STL Containers and Lambda Functions

CS 247

University of Waterloo

cs247@uwaterloo.ca

July 25th, 2018
Overview

1 STL Containers
   - Random Access Containers
   - Lists
   - Associative Collections

2 Algorithms
   - Non-modifying vs. Modifying Algorithms
   - Algorithms over Multiple Ranges
   - Algorithms Using Functions
   - Algorithms Using Function Objects

3 Lambda Functions
   - Motivation for Lambda
   - Lambda Functions
   - Capture Lists
   - Naming a Lambda
The standard template library has concrete implementations of many different containers. A good reference is available at http://www.cplusplus.com/reference/stl/.
You’ll want to choose a container based on the situation you’re in.

- **Array**: if you know in advance how many elements you’ll have.
- **Vector**: if you’ll be adding items to the end.
- **Deque (double-ended queue)**: if you’ll be adding items to both ends.

All of the above have the benefit of random-access, which is handy.
Lists don’t have the ability to do random access, but they have the added benefit of being able to insert into the *middle* without much work. A forward list (new to C++11) is a list without the double-ended pointers, so you can only iterate through them in one direction (on the plus side, they take up a little less space).
An Associative Collection is a container that doesn’t keep track of the order in which elements were inserted (e.g. a Set, or a Map). These are typically implemented as Binary Search Trees, and can be iterated over in order (i.e. alphabetically by key).

Unordered Associative Collections are similar, but are implemented as hash functions instead of BSTs. This makes lookup faster (O(1) instead of O(log n)) but does not keep the keys in meaningful order.
Algorithms over a container will use the iterators associated with the same container. An algorithm that requires X type of iterator will still work if given an iterator that is more powerful; e.g. if a bi-directional iterator is needed, a random-access iterator (which is more powerful) will work as well.
Non-modifying Algorithms

As the name implies, these algorithms do not modify the containers they iterate over.

e.g. count:

```cpp
int count(InputIterator first,
          InputIterator last,
          const T& value)
```

will count the number of elements in the range equal to value.
Modifying Algorithms will insert (or overwrite) values in the destination container.

```
OutputIterator copy (InputIterator first, 
               InputIterator last, 
               OutputIterator result);
```
Sometimes, you may want to iterate over multiple containers in lock-step, e.g. to see if two arrays are equal, add their values together, etc.

```cpp
template <class InputIterator1, class InputIterator2>
bool equal ( InputIterator1 first1,
            InputIterator1 last1,
            InputIterator2 first2 );
```
Trouble

You can get in trouble if you aren’t careful with your iterators. The “range” of the second container is assumed to be equal to the range of the first (which is why e.g. in `equal` above, there is no iterator `last2`). If the second container is larger, this is mostly harmless. However, if the first container is larger, you will end up reading memory you shouldn’t be reading from the second container, which leads to unpredictable/undefined behaviour.

The same is true for the destination range (for functions like `transform`) - if it is not large enough, you may end up overwriting memory you shouldn’t be changing.
replace_copy_if copies the elements in the range \([\text{first}, \text{last})\) to the range beginning at result, replacing those for which \(\text{pred}\) returns true.

```cpp
std::vector<int> v{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
std::replace_copy_if(v.begin(), v.end(),
    std::ostream_iterator<int>(std::cout, " "),
    [](int n){return n > 5;}, 99);
std::cout << std::endl;
```
for_each will take a function and apply that function to a range.

```cpp
void Double(int &x){x = 2*x;}

vector<int> nums{1,2,3,4,5};
for_each(nums.begin(), nums.end(), Double);
// doubles all values in nums
```
transform will take a function, apply it to a range, and then store the result into a destination.

```cpp
int Double(int x){return 2 * x;}
vector<int> nums{1,2,3,4,5};
vector<int> newNums{0,0,0,0,0};
transform(nums.begin(),
    nums.end(),
    newNums.begin(),
    Double);
// doubles all values in nums, stores result in newNums
Function Objects

A function object is an object with an overloaded operator(), which allows the algorithms library to use it as a function.

class divisibleBy{
    int modulo_;  
public:
    divisibleBy(int d): modulo_(d) {}
    bool operator() (int n) {return (n%modulo_==0;)}
};

divisibleBy DBobject(3);
cout << DBobject(7) << endl;
Motivation for Lambda

When we have to declare a function (or a class) to be used with algorithms - we have to write a function definition for a function we might only use once. This isn’t the most elegant thing in the world: equivalent to

```cpp
string t = "tutorials are fun!";
cout << t << endl;
```

as opposed to

```cpp
cout << "tutorials are fun!" << endl;
```

Lambda Functions follow the same logic. You can declare an in-line, anonymous function exactly where you need to use it, and then forget about it (i.e. you don’t pollute the namespace.)
Lambda Functions

Lambda functions look like this:

```
[ capture list ] ( param list ) -> return type
  { function body };
```

The capture list is the set of local variables to “copy over” into the lambda function.
The return type of a function can be inferred (compilers are smart).
By default, a “captured” variable is copied, and so doesn’t change within the lambda function. They can, however, be captured by reference with “&”, in the same way parameters can be passed-by-reference.
If you’re declaring a lambda as part of a class’ member function, the lambda by default can’t access that class’ members, *even if you put those members in the capture list.* To get around this, you can put *this* in the capture list.
Naming a Lambda

In some cases you may want to give a lambda function a name (this is useful so that you can, for example, define a function inside of another function).

```cpp
auto fn_name = [ capture list ]
    ( params )
    -> return type
    { function body }
```
char* v[] = {"Hello!", "World!
"};
int repeats = 3;

// Use lambda function to print "Hello"
// in "repeats" times followed by "World" in "repeats" times
Another Lambda Function Example

```c
int nums[] = {11, 13, 17, 29, 173, 2015};
int modulo = 3;
// Use lambda function to divide each element by "modulo"
// store the result in nums
```
vector <accountant> emps {{Josh, 2100.0}, {Kate, 2900.0}, {Rose, 1700.0}};
const auto lower_limit = 1600.0;
const auto upper_limit = 1.5*min_wage;

//report the first accountant who has a salary that is within specific range
End