STL Algorithms, Part I

Introduction

Algorithms are an amazing collection of template functions that work on ranges defined by iterators. They encompass a wide scope of functionality, enabling you to leverage off of preexisting code. While algorithms do not introduce any new functionality to your code, they nonetheless provide incredible speed bonuses during design and at runtime. However, algorithms are a complicated and require some adjusting to get used to. This handout discusses the basic syntax for algorithms, some of their programming conventions, and the more common algorithms you'll encounter in practice.

A Word on Readability

Used correctly, STL algorithms can clarify your code by making explicit what operation you're performing on a set of data. However, algorithm syntax can rapidly become a stylistic nightmare. When we revisit algorithms after discussing operator overloading, this warning will be even more relevant.

In general, when working with algorithms, try to keep the code as readable as possible. You'll be writing single lines of code that do the work of three or four, so make sure that you comment your code and add whitespace when appropriate. Strongly consider adding line breaks in the middle of your function calls to separate out different blocks of code. If you're not used to commenting your code extensively, I strongly urge you to begin doing so when working with algorithms. Code using STL algorithms can be incredibly cryptic and code that you've written only ten or twenty minutes earlier can be unrecognizable.

If you're able to keep up the good style, you'll reap some incredible benefits from STL algorithms. The speed boosts over your hand-written loops or searching routines are incredible, and you'll be in for a real treat when you see your code in action.

A Word on Compiler Errors

More so than with any other portion of the STL, when working with algorithms the compiler errors you'll get can be completely incomprehensible. When you pass invalid parameters to an algorithm, the compiler may report errors in the code for the STL algorithms rather than in the code you wrote. If this happens, search the errors carefully and look for messages like “during template instantiation of” or “while instantiating.” These phrases identify the spot where the compiler choked on your code. Read over your code carefully and make sure that your parameters are valid. Did you provide an output iterator instead of an input iterator? Is your comparison function accepting the right parameter types? Deciphering these types of errors is a skill you can only learn with time.

A Word on Header Files

STL algorithms are in either the `<algorithm>` or `<numeric>` header files. All algorithms have the same basic syntax and there isn't much of a functional distinction between the two libraries. However, algorithms from `<numeric>` tend to be based more on computational programming, while

algorithms from <algorithm> tend to be more general-purpose. It's common to see both headers at the top of professional code, so don't be confused by the <numeric>. If you receive compiler errors about undefined functions, make sure you've included both these headers.

Basic STL Algorithms

The best way to understand STL algorithms is to see them in action. For example, the following code snippet prints the sum of all of the elements in a set<int>:

```
cout << accumulate(mySet.begin(), mySet.end(), 0) << endl;
```

In that tiny line of code, you're effectively iterating over the entire set and summing the values together. This exemplifies STL algorithms – performing huge tasks in a tiny space.

The accumulate function, defined in the <numeric> header, takes three parameters – two iterators that define a range of elements and an initial value to use in the summation – and returns the sum of the elements in that range plus the base value. Notice that nothing about this function requires that the elements be in a set – you could just as well pass in iterators from a vector, or even pointers delineating a array. Also note that there's no requirement that the iterators define the range of an entire container. For example, here's code to print the sum of all the elements in a set<int> between 42 and 137, inclusive:

```
cout << accumulate(mySet.lower_bound(42),
                   mySet.upper_bound(137), 0) << endl;
```

Behind the scenes, accumulate is implemented as a template function that accepts two iterators and simply uses a loop to sum together the values. Here's one possible implementation of accumulate:

```
template<typename InputIterator, typename Type> inline
Type accumulate(InputIterator start, InputIterator stop, Type initial)
{
   while(start != stop)
   {
      initial += *start;
      ++start;
   }
   return initial;
}
```

While some of the syntax specifics might be a bit confusing (notably the template header and the inline keyword), you can still see that the heart of the code is just a standard iterator loop that continues advancing the start iterator forward until it reaches the destination. There's nothing magic about accumulate, and the fact that the function call is a single line of code doesn't change the fact that it still uses a loop to sum all the values together.

If STL algorithms are just functions that use loops behind the scenes, why even bother with them? There are several reasons, the first of which is *simplicity*. With STL algorithms, you can leverage off of code that's already been written for you rather than reinventing the code from scratch. This can be a great time-saver and also leads into the second reason, *correctness*. If you had to rewrite all the algorithms from scratch every time you needed to use them, odds are that at some point you'd slip up and make a mistake. You might, for example, write a sorting routine that accidentally uses < when

you meant > and consequently does not work at all. Not so with the STL algorithms – they've been thoroughly tested and you will work correctly for any given input. The third reason to use algorithms is *speed*. In general, you can assume that if there's an algorithm to solve a problem, it's going to be faster than most code you could write by hand. Through advanced techniques like template specialization and template metaprogramming, STL algorithms are transparently optimized to work as fast as possible. Finally, STL algorithms offer *clarity*. With algorithms, you can immediately tell that a call to `accumulate` adds up numbers in a range. With a for loop that sums up values, you'd have to read each line in the loop before you understood what the code did.

Algorithm Naming Conventions

There are over fifty STL algorithms and memorizing them all would be a chore, to say the least. Fortunately, many of them have common naming conventions so you can recognize algorithms even if you've never encountered them before.

The suffix `_if` on an algorithm (`replace_if`, `count_if`, etc.) means the algorithm will perform a task on elements only if they meet a certain criterion. Functions ending in `_if` require you to pass in a "predicate function" that accepts an element and returns a `bool` indicating whether the element matches the criterion.

Algorithms containing the word `copy` (`remove_copy`, `partial_sort_copy`, etc.) will perform some task on a range of data and store the result in the location pointed at by an extra iterator parameter. With `copy` functions, you'll specify all the normal data for the algorithm plus an extra iterator specifying a destination for the result. We'll cover what this means from a practical standpoint later.

If an algorithm ends in `_n` (`fill_n`, `generate_n`, etc), then it will perform a certain operation `n` times. These functions are useful for cases where the number of times you perform an operation is meaningful, rather than the range over which you perform it.

Reordering Algorithms

While the STL algorithms aren't formally categorized, for this handout it's useful to discuss the different algorithms in terms of their basic functionality. The first major grouping of algorithms we'll talk about are the *reordering algorithms*, algorithms that reorder but do not modify the elements in a container.

Perhaps the most useful of the reordering algorithms is `sort`, which sorts the elements of a container in ascending order. For example, the following code will sort a `vector<int>` from lowest to highest:

```
sort(myVector.begin(), myVector.end());
```

`sort` requires that the iterators you pass in be random-access iterators, so you cannot use `sort` to sort a `map` or `set`. However, since `map` and `set` are always stored in sorted order, this shouldn't be a problem.

By default, `sort` uses the < operator for whatever element types it's sorting, but you can specify a different comparison function if you wish. Whenever you write a comparison function for an STL algorithm, it should accept two parameters representing the elements to compare and return a `bool`

indicating whether the first element is strictly less than the second element (this is sometimes referred to as a "strict weak ordering.") In other words, your callback should mimic the < operator. Note that this is different than the comparison functions you've seen in CS106X, which return an `int` specifying the relation between the two elements.

For example, suppose we had a `vector<placeT>`, where `placeT` was defined as

```
struct placeT
{
      int x;
      int y;
};
```

Then we could `sort` the `vector` only if we wrote a comparison function for `placeT`s as follows:*

```
bool ComparePlaces(placeT one, placeT two)
{
     if(one.x < two.x)
          return true;
     if(one.x > two.x)
          return false;
     return one.y < two.y;
}

sort(myPlaceVector.begin(), myPlaceVector.end(), ComparePlaces);
```

You can also use custom comparison functions to reorder elements even if they have a default comparison function. For example, here is some code that sorts a `vector<string>` by length, ignoring whether the strings are in alphabetical order:

```
bool CompareStringLength(string one, string two)
{
      return one.length() < two.length();
}

sort(myVector.begin(), myVector.end(), CompareStringLength);
```

One last note on comparison functions is that they should either accept the parameters by value or by "reference to `const`." Since we won't cover `const` until next week, for now your comparison functions should accept their parameters by value. Otherwise you can get some pretty ferocious compiler errors.

Another useful reordering function is `random_shuffle`, which randomly scrambles the elements of a container.† Because the scrambling is random, there's no need to pass in a comparison function. Here's some code showing off `random_shuffle`:

```
random_shuffle(myVector.begin(), myVector.end());
```

---

* When we cover operator overloading in several weeks, you'll see how to create functions that `sort` will use automatically.

† Internally, `random_shuffle` calls the C library function `rand` to reorder the elements. Thus you should seed the randomizer using `srand` before calling `random_shuffle`. You can learn more about `srand` from a C or C++ reference.

As with `sort`, the iterators must be random-access iterators, so you can't scramble a `set` or `map`. Then again, since they're sorted containers, you wouldn't want to in the first place.

The last major algorithm in this category is `rotate`, which cycles the elements in a container. For example, given the input container (0, 1, 2, 3, 4, 5), rotating the container around position 2 would result in the container (2, 3, 4, 5, 0, 1).

Searching Algorithms

Commonly you're interested in checking membership in a container. For example, given a `vector`, you might want to know whether or not it contains a specific element. While the `map` and `set` naturally support `find`, `vector`s and `deque`s lack this functionality. Fortunately, you can use STL algorithms to correct this problem.

To search for an element in a container, you can use the `find` function, which works similarly to the `map` and `set` `find` functions except that you must specify an iterator range. For example:

```
if(find(myVector.begin(), myVector.end(), 137) != myVector.end())
    // ... vector contains 137 ...
```

`find` will return an iterator to the first occurrence of an element in a container, so if there are duplicates you'll need to call `find` repeatedly and starting from different indices.

Although you can legally pass `map` and `set` iterators as parameters to `find`, you should never do so. If a container class has a member function with the same name as an STL algorithm, you should use the member function instead of the algorithm because member functions can use information about the container's internal data representation to work much more quickly. Algorithms, however, must work for all iterators and thus can't make any optimizations. As an example, with a `set` containing one million elements, the `set`'s `find` member function can locate elements in around twenty steps using binary search, while the STL `find` function could take up to one million steps to linearly iterate over the entire container. That's a staggering difference and really should hit home how important it is to use member functions over STL algorithms.

Just as a sorted `map` and `set` can use binary search to outperform the linear STL `find` algorithm, if you have a sorted container (for example, a sorted `vector`), you can use the STL algorithm `binary_search` to perform the search in a fraction of the time. For example:

```
// Assume myVector is sorted.
if(binary_search(myVector.begin(), myVector.end(), 137))
     // Found 137
```

Note that `binary_search` doesn't return an iterator to the element – it simply checks to see if it's in the container. Also, as with `sort`, if the container is sorted using a special sorting function, you can pass that function in as a parameter to `binary_search`. However, make sure you're consistent about what comparison function you use, because if you mix them up `binary_search` might not work correctly.

There are also algorithm versions of `lower_bound` and `upper_bound`, which you can use like the `map` and `set` versions. As with `binary_search`, the container must be in sorted order for this algorithm to work correctly.

Set Algorithms

If you'll recall from last week, the `set` container class does not support intersection, union, and subset. However, the STL algorithms provide a useful set of functions that can perform these operations. Unlike the CS106 `Set` functions, the STL algorithms will work on any sorted container type, not just a `set`, so you can intersect two `vector`s, a `set` and a `deque`, or even two `map`s.

Set algorithms need a place to store the result of the set operations, which you specify by providing an iterator to the start of the range that will hold the new values. Be careful, though, because this destination must have enough space to store the result of the operation or you'll begin writing out of bounds. In other words, if the two sets have **m** and **n** elements, respectively, your output container must have space for **m** + **n** elements before you run the algorithm.

Because this restriction can be tricky, it's common to use iterator adapters in conjunction with the set algorithms. If you want to take the union of two sets, rather than allocating enough space for the result, just specify an `insert_iterator` as the destination iterator and the container storing the result of the operation will be automatically grow to contain all of the resulting elements.

There are four major set algorithms: `set_union`, which computes the union of two sets; `set_intersection`, which computes the intersection; `set_difference`, which creates a set of all elements in the first container except for those in the second container, and `set_symmetric_difference`, the set of all elements in either the first or second container but not both. Below is some code showcasing `set_union`, although any of the above four functions can be substituted in:

```
set<int> setOne;
set<int> setTwo;
// Initialize setOne and setTwo
set<int> result;
set_union(setOne.begin(), setOne.end(),  // All of the elements in setOne
          setTwo.begin(), setTwo.end(),  // All of the elements in setTwo
          inserter(result, result.begin())); // Store in result.
```

Two other useful algorithms are `includes` and `equal`. `includes` accepts two sorted iterator ranges and returns whether all elements in the second range are contained in the first range; this is equivalent to testing whether one range is a subset of another. `equal` accepts two sorted iterator ranges and returns whether they're equal. For example:

```
set<int> setOne;
vector<int> setTwo;
// Initialize setOne and setTwo, setTwo must be sorted.
if(includes(setOne.begin(), setOne.end(), setTwo.begin(), setTwo.end()))
    // ... setTwo is a subset of setOne ...
if(equal(setOne.begin(), setOne.end(), setTwo.begin(), setTwo.end()))
    /// ... setOne == setTwo
```

Removal Algorithms

The STL provides several algorithms for removing elements from containers. However, removal algorithms have some idiosyncrasies that can take some time to adjust to.

When working with removal algorithms, the key point to remember is that the algorithms **do not** actually remove elements from containers. This is somewhat counterintuitive – after all, they're called removal algorithms – but makes sense when you think about how algorithms work. Algorithms accept iterators to containers, not containers themselves, and thus do not know how to erase elements from containers. Consider, for example, what would happen if you passed in pointers to a regular C++ array into a removal algorithm. What should the algorithm do to the unused space remaining at the end of the array? While you might say that the algorithm should reallocate the array to only have enough space for the final elements, this approach won't work with static arrays, which can't be resized.

Removal functions work by shuffling down the contents of the container to overwrite all elements that need to be erased. Once finished, they return iterators to the first element not in the modified range. So for example, if you have a `vector` initialized to 0, 1, 2, 3, 3, 3, 4 and then `remove` all instances of the number 3, the resulting `vector` will contain 0, 1, 2, 4, 3, 3, 4 and the function will return an iterator to one spot past the first 4. If you'll notice, the elements in the iterator range starting at `begin` and ending with the element one past the four are the sequence 0, 1, 2, 4 – exactly the range we wanted.

To truly remove elements from a container with the removal algorithms, you can use the container classes' member function `erase` to erase the range of values that aren't in the result. For example, here's a code snippet that removes all copies of the number 137 from a `vector`:

```
myVector.erase(remove(myVector.begin(), myVector.end(), 137),
               myVector.end());
```

Note that we're erasing elements in the range [`start`, `end`), where `start` is the value returned by the `remove` algorithm.

There is another useful removal function, `remove_if`, that removes all elements from a container that satisfy a condition specified as the final parameter. For example, using the `ispunct` function from the header file `<cctype>`, we can write a `StripPunctuation` function that returns a copy of a string with all the punctuation removed:

```
string StripPunctuation(string input)
{
    input.erase(remove_if(input.begin(), input.end(), ispunct),
                input.end());
    return input;
}
```

(Isn't it amazing how much you can do with a single line of code? That's the real beauty of STL algorithms.)

If you're shaky about how to actually remove elements in a container using `remove`, you might want to consider the `remove_copy` and `remove_copy_if` algorithms. These algorithms act just like `remove` and `remove_if`, except that instead of modifying the original range of elements, they copy the elements that aren't removed into another container. While this can be a bit less memory efficient, in some cases it's exactly what you're looking for.

Miscellaneous Algorithms

While there are a number of other useful algorithms, of which three, `copy`, `for_each` and `transform`, we'll cover in the next section.

The `copy` algorithm is a simple STL algorithm that copies a range of iterators into the destination specified by a third iterator parameter. As with the set algorithms, the destination iterator must point to the start of a range capable of holding all of the specified elements. Here is a code snippet that allocates a regular C++ array and copies the contents of a `set` into it:

```
int *dynamicArray = new int[mySet.size()]
copy(mySet.begin(), mySet.end(), dynamicArray);
```

`for_each` is an incredibly useful algorithm that simply calls a function on each element in the specified range. Note that `for_each` doesn't modify the values; rather it simply mimics an iterator loop that calls a function. Currently, `for_each` might not seem all that useful, especially since the function to call must take exactly one parameter, but in a few weeks when we cover functors `for_each` will become an invaluable weapon in your C++ arsenal.

Here is some code that simply iterates over a `vector<int>` and prints out each element:

```
void PrintSingleInt(int value)
{
     cout << value << endl;
}

// Later on, assume myVector is initialized.
for_each(myVector.begin(), myVector.end(), PrintSingleInt);
```

Another useful function is `transform`, which applies a function to a range of elements and stores the result in the specified destination. `transform` is especially elegant when combined with functors, but even without them is useful for a whole range of tasks. For example, consider the `tolower` function, a C library function declared in the header `<cctype>` that accepts a `char` and returns the lowercase representation of that character. Combined with `transform`, this lets us write `ConvertToLowerCase` from strutils.h in two lines of code, one of which is a `return` statement:

```
string ConvertToLowerCase(string text)
{
    transform(text.begin(), text.end(), text.begin(), tolower);
    return text;
}
```

Note that after specifying the range `text.begin()`, `text.end()` we have another call to `text.begin()`. This is because we need to provide an iterator that tells `transform` where to put its output. Since we want to overwrite the old contents of our container with the new values, we simply use `text.begin()` another time to indicate that `transform` should start writing elements to the beginning of the string as it generates them.

There is no requirement that the function you pass to `transform` return elements of the same type as those stored in the container. It's legal to `transform` a set of `strings` into a set of `doubles`, for example.

The following table lists some of the more common STL algorithms. It's by no means an exhaustive list, and you should consult a reference to get a complete list of all the algorithms available to you.

`Type accumulate(InputItr start,`
`InputItr stop,`
`Type value)`
	Returns the sum of the elements in the range [`start`, `stop`) plus the value of `value`.

`bool binary_search(RandomItr start,`
`RandomItr stop,`
`const Type &value)`
	Performs binary search on the sorted range specified by [`start`, `stop`) and returns whether it finds the element `value`. If the elements are sorted using a special comparison function, you must specify the function as the final parameter.

`OutItr copy(InputItr start,`
`InputItr stop,`
`OutItr outputStart)`
	Copies the elements in the range [`start`, `stop`) into the output range starting at `outputStart`. `copy` returns an iterator to one past the end of the range written to.

`size_t count(InputItr start,`
`InputItr end,`
`const Type &value)`
	Returns the number of elements in the range [`start`, `stop`) equal to `value`.

`size_t count_if(InputItr start,`
`InputItr end,`
`PredicateFunction fn)`
	Returns the number of elements in the range [`start`, `stop`) for which `fn` returns true. Useful for determining how many elements have a certain property.

`bool equal(InputItr start1,`
`InputItr stop1,`
`InputItr start2,`
`InputItr stop2)`
	Returns whether the elements in the two sorted ranges defined by [`start1`, `stop1`) and [`start2`, `stop2`) are equal. If you have a special comparison function to compare two elements, you can specify it as the final parameter.

`pair<iterator, iterator>`
`equal_range(InputItr start,`
`InputItr stop,`
`const Type &value)`
	Returns two iterators as a `pair` that defines the sub-range of elements in the sorted range [`start`, `stop`) that are equal to `value`. In other words, every element in the range defined by the returned iterators is equal to `value`. You can specify a special comparison function as a final parameter.

`void fill(InputItr start,`
`InputItr stop,`
`const Type &value)`
	Sets every element in the range [`start`, `stop`) to `value`.

`void fill_n(InputItr start,`
`size_t num,`
`const Type &value)`
	Sets the first `num` elements, starting at `start`, to `value`.

`iterator find(InputItr start,`
`InputItr stop,`
`const Type &value)`
	Returns an iterator to the first element in [`start`, `stop`) that is equal to `value`, or `stop` if the value isn't found. The range doesn't need to be sorted.

`iterator find_if(InputItr start,`
`InputItr stop,`
`PredicateFunc fn)`
	Returns an iterator to the first element in [`start`, `stop`) for which `fn` is true, or `stop` otherwise.

`Function for_each(InputItr start,`
`InputItr stop,`
`Function fn)`
	Calls the function `fn` on each element in the range [`start`, `stop`). We will explore `for_each` in more detail when we talk about functors.

Common STL algorithms,contd.

`bool includes(InputItr start1,`
`InputItr stop1,`
`InputItr start2,`
`InputItr stop2)`
	Returns whether every element in the sorted range [`start2`, `stop2`) is also in [`start1`, `stop1`). If you need to use a special comparison function, you can specify it as the final parameter.

`iterator lower_bound(InputItr start,`
`InputItr stop,`
`const Type &elem)`
	Returns an iterator to the first element greater than or equal to the element `elem` in the sorted range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.

`iterator max_element(InputItr start,` `InputItr stop)`	Returns an iterator to the largest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`iterator min_element(InputItr start,` `InputItr stop)`	Returns an iterator to the smallest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`void random_shuffle(RandomItr start,` `RandomItr stop)`	Randomly reorders the elements in the range [`start`, `stop`) using the `rand` function.

`iterator remove(InputItr start,`
`InputItr stop,`
`const Type &value)`
	Removes all elements in the range [`start`, `stop`) that are equal to `value`. This function will `not` remove elements from a container. To shrink the container, use the container's `erase` function to erase all values in the range [`rvalue`, `end()`), where `rvalue` is the return value of `remove`.

`iterator remove_if(InputItr start,`
`InputItr stop,`
`PredicateFunc fn)`
	Removes all elements in the range [`start`, `stop`) for which `fn` returns true. See `remove` for information about how to actually remove elements from the container.

`void replace(InputItr start,`
`InputItr stop,`
`const Type &toReplace,`
`const Type &replaceWith)`
	Replaces all values in the range [`start`, `stop`) that are equal to `toReplace` with `replaceWith`.

`void replace_if(InputItr start,`
`InputItr stop,`
`PredicateFunction fn,`
`const Type &with)`
	Replaces all elements in the range [`start`, `stop`) for which `fn` returns true with the value `with`.

`iterator rotate(InputItr start,`
`InputItr middle,`
`InputItr stop)`
	Rotates the elements of the container such that the sequence [`middle`, `stop`) is at the front and the range [`start`, `middle`) goes from the new middle to the end. `rotate` returns an iterator to the new position of `start`.

`iterator set_difference(`
`InputItr start1,`
`InputItr stop1,`
`InputItr start2,`
`InputItr stop2,`
`OutItr dest)`
	Stores all elements that are in the sorted range [`start1`, `stop1`) but not in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

Common STL algorithms,contd.

`iterator set_intersection(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in both the sorted range [`start1`, `stop1`) and the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`iterator set_union(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in either the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`iterator set_symmetric_difference(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`), but not both, in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`iterator transform(InputItr start,`
`  InputItr stop,`
`  OutputItr dest,`
`  Function fn)`
	Applies the function `fn` to all of the elements in the range [`start`, `stop`) and stores the result in the range beginning with `dest`. The return value is an iterator one past the end of the last value written.

`iterator upper_bound(InputItr start,`
`  InputItr stop,`
`  const Type &val)`
	Returns an iterator to the first element in the sorted range [`start`, `stop`) that is strictly greater than the value `val`. If you need to specify a special comparison function, you can do so as the final parameter.

More to Explore

While this handout lists some of the more common algorithms, there are many others that are useful in a variety of contexts. Additionally, there are some useful C/C++ library functions that work well with algorithms. If you're interested in maximizing your algorithmic firepower, consider looking into some of these topics:

1. **`<cctype>`**: This handout briefly mentioned the `<cctype>` header, the C runtime library's character type library. `<cctype>` include support for categorizing characters (for example, `isalpha` to return if a character is a letter and `isxdigit` to return if a character is a valid hexadecimal digit) and formatting conversions (`toupper` and `tolower`).
2. **`<cmath>`**: The C mathematics library has all sorts of nifty functions that perform arithmetic operations like `sin`, `sqrt`, and `exp`. Consider looking into these functions if you want to use `transform` on your containers.
3. **Permutation Algorithms**: Because generating different permutations of a container is such a common task, there are two STL algorithms, `next_permutation` and `prev_permutation`, that can generate them for you.
4. **Boost Algorithms**: As with most of the C++ Standard Library, the Boost C++ Libraries have a whole host of useful STL algorithms ready for you to use. One of the more useful Boost algorithm sets is the string algorithms, which extend the functionality of the `find` and `replace` algorithms on `string`s from dealing with single characters to dealing with entire strings.

Practice Problems

Algorithms are ideally suited for solving a wide variety of problems in a small space. Most of the following programming problems have short solutions – see if you can whittle down the space and let the algorithms do the work for you!

1. Write a function `PrintVector` that accepts a `vector<int>` as a parameter and prints its contents separated by space characters. *(Hint: Use an `ostream_iterator` and `copy`)*
2. Using `remove_if` and a custom callback function, write a function `RemoveShortWords` that accepts a `set<string>` and removes all strings of length 3 or less from it. Note that `remove_if` will not interfere with the internal ordering of the `set`, since it preserves the order in which the elements were originally provided.
3. In N-dimensional space, the distance from a point (x_1, x_2, x_3, ... , x_n) to the origin is $\sqrt{(x_1^2+x_2^2+x_3^2+...+x_n^2)}$. Write a function `DistanceToOrigin` that accepts a `vector<double>` representing a point in space and returns the distance from that point to the origin. Do not use any loops – let the algorithms do the heavy lifting for you. *(Hint: Use the `transform` algorithm and a custom callback function to square all of the elements of the `vector`. When we cover functors you'll see a more elegant way to do this.)*
4. Write a function `BiasedSort` that accepts a `vector<string>` by reference and sorts the `vector` lexicographically, except that if the `vector` contains the string "Me First," that string is always at the front of the sorted list.
5. Write a function `CriticsPick` that accepts a `map<string, double>` of movies and their ratings (between 0.0 and 10.0) and returns a `set<string>` of the names of the top ten movies in the `map`. If there are fewer than ten elements in the `map`, then the resulting `set` should contain every single string in the `map`. *(Hint: Remember that all elements in a `map<string, double>` are stored internally as `pair<string, double>`)*