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?

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

int main() {
  std::vector<int> nums{1,2,3,4,5};

  int sum = 0;
  for (int i = 0; i <= nums.size(); ++i) {
    sum += i;
  }
  std::cout << sum << "\n";
};
```

Simple Example

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

Via an iterator? Or for-range?

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> nums{1,2,3,4,5};

  auto sum = 0;
  for (auto it = nums.begin(); it != nums.end(); ++it) {
    sum += *it;
  }
  std::cout << sum << "\n";
}
```

demo207-simple-sum.cpp

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> nums{1,2,3,4,5};

  int sum = 0;

  // Internally, this uses begin and end,
  // but it abstracts it away.
  for (const auto& i : nums) {
    sum += i;
  }

  std::cout << sum << "\n";
}
```

demo208-simple-sum.cpp

Simple Example

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

Via use of an STL Algorithm

```cpp
#include <iostream>
#include <numeric>
#include <vector>

int main() {
  std::vector<int> nums{1,2,3,4,5};
  int sum = std::accumulate(nums.begin(), nums.end(), 0);
  std::cout << sum << "\n";
}
```

demo209-accum.cpp

```cpp
// What type of iterator is required here?
template <typename T, typename Container>
T sum(iterator_t<Container> first, iterator_t<Container> last) {
  T total;
  for (; first != last; ++first) {
    total += *first;
  }
  return total
}
```

(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
#include <iostream>
#include <numeric>
#include <vector>

int main() {
  std::vector<int> v{1,2,3,4,5};
  int sum = std::accumulate(v.begin(), v.end(), 0);

  // What is the type of std::multiplies<int>()
  int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());

  auto midpoint = v.begin() + (v.size() / 2);
  // This looks a lot harder to read. Why might it be better?
  auto midpoint11 = std::next(v.begin(), std::distance(v.begin(), v.end()) / 2);

  int sum2 = std::accumulate(v.begin(), midpoint, 0);

  std::cout << sum << "\n";
}
```

demo211-algos.cpp

More examples

We can also use algorithms to:

- Check if an element exists

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> nums{1,2,3,4,5};

  auto it = std::find(nums.begin(), nums.end(), 4);

  if (it != nums.end()) {
        std::cout << "Found it!" << "\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
#include <algorithm>
#include <iostream>
#include <list>
#include <vector>

int main() {
  // Lower bound does a binary search, and returns the first value >= the argument.
  std::vector<int> sortedVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::lower_bound(sortedVec.begin(), sortedVec.end(), 5);

  std::list<int> sortedLinkedList{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::lower_bound(sortedLinkedList.begin(), sortedLinkedList.end(), 5);
}
```

demo213-bound.cpp

Algorithms with output sequences

```cpp
#include <iostream>
#include <vector>

char to_upper(unsigned char value) {
  return static_cast<char>(std::toupper(static_cast<unsigned char>(value)));
}

int main() {

  std::string s = "hello world";
  // Algorithms like transform, which have output iterators,
  // use the other iterator as an output.
  auto upper = std::string(s.size(), '\0');
  std::transform(s.begin(), s.end(), upper.begin(), to_upper);
}
```

demo214-transform.cpp

Back Inserter

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

```cpp
#include <iostream>
#include <vector>

char to_upper(char value) {
  return static_cast<char>(std::toupper(static_cast<unsigned char>(value)));
}

int main() {

  std::string s = "hello world";
  // std::for_each modifies each element
  std::for_each(s.begin(), s.end(), toupper);

  std::string upper;
  // std::transform adds to third iterator.
  std::transform(s.begin(), s.end(), std::back_inserter(upper), to_upper);
}
```

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
#include <iostream>
#include <vector>

int main() {
  std::string s = "hello world";
  // std::for_each modifies each element
  std::for_each(s.begin(), s.end(), [] (char& value) { value = std::toupper(value); });
}
```

demo216-lambda1.cpp

Lambda Functions

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

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

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

Lambda Captures

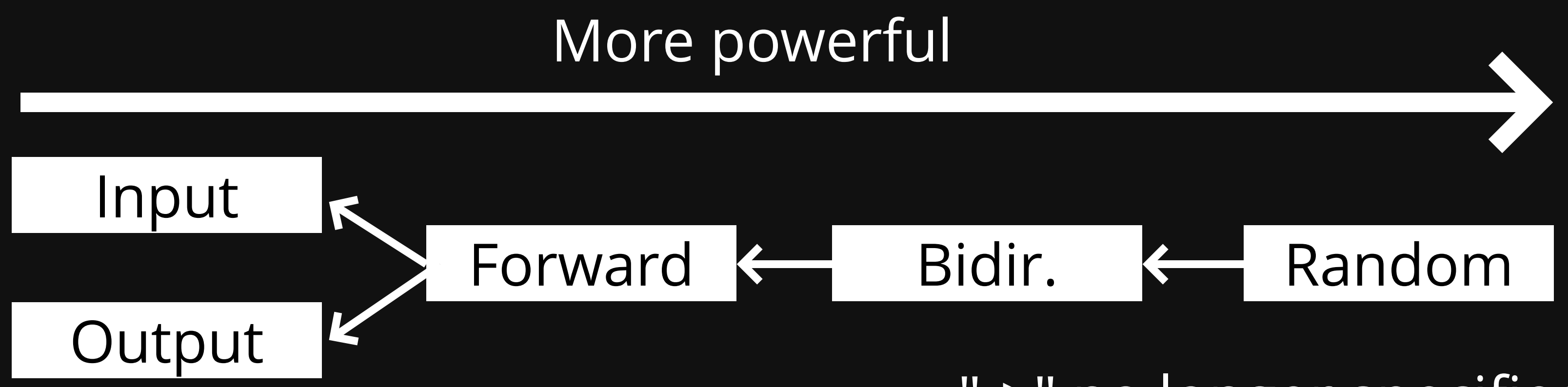
- This doesn't compile
- The lambda function can get access to the scope, but does not by default
- The scope is accessed via the capture []

```cpp
#include <iostream>
#include <vector>

void add_n(std::vector<int>& v, int n) {
  std::for_each(v.begin(), v.end(), [n] (int& val) { val = val + n; });
}

int main() {
  std::vector<int> v{1,2,3};
  add_n(v, 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		== !=	== !=	== !=	== != < > <= >=



"->" 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