COMP6771
Advanced C++ Programming

Week 2.3
STL Algorithms

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?

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

Simple Example

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

Via an iterator? Or for-range?

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

demo207-simple-sum.cpp

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

demo208-simple-sum.cpp

Simple Example

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

Via use of an STL Algorithm

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

demo209-accum.cpp

```cpp
1 // What type of iterator is required here?
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 }
```

(This is the underlying mechanics)

More examples

We can also use algorithms to:

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

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

(handwritten annotations:)
std::next
v.s.
std::advance

1+2 =3

demo211-algos.cpp

6

More examples

We can also use algorithms to:

- Check if an element exists

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

$O(n)$

map:: find better than

$O(n)$, $O(\log^n)$

demo212-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

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

demo213-bound.cpp

Algorithms with output sequences

```cpp
1  #include <iostream>
2  #include <vector>
3
4  char to_upper(unsigned char value) {
5    return static_cast<char>(std::toupper(static_cast<unsigned char>(value)));
6  }
7
8  int main() {
9
10   std::string s = "hello world";
11   // Algorithms like transform, which have output iterators,
12   // use the other iterator as an output.
13   auto upper = std::string(s.size(), '\0');
14   std::transform(s.begin(), s.end(), upper.begin(), to_upper);
15 }
```
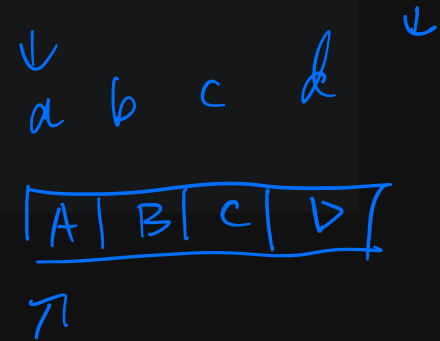
demo214-transform.cpp

Back Inserter

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

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

to_upper

some container
doesn't have
rbegin()
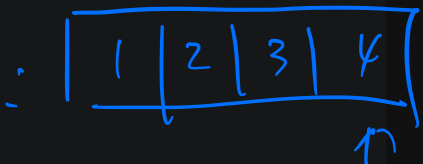
return
this
pointer.

| 1 | 2 | 3 | 4 |

demo215-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

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

capture

demo216-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 Captures

- 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 []

```cpp
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 }
```
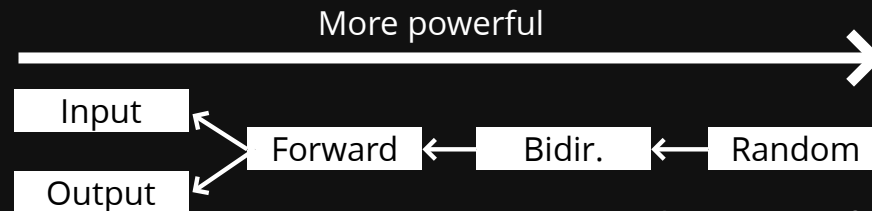
1+3, 2+3, 3+3

demo217-lambda2.cpp

Iterator Categories

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

More powerful

Input

Output

| Forward | | Bidir. | | Random |

"->" no longer specified as of C++20

Iterator Categories

An **algorithm** requires certain kinds
of iterators for their operations

- **input**: find(), equal()
- **output:** copy()
- **forward:** replace(), binary_search()
- **bi-directional**: reverse()
- **random:** sort()

A **container's** iterator falls into
a certain category

- **forward:** forward_list
- **bi-directional**: map, list
- **random:** vector, deque

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

Feedback