```
struct MyClass{
  int i, j;
  MyClass(int ii, int jj):i(ii), j(jj){
    cout << "Constructor\n";
  }
  MyClass(const MyClass& other):i(other.i), j(other.j){
    cout << "Calling Copy Constructor\n";
  }
};


int main()
{
    MyClass* p;
    MyClass obj1(99,88);
    MyClass obj2(99,88);

    vector<MyClass> v;
    v.push_back(obj1);
    v.push_back(obj2);

    std::size_t sz;
    std::tie(p, sz)  = std::get_temporary_buffer<MyClass>(v.size());
    sz = std::min(sz, v.size());
```

```
    std::uninitialized_copy(v.begin(), v.begin() + sz, p);
    for (auto * x = p; x != p+sz; ++x){
      cout << (*x).i << " " << (*x).j << "\n";
    }
    std::return_temporary_buffer(p);
}
// Constructor
// Constructor
// Calling Copy Constructor
// Calling Copy Constructor
// Calling Copy Constructor
// Calling Copy Constructor
// Calling Copy Constructor
// 99 88
// 99 88
```

### c. uninitialized_move

- Similar to uninitialized_copy, but uses move constructor.
- Moves elements from the range `[first, last)` to an uninitialized memory area beginning at `d_first`.
- See the example below `std::destroy`.

### d. std::destroy

- Destroys the objects in the range `[first, last)`.

```
struct MyClass{
  int i, j;
  MyClass(int ii, int jj):i(ii), j(jj){
    cout << "Constructor\n";
  }
  MyClass(const MyClass& other):i(other.i), j(other.j){
    cout << "Calling Copy Constructor\n";
  }
  MyClass(MyClass&& other):i(other.i), j(other.j){
    cout << "Calling Move Constructor\n";
  }
  ~MyClass(){
    cout << "Destructor\n";
  }
};

int main()
{
    MyClass* p;
    vector<MyClass> v { std::move(MyClass(99,88)), std::move(MyClass(99,88))};
```

```
    std::size_t sz;
    std::tie(p, sz)  = std::get_temporary_buffer<MyClass>(v.size());
    sz = std::min(sz, v.size());

    std::uninitialized_move(v.begin(), v.begin() + sz, p);
    for (auto * x = p; x != p+sz; ++x){
      cout << (*x).i << " " << (*x).j << "\n";
    }
    std::destroy(p, p+sz);
    std::return_temporary_buffer(p);
}
// Constructor
// Calling Move Constructor
// Constructor
// Calling Move Constructor
// Calling Copy Constructor
// Calling Copy Constructor
// Destructor
// Destructor
// Destructor
// Destructor
// Calling Move Constructor
// Calling Move Constructor
// 99 88
// 99 88
// Destructor
// Destructor
// Destructor
// Destructor
```

e. std::uninitialized_default_construct

f. std:: uninitialized_value_construct