COMP6771
Advanced C++ Programming

Week 2.3
STL Algorithms

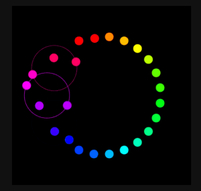
Ocean of Algorithms

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.
- Come with C++ Compiler
- template function not class
- useful but generic
- useful name
- other libraries than STL such as ASL, Boost algorithm library, and several others

Why STL: Algorithms?

- STL **Algorithms** makes code expressive
  - raising abstraction level
  - Spectacular
- avoid common mistakes
  - empty loop
  - off by one
  - complexity ?



- Declarative syntax: avoid loop
- Iterate over small sequence
- Common use (standard, basic), can build on top of it
- Whatever compiler?
- work on a number of **containers** as long as those containers can be represented via a relevant iterator.
- Designed by Experts

```
std::sort(a.begin(), a.end(), [](const employee& x, const employee& y) {
                                  return x.last < y.last; });
auto p = std::lower_bound(a.begin(), a.end(), "Parent",
                                 [](const employee& x, const string& y) {
                                   return x.last < y; });
```

↓

```
ranges::sort(a, ranges::less<>(), &employee::last);
auto p = ranges::lower_bound(a, "Parent", ranges::less<>(), &employee::last);
```

```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
bool flag = true;
for (int i = 1; (i <= v.size()) && flag; i++) {
   flag = false;
   for (int j = 0; j < (v.size() -1); j++) {
      if (v[j+1] < v[j]) {
         std::swap(v[j], v[j+1]);
         flag = true;
      }
   }
}
for (int i:v) std::cout << i << " ";
```

```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
std::sort(v.begin(), v.end());
for (int i:v) std::cout << i << " ";
```

Why Algorithms?

- Often more efficient than handwritten loops
- tested and debugged
- Cleaner and more clearly abstracted than raw loop
  - min_element(vec.begin(), vec.end());
- Contains side effect inside a clear interface
- Prevents accidental leakage
- Eases reasoning about functionality and reasoning about post condition
- Ease reasoning about surrounding code
- Less likely to fail
- Easier
  - easier to write code,
  - easier to debug code

Simple Example

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

C-style?

Many lines of code?

May have side effects ?

```
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
#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

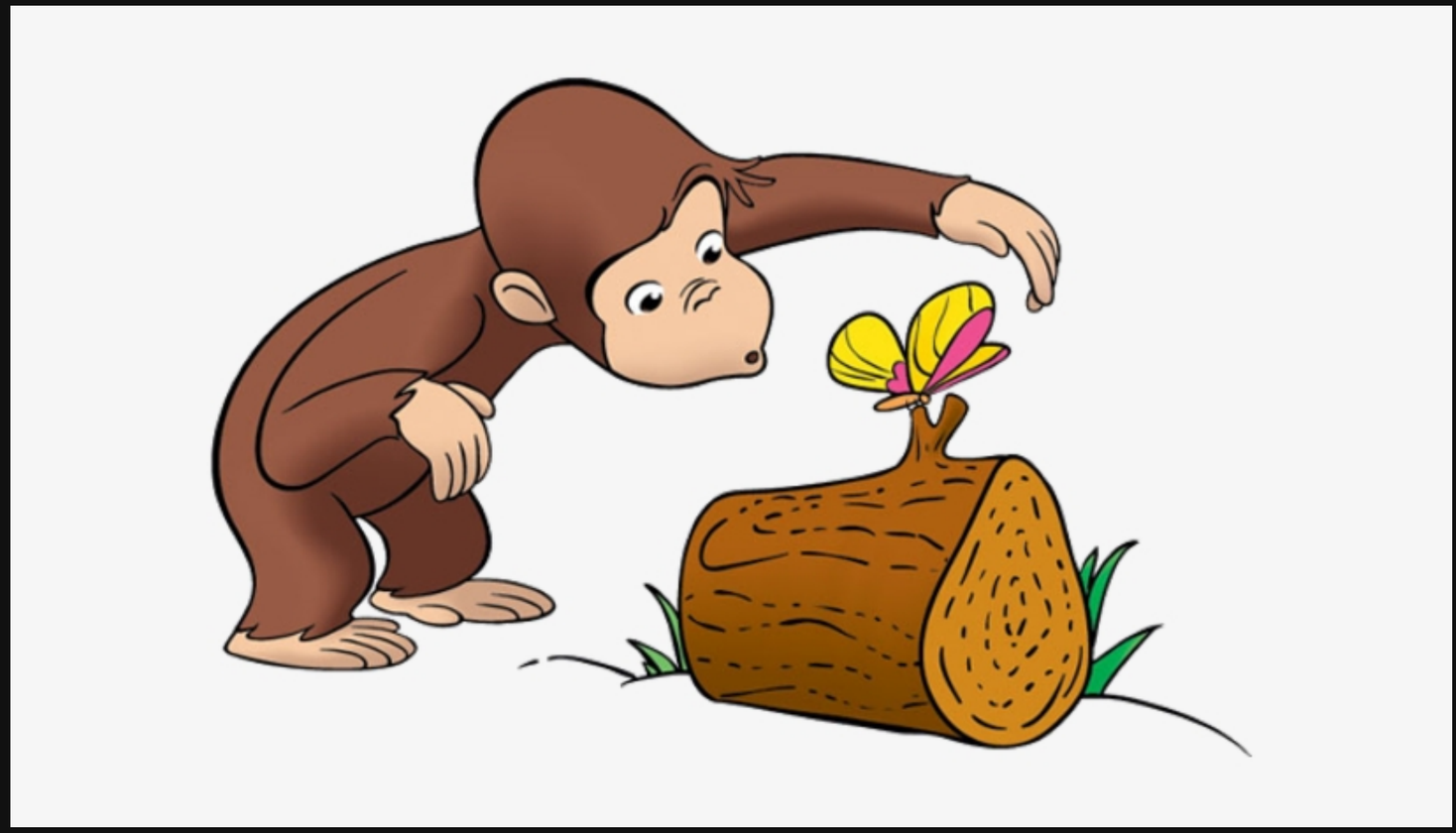
We can also use algorithms to:

- 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

for each



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

Classes of STL Algorithms

1. Non-Modifying sequence operation
2. Sorting And related operations
3. Mutating sequence operations
4. General numeric operations
5. General C Algorithms

```
template<class InputIt, class OutputIt>
OutputIt copy(InputIt first, InputIt last,
              OutputIt d_first)
{
    for (; first != last; (void)++first, (void)++d_first) {
        *d_first = *first;
    }
    return d_first;
}
```

std::min

Defined in header <algorithm>

```
template< class T >
const T& min( const T& a, const T& b );                          (1)  (until C++14)
template< class T >
constexpr const T& min( const T& a, const T& b );                     (since C++14)

template< class T, class Compare >
const T& min( const T& a, const T& b, Compare comp );            (2)  (until C++14)
template< class T, class Compare >
constexpr const T& min( const T& a, const T& b, Compare comp );        (since C++14)

template< class T >
T min( std::initializer_list<T> ilist );                         (3)  (since C++11) (until C++14)
template< class T >
constexpr T min( std::initializer_list<T> ilist );                     (since C++14)

template< class T, class Compare >
T min( std::initializer_list<T> ilist, Compare comp );           (4)  (since C++11) (until C++14)
template< class T, class Compare >
constexpr T min( std::initializer_list<T> ilist, Compare comp );       (since C++14)
```

Returns the smaller of the given values.

1-2) Returns the smaller of a and b.

3-4) Returns the smallest of the values in initializer list ilist.

Non-Modifying Sequence STL

- Dont modify the input sequence
- Dont emit a result sequence
- no impact on input sequence
- Function object, if present may impact through modification of itself,
  - e.g. for_each, all_of, any_of find, find_end, find_first_of, search, equal, count

Mutating Sequence Operation

- Dont modify input sequence except where the output overlap input resulting in modification
- Emits an output sequence of results
- Output sequence may overlap with input for certain algorithms (transform)
- Algorithms will explicitly cause side effect in output sequence
- function object, if present may impact by modifying itself or its environment. it should not modify the input or output
  - Copy (copy_n, copy_if, copy_backward), move, swap_range, transform, fill, rotate, unique, remove, reserve, partition, generate

Sorting and Related Operations

- Mix of non-modifying and mutating
- mutating operation modify sequence in place (sort) or emits output to output sequence (merge)
- Default compare function is operator <
- Explicit compare function objects, if supplied must not modify
  - e.g. Sorting (sort, stable_sort, partial sort, lower_bound,
  - heap operations (push heap, pop heap etc)
  - minimum and max
  - merge
  - operation of sorted containers

General Numeric

- Library of algorithms for number operations
- consist of components for complex number type, random number generation
- e.g. accumlate, inner_product, partial_sum, iota, adjacent_difference
-

C Library Algorithms

All discussed earlier can what thiese can do :)

bsearch, qsort

for_each and transform

generic algorithms

apply operation to each element in order

very similar complexity?

for_each

- apply operation on each element in sequence
- non-modifying sequence operation: no output: relies on function for mutation
- no-side effect by for each, however, function may
- returns a moved copy of function object

```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";
}
```

transform

- apply operation on each element in sequence
- non-modifying sequence operation: no output: relies on function for mutation
- if the input range(s) and result range are the same, or overlap mutates object in-place
- algorithms side effect, however, function may not
- explicity generate output range, hence
- returns iterator pointing one past last element in result range

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

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

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

Lambda Captures

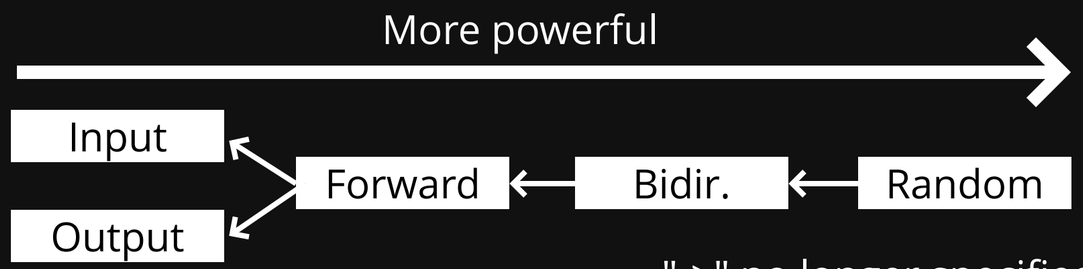
- This doesn't compile
- The lambda function can get access to the scope, but does not by default
- The scope is accessed via the capture []

```cpp
#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

Writing your Own Algo.

```
template <typename Iter, typename Func>
void adjacent_pair(Iter first, Iter last, Func f);
```

```
template <typename FwIter, typename Func>
void adjacent_pair(FwIter first, FwIter last, Func f)
{
   if (first != last)
   {
      FwIter trailer = first;
      ++first;
      for (; first != last; ++first, ++trailer)
         f(*trailer, *first);
   }
}
```

Writing your own

```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
bool flag = true;
for (int i = 1; (i <= v.size()) && flag; i++) {
   flag = false;
   for (int j = 0; j < (v.size() -1); j++) {
      if (v[j+1] < v[j]) {
         std::swap(v[j], v[j+1]);
         flag = true;
      }
   }
}
for (int i:v) std::cout << i << " ";
```

```
std::vector<int> v{0,1,3,5,7,9,2,4,6,8};
std::sort(v.begin(), v.end());
for (int i:v) std::cout << i << " ";
```

Writing your own

- Write what you need
- More general
- stepwise refinement, testing, debugging

Tips

1. **Complexity**
2. Degenerative cases i.e. empty cases
3. Think about iterator category

Feedback