

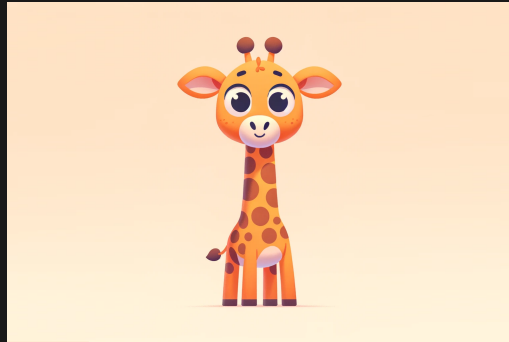
COMP6771



STL Algorithms

Lecture 2.3

Author(s): Hayden Smith



[\(Download as PDF\)](#)

STL Algorithms

- STL Algorithms are functions that execute an algorithm on an abstract notion of an iterator.
- In this way, they can work on a number of containers as long as those containers can be represented via a relevant iterator.



Simple Example

What's the best way to sum a vector of numbers?

C-Style?

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> nums { 1, 2, 3, 4, 5 };
7
8     auto sum = 0;
9     for (auto i = 0; i <= static_cast<int>(nums.size()); ++i) {
10         sum += i;
11     }
12
13     std::cout << sum << "\n";
14 }
```

c-sum.cpp



Simple Example

What's the best way to sum a vector of numbers?

Via an iterator? Or for-range?

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> nums { 1, 2, 3, 4, 5 };
7
8     auto sum = 0;
9     for (auto it = nums.begin(); it != nums.end(); ++it) {
10         sum += *it;
11     }
12     std::cout << sum << "\n";
13 }
```

simple-sum1.cpp

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> nums { 1, 2, 3, 4, 5 };
7
8     int sum = 0;
9
10    // Internally, this uses begin and end,
11    // but it abstracts it away.
12    for (const auto& i : nums) {
13        sum += i;
14    }
15
16    std::cout << sum << "\n";
17 }
```

simple-sum2.cpp



Simple Example

What's the best way to sum a vector of numbers?

Via use of an STL Algorithm

```
1 #include <iostream>
2 #include <numeric>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<int> nums { 1, 2, 3, 4, 5 };
8     int sum = std::accumulate(nums.begin(), nums.end(), 0);
9     std::cout << sum << "\n";
10 }
```

sum-stl.cpp



Simple Example

This is what goes on under the hood

```
1 /*
2  template <typename T, typename Container>
3  T sum(iterator_t<Container> first, iterator_t<Container> last) {
4      T total;
5      for (; first != last; ++first) {
6          total += *first;
7      }
8      return total
9  }
10 */
```

[sum-stl-underlying.cpp](#)

More Examples

We can also use algorithms to:

- Find the product instead of the sum
- Sum only the first half of elements

```
1 #include <iostream>
2 #include <numeric>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<int> v { 1, 2, 3, 4, 5 };
8     int sum = std::accumulate(v.begin(), v.end(), 0);
9
10    // What is the type of std::multiplies<int>()
11    int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());
12    (void)product; // dummy line
13
14    auto midpoint = v.begin() + static_cast<int>(v.size() / 2);
15    // This looks a lot harder to read. Why might it be better?
16
17    auto midpoint11 = std::next(v.begin(), std::distance(v.begin(), v.end()) / 2);
18    (void)midpoint11;
19
20    int sum2 = std::accumulate(v.begin(), midpoint, 0);
21
22    std::cout << sum << " " << sum2 << "\n";
23 }
```

algos.cpp

More Examples

We can also use algorithms to:

- Check if an element exists

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> nums { 1, 2, 3, 4, 5 };
7
8     auto it = std::find(nums.begin(), nums.end(), 4);
9
10    if (it != nums.end()) {
11        std::cout << "Found it!"
12                  << "\n";
13    }
14 }
```

find.cpp



Performance & Portability

- Consider:
 - Number of comparisons for binary search on a vector is $O(\log N)$
 - Number of comparisons for binary search on a linked list is $O(N \log N)$
 - The two implementations are completely different
- We can call the same function on both of them
 - It will end up calling a function have two different overloads, one for a forward iterator, and one for a random access iterator
- Trivial to read
- Trivial to change the type of a container

```
1 #include <algorithm>
2 #include <iostream>
3 #include <list>
4 #include <vector>
5
6 int main()
7 {
8     // Lower bound does a binary search, and returns the first value >= the argument.
9     std::vector<int> sortedVec { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10    (void)std::lower_bound(sortedVec.begin(), sortedVec.end(), 5);
11
12    std::list<int> sortedLinkedList { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13    (void)std::lower_bound(sortedLinkedList.begin(), sortedLinkedList.end(), 5);
14 }
```

bound.cpp



Output Sequence Algorithms

```
1 #include <iostream>
2 #include <vector>
3
4 char to_upper(unsigned char value)
5 {
6     return static_cast<char>(std::toupper(static_cast<unsigned char>(value)));
7 }
8
9 int main()
10 {
11
12     std::string s = "hello world";
13     // Algorithms like transform, which have output iterators,
14     // use the other iterator as an output.
15     auto upper = std::string(s.size(), '\0');
16     std::transform(s.begin(), s.end(), upper.begin(), to_upper);
17 }
```

transform.cpp



Back Inserter

Gives you an output iterator for a container that adds to the end of it

```
1 #include <iostream>
2 #include <vector>
3
4 char to_upper(char value)
5 {
6     return static_cast<char>(std::toupper(static_cast<unsigned char>(value)));
7 }
8
9 int main()
10 {
11     std::string s = "hello world";
12     // std::for_each modifies each element
13     std::for_each(s.begin(), s.end(), to_upper);
14
15     std::string upper;
16     // std::transform adds to third iterator.
17     std::transform(s.begin(), s.end(), std::back_inserter(upper), to_upper);
18 }
19 }
```

inserter.cpp



Lambda Functions

- A function that can be defined inside other functions
- Can be used with `std::function<ReturnType(Arg1, Arg2)>` (or `auto`)
 - It can be used as a parameter or variable
 - No need to use function pointers anymore

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::string s = "hello world";
7     // std::for_each modifies each element
8     std::for_each(s.begin(), s.end(), [](char& value) { value = static_cast<char>(std::toupper(value)); });
9 }
```

lambda1.cpp



Lambda Functions

- Anatomy of a lambda function
- Lambdas can be defined anonymously, or they can be stored in a variable

```
1 [](card const c) -> bool {  
2     return c.colour == 4;  
3 }
```

```
1 [capture] (parameters) -> return {  
2     body  
3 }
```



Lambda Functions

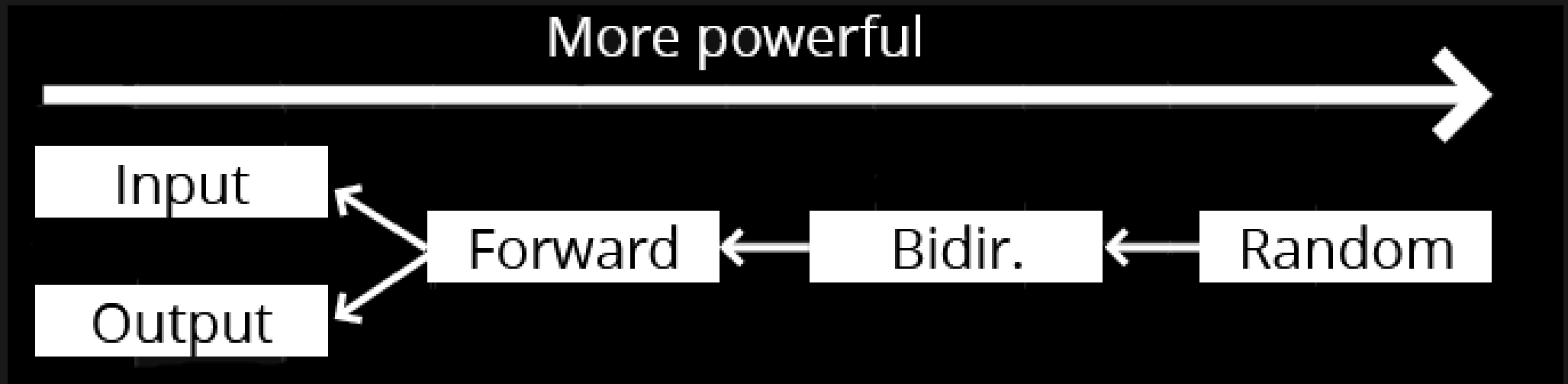
- This doesn't compile
- The lambda function can get access to the scope, but does not by default
- The scope is accessed via the capture []

```
1 #include <iostream>
2 #include <vector>
3
4 void add_n(std::vector<int>& v, int n) {
5     std::for_each(v.begin(), v.end(), [n] (int& val) { val = val + n; });
6 }
7
8 int main() {
9     std::vector<int> v{1,2,3};
10    add_n(v, 3);
11 }
```



Iterator Categories

Operation	Output	Input	Forward	Bidirectional	Random Access
Read		<code>*p</code>	<code>*p</code>	<code>*p</code>	<code>*p</code>
Access		<code>*p</code>	<code>*p</code>	<code>*p</code>	<code>*p []</code>
Write	<code>*p=</code>		<code>*p=</code>	<code>*p=</code>	<code>*p=</code>
Iteration	<code>++</code>	<code>++</code>	<code>++</code>	<code>++ --</code>	<code>++ -- + - += -=</code>
Compare		<code>== !=</code>	<code>== !=</code>	<code>=== !=</code>	<code>== != < > <= >=</code>





Iterator Categories

An **algorithm** requires certain kinds of iterators for their operations A **container's** iterator falls into a certain category

- **input:** find(), equal()
 - **output:** copy()
 - **forward:** replace(), binary_search()
 - **bi-directional:** reverse()
 - **random:** sort()
- **forward:** forward_list
 - **bi-directional:** map, list
 - **random:** vector, deque

stack, queue are container adapters, and do not have iterators

Feedback



Or go to the [form here](#).

