COMP6771
Advanced C++ Programming

2.4 Standard Algorithms

Algorithms

- The Standard Library provides a plethora of algorithms that operate on iterators.
- In this way, they can work on a large number of containers as long as those containers can be represented via an appropriate iterator.
- These algorithms can be found amongst a few header files:
  - Majority are in `<algorithm>`
  - Some are in `<numeric>` (notably, `std::accumulate`)
  - The full list of algorithms can be found here

Simple Example

- What's the best way to sum a list of numbers?
  - Is it with a C-style for-loop?
  - Is it with an iterator-based for-loop?
  - Is it with a ranged for-loop?
- The best way is to use a Standard Algorithm: `std::accumulate`!

```
#include <numeric>
auto v = std::vector<int>{42, 6771, 96};
int total =
    std::accumulate(v.begin(), v.end(), 0);

// contrast to...

int n = 0;
for (auto i = 0u; i < v.size(); ++i) {
     n += v[i];
}
```

std::accumulate Implementation

```
template <typename InputIt, typename T>
T accumulate(InputIt first, InputIt last, T n) {
    for (; first != last; ++first) {
        n += *first;
    }
    return first;
} // the underlying method of accumulate should be familiar
```

Almost all of the algorithms are implemented as **function templates** for maximum code reuse and efficiency.

At this point you do not need to know how to write a template.

Common Algorithms

- Some of the most commonly used algorithms are:
  - `std::copy` – a type-safe and more powerful replacement of `memcpy()`.
  - `std::find` – a linear search algorithm to find an element in a container.
  - `std::transform` – C++'s version of `map()` from other languages.
  - `std::swap` – a classic three-step swap implementation.
  - `std::accumulate` – performs a left fold. Can be used for sums, products, and other arbitrary operations.

Example: std::copy

```
#include <iostream>
#include <iterator>
#include <string>
int main() {
        std::copy(
                std::istream_iterator<std::string>{std::cin},
                std::istream_iterator<std::string>{},
                std::ostream_iterator<std::string>{std::cout}
        ); // this echoes each line of stdin to stdout a.k.a the cat command
}
```

Example: `std::find`

```
#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";
  }
}
```

Using Algorithms

- Some algorithms accept a predicate function that performs a task appropriate to the algorithm.
  - E.g., `std::find_if` accepts a function that says whether or not an element is “found”.
  - `std::accumulate` accepts a function that defines what operation to use instead of the default summation.
- Having to define a new function for a one-off operation makes using algorithms burdensome.
- C++11 introduced Lambda Functions to solve this problem.

Lambda Functions

- A function that can be defined inside of other functions.
- Can be passed to and returned from functions and stored in variables.
- Can capture all, none, or some of the variables in the enclosing scope.
- Convenient and replaces one-off functions

```
#include <iostream>
#include <string>

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

Anatomy of a Lambda

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

[]{ /* lambdas with no parameters can omit () */};

// lambdas can be stored into variables. Type must be auto
auto rand = []() -> int { return 6771; }

// lambdas with no captures decay into regular function pointers
int(*cmp)(int, int) = [](int a, int b) { return a < b; };
```

Lambda Captures

- By default, lambdas execute in their own scope.
- Gain access to outside scope by capturing
  - Capture by value
  - Capture by reference
- Considerations:
  - Capturing by value makes a copy. This may be expensive for large types.
  - Capturing by reference could lead to dangling references if returning a lambda from a function

```
int a = 0, b = 2;

// will not compile: did not capture a
[]() { std::cout << a << std::endl; }

// OK: captured a by value
[a]() { std::cout << a << std::endl; }

// OK: captured a by reference
[&a]() { std::cout << a << std::endl; }

// OK: captured everything by reference
[&]() { std:: cout << a + b << “\n”; }

// OK: captured everything by value
[=]() { std:: cout << f + b << “\n”; }
```

Algorithms Performance & Portaibility

- 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 with 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

```
#include <algorithm>
#include <list>
#include <vector>

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

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

Algorithms’ Iterator Requirements

An **algorithm** requires certain kinds of iterators for its operation

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

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

forward: forward_list

bidirectional: map, list

random: vector, deque

The container adaptors (stack, queue, etc.) do not have iterators

Feedback (stop recording)