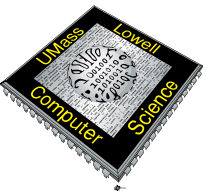
Lambda Expressions Part 1

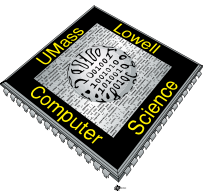
COMP.2040 – Computing IV

Dr. Tom Wilkes



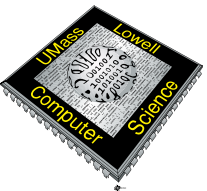
Sources

- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, *C++ Primer* (5th edition)



Generic Algorithms

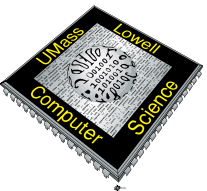
- *The sequential containers* define few operations: For the most part, we can add and remove elements, access the first or last element, determine whether a container is empty, and obtain iterators to the first or one past the last element.
- We can imagine many other useful operations one might want to do: We might want to find a particular element, replace or remove a particular value, reorder the container elements, and so on.
- Rather than define each of these operations as members of each container type, the standard library defines a set of **generic algorithms**: "algorithms" because they implement common classical algorithms such as sorting and searching, and "generic" because they operate on elements of differing type and across multiple container types—not only library types such as vector or list, but also the built-in array type—and, as we shall see, over other kinds of sequences as well.



Overview of Algorithms

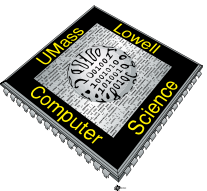
- In general, the algorithms do not work directly on a container. Instead, they operate by traversing a range of elements bounded by two iterators.
- Typically, as the algorithm traverses the range, it does something with each element.
- For example, suppose we have a vector of `ints` and we want to know if that vector holds a particular value. The easiest way to answer this question is to call the library `find` algorithm:

```
int val = 42; // value we'll look for

// result will denote the element we want if it's in vec, or
// vec.cend() if not
auto result = find(vec.cbegin(), vec.cend(), val);

// report the result
cout << "The value " << val
     << (result == vec.cend()
           ? " is not present" : " is present") << endl;
```



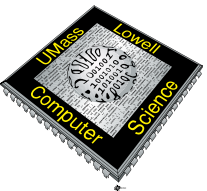
How the Algorithms Work

- To see how the algorithms can be used on varying types of containers, let's look a bit more closely at `find`. Its job is to find a particular element in an unsorted sequence of elements. Conceptually, we can list the steps `find` must take:
  1. It accesses the first element in the sequence.
  2. It compares that element to the value we want.
  3. If this element matches the one we want, `find` returns a value that identifies this element.
  4. Otherwise, `find` advances to the next element and repeats steps 2 and 3.
  5. `find` must stop when it has reached the end of the sequence.
  6. If `find` gets to the end of the sequence, it needs to return a value indicating that the element was not found. This value and the one returned from step 3 must have compatible types.
- None of these operations depends on the type of the container that holds the elements. So long as there is an iterator that can be used to access the elements, `find` doesn't depend in any way on the container type (or even whether the elements are stored in a container).



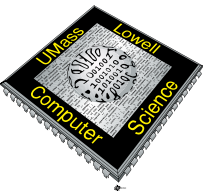
Iterators

- **Iterators Make the Algorithms Container Independent, ...**
  - All but the second step in the find function can be handled by iterator operations: The iterator dereference operator gives access to an element's value; if a matching element is found, find can return an iterator to that element; the iterator increment operator moves to the next element; the "off-the-end" iterator will indicate when find has reached the end of its given sequence; and find can return the off-the-end iterator to indicate that the given value wasn't found.
- **...But Algorithms Do Depend on Element-Type Operations**
  - Although iterators make the algorithms container independent, most of the algorithms use one (or more) operation(s) on the element type. For example, step 2, uses the element type's == operator to compare each element to the given value.
  - Other algorithms require that the element type have the < operator. However, as we'll see, most algorithms provide a way for us to supply our own operation to use in place of the default operator.



Key Concept: Algorithms Never Execute Container Operations

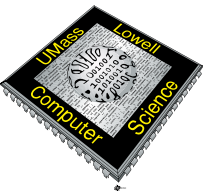
- The generic algorithms do not themselves execute container operations. They operate solely in terms of iterators and iterator operations. The fact that the algorithms operate in terms of iterators and not container operations has a perhaps surprising but essential implication: Algorithms never change the size of the underlying container. Algorithms may change the values of the elements stored in the container, and they may move elements around within the container. They do not, however, ever add or remove elements directly.
- There is a special class of iterator, the inserters, that do more than traverse the sequence to which they are bound. When we assign to these iterators, they execute insert operations on the underlying container. When an algorithm operates on one of these iterators, the *iterator* may have the effect of adding elements to the container. The *algorithm* itself, however, never does so.



Types of Algorithms

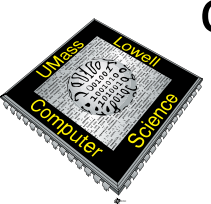
The standard library contains well over 100 algorithms. The algorithms fit roughly into the following types:

- Read-only algorithms
- Algorithms that write container elements
- Algorithms that reorder container elements



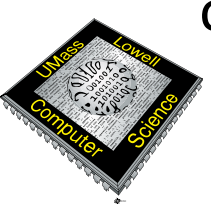
Customizing Operations

- Many of the algorithms compare elements in the input sequence. By default, such algorithms use either the element type's < or == operator. The library also defines versions of these algorithms that let us supply our own operation to use in place of the default operator.
- For example, the `sort` algorithm uses the element type's < operator. However, we might want to sort a sequence into a different order from that defined by <, or our sequence might have elements of a type (such as `Sales_data`) that does not have a < operator. In both cases, we need to override the default behavior of `sort`.



Passing a Function to an Algorithm

- As one example, assume that we want to print the vector after we call `elimDups`. However, we'll also assume that we want to see the words ordered by their size, and then alphabetically within each size. To reorder the vector by length, we'll use a second, overloaded version of sort. This version of sort takes a third argument that is a **predicate**.
- A predicate is an expression that can be called and that returns a value that can be used as a condition. The predicates used by library algorithms are either **unary predicates** (meaning they have a single parameter) or **binary predicates** (meaning they have two parameters). The algorithms that take predicates call the given predicate on the elements in the input range. As a result, it must be possible to convert the element type to the parameter type of the predicate.

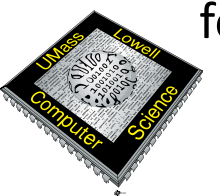


Predicates

- The version of `sort` that takes a binary predicate uses the given predicate in place of < to compare elements. The predicates that we supply to sort must meet the requirements that we'll describe in a later chapter. For now, what we need to know is that the operation must define a consistent order for all possible elements in the input sequence.
- Our `isShorter` function is an example of a function that meets these requirements, so we can pass `isShorter` to sort. Doing so will reorder the elements by size:

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}


// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

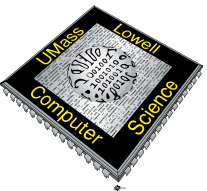
- Given an example data set, this call would reorder words so that all the words of length 3 appear before words of length 4, which in turn are followed by words of length 5, and so on.



Sorting Algorithms

- When we sort words by size, we also want to maintain alphabetic order among the elements that have the same length. To keep the words of the same length in alphabetical order we can use the `stable_sort` algorithm. A stable sort maintains the original order among equal elements.
- Ordinarily, we don't care about the relative order of equal elements in a sorted sequence. After all, they're equal. However, in this case, we have defined "equal" to mean "have the same length." Elements that have the same length still differ from one another when we view their contents. By calling `stable_sort`, we can maintain alphabetical order among those elements that have the same length:

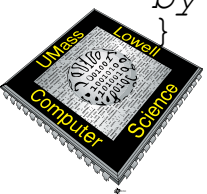
```
elimDups(words);// put words in alphabetical order & remove duplicates


// resort by length, maintaining alphabetical order among words
// of the same length
stable_sort(words.begin(), words.end(), isShorter);


for (const auto &s : words)  // no need to copy the strings
    cout << s << " ";  // print each element separated by a space


cout << endl;
```



Limitations of Predicates

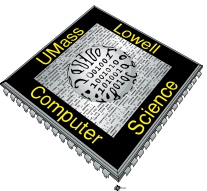
- The predicates we pass to an algorithm must have exactly one or two parameters, depending on whether the algorithm takes a unary or binary predicate, respectively. However, sometimes we want to do processing that requires more arguments than the algorithm's predicate allows. For example, a solution would have to hard-wire the size into the predicate used to partition the sequence. It would be move useful to be able to partition a sequence without having to write a separate predicate for every possible size.
- As a related example, we'll revise our program to report how many words are of a given size or greater. We'll also change the output so that it prints only the words of the given length or greater.
- A sketch of this function, which we'll name biggies, is as follows:

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // put words in alphabetical order and remove
duplicates
    // resort by length, maintaining alphabetical order among words
of the same length
    stable_sort(words.begin(), words.end(), isShorter);
    // get an iterator to the first element whose size() is >= sz
    // compute the number of elements with size >= sz
    // print words of the given size or longer, each one followed
by a space
}
```



Limitations of Predicates (cont.)

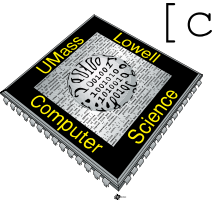
- Our new problem is to find the first element in the vector that has the given size. Once we know that element, we can use its position to compute how many elements have that size or greater.
- We can use the library `find_if` algorithm to find an element that has a particular size.
  - Like `find`, the `find_if` algorithm takes a pair of iterators denoting a range.
  - Unlike `find`, the third argument to `find_if` is a predicate.
  - The `find_if` algorithm calls the given predicate on each element in the input range. It returns the first element for which the predicate returns a nonzero value, or its end iterator if no such element is found.
- It would be easy to write a function that takes a string and a size and returns a bool indicating whether the size of a given string is greater than the given size.
  - However, `find_if` takes a unary predicate—any function we pass to `find_if` must have exactly one parameter that can be called with an element from the input sequence.
  - There is no way to pass a second argument representing the size.
  - To solve this part of our problem we'll need to use some additional language facilities.



Introducing Lambdas

- We can pass any kind of callable object to an algorithm. An object or expression is callable if we can apply the call operator to it.
- That is, if `e` is a callable expression, we can write `e(args)` where `args` is a comma-separated list of zero or more arguments.
- The only callables we've used so far are functions and function pointers.
- There are two other kinds of callables: classes that overload the function-call operator, and lambda expressions.
- A lambda expression represents a callable unit of code. It can be thought of as an unnamed, inline function. Like any function, a lambda has a return type, a parameter list, and a function body. Unlike a function, lambdas may be defined inside a function.
- A lamba expression has the form:

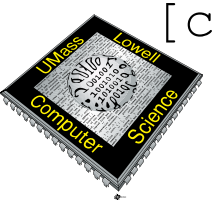
```
[capture list] (parameter list) -> return type
    { function body }
```



Introducing Lambdas

- We can pass any kind of callable object to an algorithm. An object or expression is callable if we can apply the call operator to it.
- That is, if `e` is a callable expression, we can write `e(args)` where `args` is a comma-separated list of zero or more arguments.
- The only callables we've used so far are functions and function pointers.
- There are two other kinds of callables: classes that overload the function-call operator, and lambda expressions.
- A lambda expression represents a callable unit of code. It can be thought of as an unnamed, inline function. Like any function, a lambda has a return type, a parameter list, and a function body. Unlike a function, lambdas may be defined inside a function.
- A lamba expression has the form:

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



A Simple Lambda Expression

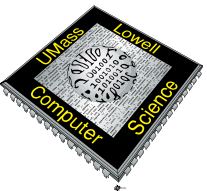
The following simple lambda has no capture list, parameter list, or explicit return type, and the return type is deduced as `int`:

```
auto f = [] { return 42; };
```

Here's how it can be called:

```
cout << f() << endl;  // prints 42
```

Note: Lambdas with function bodies that contain anything other than a single return statement that do not specify a return type return `void`.



`is_Shorter` function as a Lambda

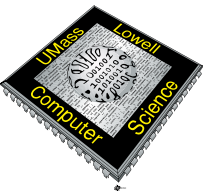
We can rewrite our `is_Shorter` function from the Part 2 slides as a lambda that takes two parameters:

```
[](const string &a, const string &b)
        { return a.size() < b.size();}
```

Because the capture list is empty, this lambda does not use any local variables from the surrounding environment (e.g., function body).

Now the call to `stable_sort` looks like this:

```
// sort words by size, but maintain alphabetical
// order for words of the same size
stable_sort(words.begin(), words.end(),
        [](const string &a, const string &b)
                { return a.size() < b.size();});
```



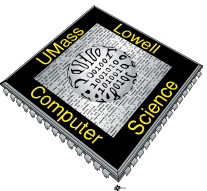
Lambda with Capture List

Recall our `biggies` function from the Part 2 slides:

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // put words in alphabetical order
and remove duplicates
    // resort by length, maintaining alphabetical order
among words of the same length
    stable_sort(words.begin(), words.end(), isShorter);
    // get an iterator to the first element whose size()
is >= sz
    // compute the number of elements with size >= sz
    // print words of the given size or longer, each one
followed by a space
}
```

We're now ready to write a lambda for the `find_if` iterator:

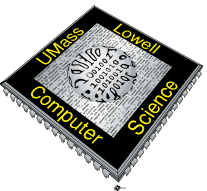
```
// get an iterator to the first element whose size() is >= sz
auto wc = find_if(words.begin(), words.end(),
                  [sz](const string &a)
                      { return a.size() >= sz; });
```



The `for_each` Algorithm

- The last part of our problem is to print the elements in words that have length `sz` or greater.
- To do so, we'll use the `for_each` algorithm.
  - This algorithm takes a callable object and calls that object on each element in the input range.

```
// print words of the given size or longer, each
// one followed by a space
for_each(wc, words.end(),
         [](const string &s){cout << s << " ";});
cout << endl;
```



Putting It All Together

```
void biggies(vector<string> &words, vector<string>::size_type sz)
{
   // put words in alphabetical order and remove duplicates
   elimDups(words);

   // sort words by size, but maintain alphabetical order for words
   // of the same size
   stable_sort(words.begin(), words.end(),
      [](const string &a, const string &b)
         { return a.size() < b.size();});

   // get an iterator to the first element whose size() is >= sz
   auto wc = find_if(words.begin(), words.end(),
      [sz](const string &a)
         { return a.size() >= sz; });

   // compute the number of elements with size >= sz
   auto count = words.end() - wc;
   cout << count << " " << make_plural(count, "word", "s")
        << " of length " << sz << " or longer" << endl;

   // print words of the given size or longer, each one followed by a space
   for_each(wc, words.end(),
      [](const string &s){cout << s << " ";});
   cout << endl;
}
```