

# C++ Programming II

C++ 11 Standard Template Library - STL Algorithms  
A Walk-through

*C++ Programming II*  
*October 1, 2018*

Prof. Dr. P. Arnold  
Bern University of Applied Sciences

# Agenda

## ► STL Algorithms

### ► Lambda Expression



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

## ▶ STL Algorithms

▶ Lambda Expression

## ▶ Non-Modifying Algorithms



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

- ▶ **STL Algorithms**
  - ▶ Lambda Expression
- ▶ **Non-Modifying Algorithms**
- ▶ **Modifying Algorithms**



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

- ▶ **STL Algorithms**
  - ▶ Lambda Expression
- ▶ **Non-Modifying Algorithms**
- ▶ **Modifying Algorithms**
- ▶ **Sort Algorithms**



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

- ▶ **STL Algorithms**
  - ▶ Lambda Expression
- ▶ **Non-Modifying Algorithms**
- ▶ **Modifying Algorithms**
- ▶ **Sort Algorithms**
- ▶ **Sorted Data Algorithms**



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

- ▶ **STL Algorithms**
  - ▶ Lambda Expression
- ▶ **Non-Modifying Algorithms**
- ▶ **Modifying Algorithms**
- ▶ **Sort Algorithms**
- ▶ **Sorted Data Algorithms**
- ▶ **Numerical Algorithms**



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms



# STL Algorithms

## STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms



- ▶ Most people use STL-Containers
- ▶ But, most people don't use STL-Algorithms!
- ▶ The goal of this walk-through is to give you an overview of the existing algorithms
- ▶ Later, you may remember existing implementations in STL when facing a coding problem

- ▶ Most people use STL-Containers
- ▶ But, most people don't use STL-Algorithms!
- ▶ The goal of this walk-through is to give you an overview of the existing algorithms
- ▶ Later, you may remember existing implementations in STL when facing a coding problem
- ▶ We'll cover 4 different types of algorithm:
  1. **Non-Modifying** Algorithms
  2. **Modifying** Algorithms
  3. **Sort** Algorithms
  4. **Sorted Data** Algorithms

# Lambda Expression / Function

## What is a Lambda Function

- ▶ During these slides the so called lambda-function are used often
- ▶ Lambda Functions are basically functions without a name!
- ▶ Lambda functions were introduced in C++ 11 and can help in the usage of STL algorithms, *e.g.* to sort or process data

# Lambda Expression / Function

## What is a Lambda Function

- ▶ During these slides the so called lambda-function are used often
- ▶ Lambda Functions are basically functions without a name!
- ▶ Lambda functions were introduced in C++ 11 and can help in the usage of STL algorithms, e.g. to sort or process data

```
1 // C++ 11 Lambda Function: Function without name
2 num = count_if(vec.begin(), vec.end(),
3               [](int x){return x<10;} // lambda expression
4 );
5
6 // same as:
7 bool lessThan10(int x)
8 {
9     return x<10;
10 }
11
12 num = count_if(vec.begin(), vec.end(), lessThan10);
```

1. [ ] - capture clause / lambda introducer: A lambda can introduce new variables in its body (in C++ 14), and it can also access - or capture -variables from the surrounding scope. An empty capture clause, [ ], indicates that the body of the lambda expression accesses no variables in the enclosing scope.
2. ( ) - parameter list (comma separated parameter list)
3. { } - lambda function body (same as function body)

# Lambda Expression

## Defining Lambda Functions on the Fly

- ▶ Defining Lambda with no parameters, returning integer constants:

```
auto just_one ( [] () { return 1; });  
auto just_two ( [] { return 2; }); // () optional  
  
cout << just_one() << ", " << just_two();
```

# Lambda Expression

## Defining Lambda Functions on the Fly

- ▶ Defining Lambda with no parameters, returning integer constants:

```
auto just_one ( [](){ return 1; });  
auto just_two ( [] { return 2; }); // () optional  
  
cout << just_one() << ", " << just_two();
```

- ▶ Defining Lambda with two input parameters, returning an integer

```
auto plus( [](auto l, auto r){ return l + r; });  
  
cout << plus(1, 2) << '\n';  
cout << plus(string{"a"}, "b");
```

# Lambda Expression

## Defining Lambda Functions on the Fly

- ▶ Defining Lambda with no parameters, returning integer constants:

```
auto just_one ( [](){ return 1; });  
auto just_two ( [] { return 2; }); // () optional  
  
cout << just_one() << ", " << just_two();
```

- ▶ Defining Lambda with two input parameters, returning an integer

```
auto plus( [](auto l, auto r){ return l + r; });  
  
cout << plus(1, 2) << '\n';  
cout << plus(string{"a"}, "b");
```

- ▶ Same Lambda, but defined in-place with parameters in () just behind

```
cout << [](auto l, auto r){ return l + r; }(1, 2);
```

- ▶ What's the output?



# Lambda Expression

## Defining Lambda Functions on the Fly

- ▶ Defining Lambda with no parameters, returning integer constants:

```
auto just_one ( [](){ return 1; });  
auto just_two ( [] { return 2; }); // () optional  
  
cout << just_one() << ", " << just_two();
```

- ▶ Defining Lambda with two input parameters, returning an integer

```
auto plus( [](auto l, auto r){ return l + r; });  
  
cout << plus(1, 2) << '\n';  
cout << plus(string{"a"}, "b");
```

- ▶ Same Lambda, but defined in-place with parameters in () just behind

```
cout << [](auto l, auto r){ return l + r; }(1, 2);
```

- ▶ What's the output?

```
1, 2  
3  
ab  
3
```





# Lambda Expression

## Defining Lambda Functions on the Fly

- ▶ Lambda function, carrying a counter initialized to zero with it. In order to modify its own capture, the keyword `mutable` is used:

```
auto counter( [count=0] () mutable { return ++count; });  
  
for (size_t i{0}; i < 5; ++i)  
    cout << counter() << ", ";
```



# Lambda Expression

## Defining Lambda Functions on the Fly

- ▶ Lambda function, carrying a counter initialized to zero with it. In order to modify its own capture, the keyword `mutable` is used:

```
auto counter( [count=0] () mutable { return ++count; });  
  
for (size_t i{0}; i < 5; ++i)  
    cout << counter() << ", ";
```

- ▶ or we can capture values by reference. Like this the values are accessible from outside:

```
int a{0};  
auto incrementer ( [&a] { ++a; } ); // capture by reference  
  
incrementer();  
incrementer();  
incrementer();  
cout << "Value of 'a' after 3 incrementer() calls: " << a;
```

- ▶ What's the output?

# Lambda Expression

## Defining Lambda Functions on the Fly

- ▶ Lambda function, carrying a counter initialized to zero with it. In order to modify its own capture, the keyword `mutable` is used:

```
auto counter( [count=0] () mutable { return ++count; });  
  
for (size_t i{0}; i < 5; ++i)  
    cout << counter() << ", ";
```

- ▶ or we can capture values by reference. Like this the values are accessible from outside:

```
int a{0};  
auto incremter ( [&a] { ++a; } ); // capture by reference  
  
incremter();  
incremter();  
incremter();  
cout << "Value of 'a' after 3 incremter() calls: " << a;
```

- ▶ What's the output?

```
1, 2, 3, 4, 5,  
Value of 'a' after 3 incremter() calls: 3
```

```
[capture list] (parameters)
    mutable                (optional)
    constexpr              (optional)
    exception attr        (optional)
    -> return type         (optional)

{
    body
}
```

### ► Capture list:

- If we write `[=] () { ... }`, we capture every variable the closure references from outside by value, *i.e.* that the values are **copied**
- Writing `[&] () { ... }` means that everything the closure references outside is only captured by reference, which does not lead to a copy

### ► Capture list:

- `[a, &b] () {...}`: This captures `a` by copy and `b` by reference
- `[&, a] () {...}`: This captures `a` by copy and any other used variable by reference
- `[=, &b, i{22}, this] () {...}`: This captures `b` by reference, `this` by copy, initializes a new variable `i` with value 22, and captures any other used variable by copy
- If you want to capture **member variables** you have to capture `this`



- ▶ `mutual` (optional):
  - ▶ If the function object should be able to modify the variables it captures by copy ( `[=]` ), it must be defined mutable. This includes calling non-const methods of captured objects

- ▶ `mutual` (optional):
  - ▶ If the function object should be able to modify the variables it captures by copy ( `[=]` ), it must be defined mutable. This includes calling non-const methods of captured objects
- ▶ `constexpr` (optional):
  - ▶ If marked `constexpr` the compiler will error out if it does not satisfy the criteria of `constexpr` functions. If not explicitly declare the lambda expression to be `constexpr` but it fits the requirements for that, it will be implicitly `constexpr` anyway.

- ▶ `mutable` (optional):
  - ▶ If the function object should be able to modify the variables it captures by copy ( `[=]` ), it must be defined mutable. This includes calling non-const methods of captured objects
- ▶ `constexpr` (optional):
  - ▶ If marked `constexpr` the compiler will error out if it does not satisfy the criteria of `constexpr` functions. If not explicitly declare the lambda expression to be `constexpr` but it fits the requirements for that, it will be implicitly `constexpr` anyway.
- ▶ `exception attr` (optional):
  - ▶ This is the place to specify if the function object can throw exceptions when it's called and runs into an error case.



- ▶ `mutual` (optional):
  - ▶ If the function object should be able to modify the variables it captures by copy ( `[=]` ), it must be defined mutable. This includes calling non-const methods of captured objects
- ▶ `constexpr` (optional):
  - ▶ If marked `constexpr` the compiler will error out if it does not satisfy the criteria of `constexpr` functions. If not explicitly declare the lambda expression to be `constexpr` but it fits the requirements for that, it will be implicitly `constexpr` anyway.
- ▶ `exception attr` (optional):
  - ▶ This is the place to specify if the function object can throw exceptions when it's called and runs into an error case.
- ▶ `return type` (optional):
  - ▶ Allows full control over the return type, we may not want the compiler to deduce it for us automatically. In such a case, we can just write `[] () -> Foo {}` , which tells the compiler that we will really always return the `Foo` type.



# Non-Modifying Algorithms

STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

- ▶ Algorithms not modifying the data
  1. count
  2. min and max
  3. compare
  4. linear search
  5. attribute



- The following variables are used for the examples:

```
1 vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};  
2 vector<int> vec2 = {9,60,70,8,45,87};  
3 vector<int>::iterator itr, itr2;  
4 pair<vector<int>::iterator, vector<int>::iterator> pair_of_itr;
```





### ► Counting, min/max

```
1 vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};
2
3 // 1. Counting
4 // Algorithm Data Operation Result
5 int n = count(vec.begin()+2, vec.end()-1, 69); // 2
6 int m = count_if(vec.begin(), vec.end(),
7                 [](int x){return x==69;}); // 2
8
9 int m = count_if(vec.begin(), vec.end(),
10                 [](int x){return x<10;}); // 3
11
12 // 2. Min and Max
13 itr = max_element(vec.begin()+2, vec.end()); // 90
14 // It returns the first max value
15 itr = max_element(vec.begin(), vec.end(),
16                 [](int x, int y){ return (x%10)<(y%10);}); // 9
17
18
19 // Most algorithms have a simple form and a generalized form
20 // which allow to define own comparison function
21 itr = min_element(vec.begin(), vec.end()); // 7
22 // Generalized form: min_element()
23
24 pair_of_itr = minmax_element(vec.begin(), vec.end(), // {60, 69}
25                             [](int x, int y){ return (x%10)<(y%10);});
26 // returns a pair, which contains first of min and last of max
```

### ► Linear Searching (used when data is not sorted)

```
1 vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};
2
3 // Returns the first match
4 itr = find(vec.begin(), vec.end(), 55);
5
6 itr = find_if(vec.begin(), vec.end(), [](int x){ return x>80; });
7
8 itr = find_if_not(vec.begin(), vec.end(), [](int x){ return
9     x>80; });
10
11 itr = search_n(vec.begin(), vec.end(), 2, 69); // Consecutive 2
12         items of 69
13 // Generalized form: search_n()
14
15 // Search subrange
16 vector<int> sub = {45, 87, 90};
17 itr = search( vec.begin(), vec.end(), sub.begin(), sub.end());
18 // search first subrange
19 itr = find_end( vec.begin(), vec.end(), sub.begin(), sub.end());
20 // search last subrange
21 // Generalized form: search(), find_end()
```



► Linear Searching (used when data is not sorted)

```
1 vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};
2
3 // Search any_of
4 vector<int> items = {87, 69};
5 itr = find_first_of(vec.begin(), vec.end(),
6                     items.begin(), items.end());
7
8 // Search any one of the item in items
9 itr = find_first_of(vec.begin(), vec.end(),
10                    items.begin(), items.end(),
11                    [](int x, int y) { return x==y*4;});
12 // Search any one of the item in items that satisfy: x==y*4;
13
14 // // find two adjacent items that
15 itr = adjacent_find(vec.begin(), vec.end());
16 // are same
17 itr = adjacent_find(vec.begin(), vec.end(),
18                    [](int x, int y){ return x==y*4;});
19 // find two adjacent items that satisfy: x==y*4;
```



### ► Comparing Ranges

```
1 // 4. Comparing Ranges
2 if (equal(vec.begin(), vec.end(), vec2.begin()))
3 {
4     cout << "vec and vec2 are same.\n";
5 }
6
7 if (is_permutation(vec.begin(), vec.end(), vec2.begin()))
8 {
9     cout << "vec and vec2 have same items, but reversed.\n";
10 }
11
12 pair_of_itr = mismatch(vec.begin(), vec.end(), vec2.begin());
13 // find first difference
14 // pair_of_itr.first is an iterator of vec
15 // pair_of_itr.second is an iterator of vec2
16
17 //Lexicographical Comparison: one-by-one comparison with "less
   than"
18 lexicographical_compare(vec.begin(), vec.end(),
19                         vec2.begin(), vec2.end());
20 // {1,2,3,5} < {1,2,4,5}
21 // {1,2}    < {1,2,3}
22
23 // Generalized forms:
24 // equal(), is_permutation(), mismatch(),
   lexicographical_compare()
```





### ► Check Attributes

```
1 is_sorted(vec.begin(), vec.end()); // Check if vec is sorted
2
3 itr = is_sorted_until(vec.begin(), vec.end());
4 // itr points to first place to where elements are no longer
  sorted
5 // Generalized forms: is_sorted(), is_sorted_until()
6
7 is_partitioned(vec.begin(), vec.end(), [](int x){return x>80;} );
8 // Check if vec is partitioned according to condition of (x>80)
9
10 is_heap(vec.begin(), vec.end()); // Check if vec is a heap
11 itr = is_heap_until(vec.begin(), vec.end());
12 // find the first place where it is no longer a heap
13 // Generalized forms: is_heap(), is_heap_until()
```



### ► Check Attributes

```
1 // All, any, none
2 all_of(vec.begin(), vec.end(), [](int x) {return x>80} );
3 // If all of vec is bigger than 80
4
5 any_of(vec.begin(), vec.end(), [](int x) {return x>80} );
6 // If any of vec is bigger than 80
7
8 none_of(vec.begin(), vec.end(), [](int x) {return x>80} );
9 // If none of vec is bigger than 80
```





# Modifying Algorithms

STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

### ► Value changing algorithms

1. copy
2. move
3. transform
4. swap
5. replace
6. fill
7. remove



- The following variables are used for the examples:

```
1 vector<int> vec = {9,60,70,8,45,87,90};           // 7 items
2 vector<int> vec2 = {0,0,0,0,0,0,0,0,0,0,0,0};    // 11 items
3 vector<int>::iterator itr, itr2;
4 pair<vector<int>::iterator, vector<int>::iterator> pair_of_itr;
```



# STL - Algorithms

## Modifying Algorithms I

► copy, copy\_if, copy\_n, copy\_backward

```
1 vector<int> vec = {9,60,70,8,45,87,90}; // 7 items
2 vector<int> vec2 = {0,0,0,0,0,0,0,0,0,0,0,0}; // 11 items
3
4 // Copy
5 copy(vec.begin(), vec.end(), // Source
6      vec2.begin()); // Destination
7
8 copy_if(vec.begin(), vec.end(), // Source
9         vec2.begin(), // Destination
10        [](int x){ return x>80;}); // Condition
11 // vec2: {87, 90, 0, 0, 0, 0, 0, 0, 0, 0, 0}
12
13 copy_n(vec.begin(), 4, vec2.begin());
14 // vec2: {9, 60, 70, 8, 0, 0, 0, 0, 0, 0, 0}
15
16 copy_backward(vec.begin(), vec.end(), // Source
17              vec2.end()); // Destination
18 // vec2: {0, 0, 0, 0, 9, 60, 70, 8, 45, 87, 90}
```



# STL - Algorithms

## Modifying Algorithms I

► move, move\_backward

```
1 // Move
2 vector<string> vec = {"apple", "orange", "pear", "grape"}; // 4 items
3 vector<string> vec2 = {"", "", "", "", "", ""}; // 6 items
4
5 move(vec.begin(), vec.end(), vec2.begin());
6 // vec: {"", "", "", ""} // Undefined
7 // vec2: {"apple", "orange", "pear", "grape", "", ""};
8 //
9 // If move semantics are defined for the element type, elements
10 // are moved over, otherwise they are copied over with copy
11 // constructor, just like copy().
12
13 move_backward(vec.begin(), vec.end(), vec2.end());
14 // vec2: {"", "", "apple", "orange", "pear", "grape"};
```



# STL - Algorithms

## Modifying Algorithms I

### ► transform, swap

```
1 // Transform
2 transform(vec.begin(), vec.end(), // Source
3          vec3.begin(), // Destination
4          [](int x){ return x-1;}); // Operation
5
6 transform(vec.begin(), vec.end(), // Source #1
7          vec2.begin(), // Source #2
8          vec3.begin(), // Destination
9          [](int x, int y){ return x+y;}); // Operation
10
11 // Add items from vec and vec2 and save in vec3
12 // vec3[0] = vec[0] + vec2[0]
13 // vec3[1] = vec[1] + vec2[1]
14 // ...
15
16 // Swap – two way copying
17 swap_ranges(vec.begin(), vec.end(), vec2.begin());
```





# STL - Algorithms

## Modifying Algorithms I

### ► fill

```
1 // Fill
2 vector<int> vec = {0, 0, 0, 0, 0};
3
4 fill(vec.begin(), vec.end(), 9); // vec: {9, 9, 9, 9, 9}
5
6 fill_n(vec.begin(), 3, 9);        // vec: {9, 9, 9, 0, 0}
7
8 generate(vec.begin(), vec.end(), rand);
9
10 generate_n(vec.begin(), 3, rand);
```





# STL - Algorithms

## Modifying Algorithms I

### ► replace

```
1 // 6. Replace
2 replace(vec.begin(), vec.end(), // Data Range
3       6, // Old value condition
4       9); // new value
5
6 replace_if(vec.begin(), vec.end(), // Data Range
7           [] (int x) {return x>80;}, // Old value condition
8           9); // new value
9
10 replace_copy(vec.begin(), vec.end(), // Source
11             vec2.begin(), // Destination
12             6, // Old value condition
13             9); // new value
14 // Generalized form: replace_copy_if()
```



# STL - Algorithms

## Modifying Algorithms I

► remove, remove\_if, remove\_copy, unique, unique\_copy

```
1 // Remove
2 remove(vec.begin(), vec.end(), 3); // Remove all 3's
3 remove_if(vec.begin(), vec.end(), [](int x){return x>80;});
4 // Remove items bigger than 80
5
6 remove_copy(vec.begin(), vec.end(), // Source
7             vec2.begin(),           // Destination
8             6);                      // Condition
9 // Remove all 6's, and copy the remain items to vec2
10 // Generalized form: remove_copy_if()
11
12 unique(vec.begin(), vec.end()); // Remove consecutive equal elems
13
14 unique(vec.begin(), vec.end(), less<int>());
15 // Remove elements whose previous element is less than itself
16
17 unique_copy(vec.begin(), vec.end(), vec2.begin());
18 // Remove consecutive equal elements, and then copy the
19 // items to vec2
20 // Generalized form: unique_copy()
```



► Order changing algorithms

1. reverse
2. rotate
3. permute
4. shuffle



- The following variables are used for the examples:

```
1 vector<int> vec = {9,60,70,8,45,87,90};           // 7 items
2 vector<int> vec2 = {0,0,0,0,0,0,0};              // 7 items
```



- ▶ reverse, reverse\_copy, rotate, copy\_backward

```
1  vector<int> vec = {9,60,70,8,45,87,90};           // 7 items
2  vector<int> vec2 = {0,0,0,0,0,0,0};              // 7 items
3
4  // Reverse
5  reverse(vec.begin()+1, vec.end()-1);
6  // vec: {9,87,45,8,70,60,90};           // 7 items
7
8  reverse_copy(vec.begin()+1, vec.end()-1, vec2.begin());
9  // vec2: {87,45,8,70,60,0,0};
10
11 // Rotate
12 rotate(vec.begin(), vec.begin()+3, vec.end());
13 // vec: {8,45,87,90,9,60,70};           // 7 items
14
15 rotate_copy(vec.begin(), vec.begin()+3, vec.end(), // Source
16            vec2.begin());                          // Destination
17 // Copy vec to vec2 in rotated order
18 // vec is unchanged
```



### ► permute

```
1 vector<int> vec = {9,60,70,8,45,87,90}; // 7 items
2 vector<int> vec2 = {0,0,0,0,0,0,0}; // 7 items
3
4 next_permutation(vec.begin(), vec.end());
5 // Lexicographically next greater permutation
6 prev_permutation(vec.begin(), vec.end());
7 // Lexicographically next smaller permutation
8 // Exmple with 1, 2 and 3
9 // 3 2 1
10 // 3 1 2
11 // 2 3 1
12 // 2 1 3
13 // 1 3 2
14 // 1 2 3
```



### ► shuffle

```
1 // Shuffle
2 // Rearrange the elements randomly
3 // (swap each element with a randomly selected element)
4 random_shuffle(vec.begin(), vec.end());
5 random_shuffle(vec.begin(), vec.end(), rand);
6
7 // C++ 11
8 shuffle(vec.begin(), vec.end(), default_random_engine());
9 // Better random number generation
```







# Sort Algorithms

STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

- ▶ Sorting algorithms requires random access iterators!
- ▶ Sorting is limited to:  
vector, deque, container array, native array
- ▶ Sorting algorithms
  1. sort, partial\_sort
  2. nth\_element
  3. partition, stable\_partition



### ► sort

```
1 vector<int> vec = {9,1,10,2,45,3,90,4,9,5,8};
2
3 sort(vec.begin(), vec.end()); // sort with operator <
4 // vec: 1 2 3 4 5 8 9 9 10 45 90
5
6
7
8 bool lsb_less(int x, int y) {
9     return (x%10)<(y%10);
10 }
11 sort(vec.begin(), vec.end(), lsb_less); // sort with lsb_less()
12 // vec: 10 90 1 2 3 4 45 5 8 9 9
```



### ► `partial_sort`

```
1 // Sometime we don't need complete sorting.
2
3 // Problem #1:
4 // Finding top 5 students according to their test scores.
5 // - partial sort
6 vector<int> vec = {9,60,70,8,45,87,90,69,69,55,7};
7
8 partial_sort(vec.begin(), vec.begin()+5, vec.end(),
9             greater<int>());
10 // vec: 90 87 70 69 69 8 9 45 60 55 7
11
12 // Overloaded:
13 partial_sort(vec.begin(), vec.begin()+5, vec.end());
14 // vec: 7 8 9 45 55 90 60 87 70 69 69
```



### ► nth\_element

```
1 // Problem #2:  
2 // Finding top 5 students according to their score, but order is  
3 // not important  
4  
5 vector<int> vec = {9,60,70,8,45,87,90,69,69,55,7};  
6  
7 nth_element(vec.begin(), vec.begin() + 5, vec.end(),  
8             greater<int>());  
9 // vec: 69 87 70 90 69 60 55 45 9 8 7
```



### ► partition, stable\_partition

```
1 // Problem #3:
2 // Move the students whose score is less than 10 to the front
3 vector<int> vec = {9,60,70,8,45,87,90,69,69,55,7};
4
5 bool lessThan10(int i) {
6     return (i<10);
7 }
8 partition(vec.begin(), vec.end(), lessThan10);
9 // vec: 8 7 9 90 69 60 55 45 70 87 69
10
11 // To preserve the original order within each partition:
12 stable_partition(vec.begin(), vec.end(), lessThan10);
13 // vec: 9 8 7 60 70 45 87 90 69 69 55
```





# Sorted Data Algorithms



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

# STL - Algorithms

## Sorted Data Algorithms

- ▶ Algorithms that require data being pre-sorted
- ▶ Algorithms
  1. `binary_search`
  2. `merge`
  3. `set operations`





# STL - Algorithms

## Sorted Data Algorithms

- ▶ `binary_search`, `includes`, `lower_bound`, `upper_bound`, `equal_range`

```
1 vector<int> vec = {8,9,9,9,45,87,90};           // 7 items
2
3 // Binary Search
4
5 // Search Elements
6 bool found = binary_search(vec.begin(), vec.end(), 9);
7 // check if 9 is in vec
8
9 vector<int> s = {9, 45, 66};
10 bool found = includes(vec.begin(), vec.end(),           // Range #1
11                    s.begin(), s.end());                 // Range #2
12
13 // Return true if all elements of s is included in vec
14 // Both vec and s must be sorted
15
16 // Search Position
17 itr = lower_bound(vec.begin(), vec.end(), 9); // vec[1]
18 // Find the first position where 9 could be inserted and
19 // keep the sorting.
20
21 itr = upper_bound(vec.begin(), vec.end(), 9); // vec[4]
22 // Find the last position where 9 could be inserted and
23 // keep the sorting.
24
25 pair_of_itr = equal_range(vec.begin(), vec.end(), 9);
26 // Returns both first and last position
```



# STL - Algorithms

## Sorted Data Algorithms

### ► merge, inplace\_merge

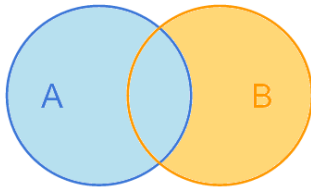
```
1 // Merge
2 vector<int> vec = {8,9,9,10};
3 vector<int> vec2 = {7,9,10};
4
5 merge(vec.begin(), vec.end(),           // Input Range #1
6       vec2.begin(), vec2.end(),         // input Range #2
7       vec_out.begin());                 // Output
8
9 // Both vec and vec2 should be sorted (same for the set
10 // operation)
11 // Nothing is dropped, all duplicates are kept.
12 // vec_out: {7,8,9,9,9,10,10}
13
14 // Both parts of vec are already sorted, i.e. 0-3, 4-8 are two
15 // sorted groups of data in vec
16 vector<int> vec = {1,2,3,4,1,2,3,4,5};
17 inplace_merge(vec.begin(), vec.begin()+4, vec.end());
18 // vec: {1,1,2,2,3,3,4,4,5} - One step of merge sort
```



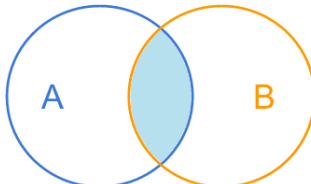
# STL - Algorithms

## Sorted Data Algorithms

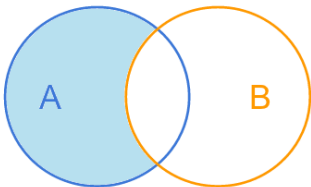
### ► Algorithms on set



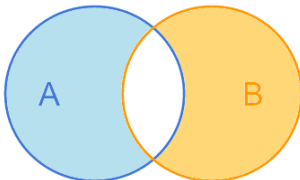
`std::set_union`



`std::set_intersection`



`std::set_difference`



`std::set_symmetric_difference`



# STL - Algorithms

## Sorted Data Algorithms

### ► set\_union, set\_intersection

```
1 // Set operations
2 // - Both vec and vec3 should be sorted
3 // - The resulted data is also sorted
4 vector<int> vec = {8,9,9,10};
5 vector<int> vec2 = {7,9,10};
6 vector<int> vec_out[5];
7 set_union(vec.begin(), vec.end(), // Input Range #1
8           vec2.begin(), vec2.end(), // input Range #2
9           vec_out.begin()); // Output
10 // if X is in both vec and vec2, only one X is kept in vec_out
11 // vec_out: {7,8,9,9,10}
12
13 set_intersection(vec.begin(), vec.end(), // Input Range #1
14                 vec2.begin(), vec2.end(), // input Range #2
15                 vec_out.begin()); // Output
16 // Only the items that are in both vec and vec2 are saved
17 // in vec_out
18 // vec_out: {9,10,0,0,0}
```



# STL - Algorithms

## Sorted Data Algorithms

### ► set\_difference, set\_symmetric\_difference

```
1 vector<int> vec = {8,9,9,10};
2 vector<int> vec2 = {7,9,10};
3 vector<int> vec_out[5];
4 set_difference(vec.begin(), vec.end(),           // In Range #1
5               vec2.begin(), vec2.end(),         // In Range #2
6               vec_out.begin());                 // Output
7 // Only the items that are in vec but not in vec2 are saved
8 // in vec_out
9 // vec_out: {8,9,0,0,0}
10
11 set_symmetric_difference(vec.begin(), vec.end(), // In Range #1
12                          vec2.begin(), vec2.end(), // In Range #2
13                          vec_out.begin());        // Output
14 // vec_out has items from either vec or vec2, but not from both
15 // vec_out: {7,8,9,0,0}
```

STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms



# Numerical Algorithms

STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

# STL - Algorithms

## Numerical Algorithms

- ▶ Numerical Algorithms are in `<numeric>`
- ▶ Algorithms
  1. Accumulate
  2. Inner product
  3. Partial sum
  4. Adjacent difference

# STL - Algorithms

## Numerical Algorithms

### ► accumulate

```
1 // Accumulate
2
3 int x = accumulate(vec.begin(), vec.end(), 10);
4 // 10 + vec[0] + vec[1] + vec[2] + ...
5 // 10 is initial value
6
7 x = accumulate(vec.begin(), vec.end(), 10, multiplies<int>());
8 // 10 * vec[0] * vec[1] * vec[2] * ...
```







### ► inner\_product

```
1 // Inner Product
2
3 vector<int> vec = {9,60,70,8,45,87,90}; // 7 items
4
5 int x = inner_product(vec.begin(), vec.begin()+3, // Range #1
6                       vec.end()-3, // Range #2
7                       10); // Init Value
8 // 10 + vec[0]*vec[4] + vec[2]*vec[5] + vec[3]*vec[6]
9
10 int x = inner_product(vec.begin(), vec.begin()+3, // Range #1
11                       vec.end()-3, // Range #2
12                       10, // Init Value
13                       multiplies<int>(),
14                       plus<int>());
15 // 10 * (vec[0]+vec[4]) * (vec[2]+vec[5]) * (vec[3]+vec[6])
```

STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

► `partial_sum`, `adjacent_difference`

```
1 // Partial Sum
2 partial_sum(vec.begin(), vec.end(), vec2.begin());
3 // vec2[0] = vec[0]
4 // vec2[1] = vec[0] + vec[1];
5 // vec2[2] = vec[0] + vec[1] + vec[2];
6 // vec2[3] = vec[0] + vec[1] + vec[2] + vec[3];
7 // ...
8
9 partial_sum(vec.begin(), vec.end(),
10            vec2.begin(), multiplies<int>());
11
12
13 // Adjacent Difference
14 adjacent_difference(vec.begin(), vec.end(), vec2.begin());
15 // vec2[0] = vec[0]
16 // vec2[1] = vec[1] - vec[0];
17 // vec2[2] = vec[2] - vec[1];
18 // vec2[3] = vec[3] - vec[2];
19 // ...
20
21 adjacent_difference(vec.begin(), vec.end(),
22                    vec2.begin(), plus<int>());
```

STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms

# Thank You

## Questions

???



STL Algorithms

Lambda Expression

Non-Modifying  
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data  
Algorithms

Numerical Algorithms