C++ Programming II

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

C++ Programming II
September 24, 2018

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

Agenda

▶ **STL Algorithms**
  ▶ Lambda Expression

Agenda

▶ **STL Algorithms**
    ▶ Lambda Expression

▶ **Non-Modifying Algorithms**

Agenda

▶ **STL Algorithms**
  ▶ Lambda Expression

▶ **Non-Modifying Algorithms**

▶ **Modifying Algorithms**

Agenda

▶ **STL Algorithms**
  ▶ Lambda Expression

▶ **Non-Modifying Algorithms**

▶ **Modifying Algorithms**

▶ **Sort Algorithms**

Agenda

▶ **STL Algorithms**
  ▶ Lambda Expression

▶ **Non-Modifying Algorithms**

▶ **Modifying Algorithms**

▶ **Sort Algorithms**

▶ **Sorted Data Algorithms**

Agenda

► **STL Algorithms**
  ► Lambda Expression

► **Non-Modifying Algorithms**

► **Modifying Algorithms**

► **Sort Algorithms**

► **Sorted Data Algorithms**

► **Numerical Algorithms**

Bern University
of Applied Sciences

STL Algorithms

STL - Algorithms
Intro

- ▶ 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

STL - Algorithms
Intro

- ▶ 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:

```cpp
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:

```cpp
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

```cpp
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:

```cpp
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

```cpp
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

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

▶ **What's the output?**

Bern University
of Applied Sciences

STL Algorithms

Lambda Expression

Non-Modifying
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data
Algorithms

Numerical Algorithms

Lambda Expression
Defining Lambda Functions on the Fly

► Defining Lambda with no parameters, returning integer constants:

```cpp
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

```cpp
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

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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?**

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

Lambda Expression
All Parameters at Once

```
[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

Lambda Expression
Capture List Examples

Lecture 4

Prof. Dr. P. Arnold

F
H

Bern University
of Applied Sciences

STL Algorithms

Lambda Expression

Non-Modifying
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data
Algorithms

Numerical Algorithms

▶ **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`

Lambda Expression
Optional Parameters

- `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

Lambda Expression
Optional Parameters

- ▶ `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.

Lambda Expression
Optional Parameters

▶ `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.

Lambda Expression
Optional Parameters

- ▶ `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.

Bern University
of Applied Sciences

Non-Modifying Algorithms

STL - Algorithms
Non-Modifying Algorithms

▶ Algorithms not modifying the data

1. count
2. min and max
3. compare
4. linear search
5. attribute

STL - Algorithms
Non-Modifying Algorithms

▶ 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;
```

STL - Algorithms
Non-Modifying Algorithms

► Counting, min/max

```cpp
vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};

// 1. Counting
//     Algorithm    Data                    Operation        Result
int n = count(vec.begin()+2, vec.end()-1, 69);          // 2
int m = count_if(vec.begin(), vec.end(),
                 [](int x){return x==69;});              // 2

int m = count_if(vec.begin(), vec.end(),
                 [](int x){return x<10;});               // 3

// 2. Min and Max
itr = max_element(vec.begin()+2, vec.end());            // 90
// It returns the first max value
itr = max_element(vec.begin(), vec.end(),
                 [](int x, int y){ return (x%10)<(y%10);});  // 9


// Most algorithms have a simple form and a generalized form
// which allow to define own comparison function
itr = min_element(vec.begin(), vec.end());              // 7
// Generalized form: min_element()

pair_of_itr = minmax_element(vec.begin(), vec.end(),    // {60, 69}
                 [](int x, int y){ return (x%10)<(y%10);});
// returns a pair, which contains first of min and last of max
```

STL - Algorithms
Non-Modifying Algorithms

▶ Linear Searching (used when data is not sorted)

```cpp
vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};

// Returns the first match
itr = find(vec.begin(), vec.end(), 55);

itr = find_if(vec.begin(), vec.end(), [](int x){ return x>80; });

itr = find_if_not(vec.begin(), vec.end(), [](int x){ return
    x>80; });

itr = search_n(vec.begin(), vec.end(), 2, 69);  // Consecutive 2
    items of 69
// Generalized form: search_n()

// Search subrange
vector<int> sub = {45, 87, 90};
itr = search( vec.begin(), vec.end(), sub.begin(), sub.end());
// search first subrange
itr = find_end( vec.begin(), vec.end(), sub.begin(), sub.end());
// search last subrange
// Generalized form: search(), find_end()
```

STL - Algorithms
Non-Modifying Algorithms

▶ Linear Searching (used when data is not sorted)

```cpp
vector<int> vec = {9,60,90,8,45,87,90,69,69,55,7};

// Search any_of
vector<int> items = {87, 69};
itr = find_first_of(vec.begin(), vec.end(),
                    items.begin(), items.end());

// Search any one of the item in items
itr = find_first_of(vec.begin(), vec.end(),
                    items.begin(), items.end(),
                    [](int x, int y) { return x==y*4;});
// Search any one of the item in items that satisfy: x==y*4;

// // find two adjacent items that
itr = adjacent_find(vec.begin(), vec.end());
// are same
itr = adjacent_find(vec.begin(), vec.end(),
                    [](int x, int y){ return x==y*4;});
// find two adjacent items that satisfy: x==y*4;
```

STL - Algorithms
Non-Modifying Algorithms

► Comparing Ranges

```
// 4. Comparing Ranges
if (equal(vec.begin(), vec.end(), vec2.begin()))
{
    cout << "vec and vec2 are same.\n";
}

if (is_permutation(vec.begin(), vec.end(), vec2.begin()))
{
    cout << "vec and vec2 have same items, but reversed.\n";
}

pair_of_itr = mismatch(vec.begin(), vec.end(), vec2.begin());
// find first difference
// pair_of_itr.first is an iterator of vec
// pair_of_itr.second is an iterator of vec2

// Lexicographical Comparison: one-by-one comparison with "less
    than"
lexicographical_compare(vec.begin(), vec.end(),
                        vec2.begin(), vec2.end());
// {1,2,3,5} < {1,2,4,5}
// {1,2}     < {1,2,3}

// Generalized forms:
// equal(), is_permutation(), mismatch(),
    lexicographical_compare()
```

STL - Algorithms
Non-Modifying Algorithms

► 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()
```

STL - Algorithms
Non-Modifying Algorithms

► 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
Modifying Algorithms I

► Value changing algorithms

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

STL - Algorithms
Modifying Algorithms I

► The following variables are used for the examples:

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

```
// Move
vector<string> vec = {"apple","orange","pear","grape"}; // 4 items
vector<string> vec2 = {"", "", "", "", "", ""};          // 6 items

move(vec.begin(), vec.end(), vec2.begin());
// vec:  {"", "", "", ""}  // Undefined
// vec2: {"apple", "orange", "pear", "grape", "", ""};
//
// If move semantics are defined for the element type, elements
// are moved over, otherwise they are copied over with copy
// constructor, just like copy().

move_backward(vec.begin(), vec.end(), vec2.end());
// 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

```cpp
// Fill
vector<int> vec = {0, 0, 0, 0, 0};

fill(vec.begin(), vec.end(), 9);    // vec: {9, 9, 9, 9, 9}

fill_n(vec.begin(), 3, 9);          // vec: {9, 9, 9, 0, 0}

generate(vec.begin(), vec.end(), rand);

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()
```

STL - Algorithms
Modifying Algorithms II

▶ Order changing algorithms

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

STL - Algorithms
Modifying Algorithms II

Lecture 4

Prof. Dr. P. Arnold

Bern University
of Applied Sciences

STL Algorithms
  Lambda Expression

Non-Modifying
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data
Algorithms

Numerical Algorithms

► 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
```

STL - Algorithms
Modifying Algorithms II

▶ `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
```

STL - Algorithms
Modifying Algorithms II

▶ permute

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

next_permutation(vec.begin(), vec.end());
// Lexicographically next greater permutation
prev_permutation(vec.begin(), vec.end());
// Lexicographically next smaller permutation
// {1,2,3,5} < {1,2,4,4}
// {1,2}    < {1,2,3}

//Sorted in ascending order:  {8, 9, 45, 60, 70, 87, 90}
//                              - Lexicographically smallest
//
//Sorted in descending order: {90, 87, 70, 60, 45, 9, 8}
//                              - Lexicographically greatest

// Generalized form: next_permutation(), prev_permutation()
```

STL - Algorithms
Modifying Algorithms II

▶ 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
Modifying Algorithms II

► 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
4. heap

STL - Algorithms
Modifying Algorithms II

▶ 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
```

STL - Algorithms
Modifying Algorithms II

▶ `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(),
          greater<int>());
9   // vec: 90 87 70 69 69 8 9 45 60 55 7
10
11  // Overloaded:
12  partial_sort(vec.begin(), vec.begin()+5, vec.end());
13  // vec: 7 8 9 45 55 90 60 87 70 69 69
```

STL - Algorithms
Modifying Algorithms II

Lecture 4

Prof. Dr. P. Arnold

Bern University
of Applied Sciences

STL Algorithms
  Lambda Expression

Non-Modifying
Algorithms

Modifying Algorithms

Sort Algorithms

Sorted Data
Algorithms

Numerical Algorithms

▶ nth_element

```
// Problem #2:
// Finding top 5 students according to their score, but order is
// not important

vector<int> vec = {9,60,70,8,45,87,90,69,69,55,7};

nth_element(vec.begin(), vec.begin()+5, vec.end(),
      greater<int>());
// vec: 69 87 70 90 69 60 55 45 9 8 7
```

STL - Algorithms
Modifying Algorithms II

▶ `partition`, `stable_partition`

```cpp
// Problem #3:
// Move the students whose score is less than 10 to the front
vector<int> vec = {9,60,70,8,45,87,90,69,69,55,7};

bool lessThan10(int i) {
    return (i<10);
}
partition(vec.begin(), vec.end(), lessThan10);
// vec: 8 7 9 90 69 60 55 45 70 87 69

// To preserve the original order within each partition:
stable_partition(vec.begin(), vec.end(), lessThan10);
// vec: 9 8 7 60 70 45 87 90 69 69 55
```

Sorted Data 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`

```cpp
vector<int> vec = {8,9,9,9,45,87,90};          // 7 items

// Binary Search

// Search Elements
bool found = binary_search(vec.begin(), vec.end(), 9);
// check if 9 is in vec

vector<int> s = {9, 45, 66};
bool found = includes(vec.begin(), vec.end(),       // Range #1
                      s.begin(), s.end());          // Range #2

// Return true if all elements of s is included in vec
// Both vec and s must be sorted

// Search Position
itr = lower_bound(vec.begin(), vec.end(), 9);   // vec[1]
// Find the first position where 9 could be inserted and
// keep the sorting.

itr = upper_bound(vec.begin(), vec.end(), 9);   // vec[4]
// Find the last position where 9 could be inserted and
// keep the sorting.

pair_of_itr = equal_range(vec.begin(), vec.end(), 9);
// Returns both first and last position
```

STL - Algorithms
Sorted Data Algorithms

▶ `merge`, `inplace_merge`

```cpp
// Merge
vector<int> vec = {8,9,9,10};
vector<int> vec2 = {7,9,10};

merge(vec.begin(), vec.end(),        // Input Range #1
      vec2.begin(), vec2.end(),      // input Range #2
      vec_out.begin());              // Output

// Both vec and vec2 should be sorted (same for the set
      operation)
// Nothing is dropped, all duplicates are kept.
// vec_out: {7,8,9,9,9,10,10}

// Both parts of vec are already sorted, i.e. 0-3, 4-8 are two
// sorted groups of data in vec
vector<int> vec = {1,2,3,4,1,2,3,4,5};
inplace_merge(vec.begin(), vec.begin()+4, vec.end());
// vec: {1,1,2,2,3,3,4,4,5} - One step of merge sort
```

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}
```

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] * ...
```

STL - Algorithms
Numerical Algorithms

Lecture 4

Prof. Dr. P. Arnold

Bern University
of Applied Sciences

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

▶ `inner_product`

```cpp
// Inner Product

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

int x = inner_product(vec.begin(), vec.begin()+3,  // Range #1
                      vec.end()-3,                 // Range #2
                      10);                          // Init Value
// 10 + vec[0]*vec[4] + vec[2]*vec[5] + vec[3]*vec[6]

int x = inner_product(vec.begin(), vec.begin()+3,  // Range #1
                      vec.end()-3,                 // Range #2
                      10,                           // Init Value
                      multiplies<int>(),
                      plus<int>());
// 10 * (vec[0]+vec[4]) * (vec[2]+vec[5]) * (vec[3]+vec[6])
```

STL - Algorithms
Numerical Algorithms

▶ `partial_sum`, `adjacent_difference`

```cpp
// Partial Sum
partial_sum(vec.begin(), vec.end(), vec2.begin());
// vec2[0] = vec[0]
// vec2[1] = vec[0] + vec[1];
// vec2[2] = vec[0] + vec[1] + vec[2];
// vec2[3] = vec[0] + vec[1] + vec[2] + vec[3];
// ...

partial_sum(vec.begin(), vec.end(),
            vec2.begin(), multiplies<int>());


// Adjacent Difference
adjacent_difference(vec.begin(), vec.end(), vec2.begin());
// vec2[0] = vec[0]
// vec2[1] = vec[1] - vec[0];
// vec2[2] = vec[2] - vec[1];
// vec2[3] = vec[3] - vec[2];
// ...

adjacent_difference(vec.begin(), vec.end(),
                    vec2.begin(), plus<int>());
```

Thank You
Questions

???