Chapter 7: STL Algorithms

Consider the following problem: suppose that we want to write a program that reads in a list of integers from a file (perhaps representing grades on an assignment), then prints out the average of those values. For simplicity, let's assume that this data is stored in a file called `data.txt` with one integer per line. For example:

***File:* `data.txt`**

```
100
95
92
98
87
88
100
...
```

Here is one simple program that reads in the contents of the file, stores them in an STL `multiset`, computes the average, then prints it out:

```
#include <iostream>
#include <fstream>
#include <set>
using namespace std;

int main() {
    ifstream input("data.txt");
    multiset<int> values;

    /* Read the data from the file. */
    int currValue;
    while (input >> currValue)
        values.insert(currValue);

    /* Compute the average. */
    double total = 0.0;
    for (multiset<int>::iterator itr = values.begin();
         itr != values.end(); ++itr)
        total += *itr;
    cout << "Average is: " << total / values.size() << endl;
}
```

As written, this code is perfectly legal and will work as intended. However, there's something slightly odd about it. If you were to describe what this program needs to do in plain English, it would probably be something like this:

1. Read the contents of the file.
2. Add the values together.
3. Divide by the number of elements.

In some sense, the above code matches this template. The first loop of the program reads in the contents of the file, the second loop sums together the values, and the last line divides by the number of elements. However, the code we've written is somewhat unsatisfactory. Consider this first loop:

```
int currValue;
while (input >> currValue)
    values.insert(currValue);
```

Although the intuition behind this loop is "read the contents of the file into the `multiset`," the way the code is actually written is "create an integer, and then while it's possible to read another element out of the file, do so and insert it into the `multiset`." This is a very *mechanical* means for inserting the values into the `multiset`. Our English description of this process is "read the file contents into the `multiset`," but the actual code is a step-by-step process for extracting data from the file one step at a time and inserting it into the `multiset`.

Similarly, consider this second loop, which sums together the elements of the `multiset`:

```
double total = 0.0;
for (multiset<int>::iterator itr = values.begin(); itr != values.end(); ++itr)
    total += *itr;
```

Again, we find ourselves taking a very mechanical view of the operation. Our English description "sum the elements together" is realized here as "initialize the total to zero, then iterate over the elements of the `multiset`, increasing the total by the value of the current element at each step."

The reason that we must issue commands to the computer in this mechanical fashion is precisely because the computer *is* mechanical – it's a machine for efficiently computing functions. The challenge of programming is finding a way to translate a high-level set of commands into a series of low-level instructions that control the machine. This is often a chore, as the basic operations exported by the computer are fairly limited. But programming doesn't have to be this difficult. As you've seen, we can define new functions in terms of old ones, and can build complex programs out of these increasingly more powerful subroutines. In theory, you could compile an enormous library containing solutions to all nontrivial programming problems. With this library in tow, you could easily write programs by just stitching together these prewritten components.

Unfortunately, there is no one library with the solutions to every programming problem. However, this hasn't stopped the designers of the STL from trying their best to build one. These are the STL algorithms, a library of incredibly powerful routines for processing data. The STL algorithms can't do everything, but what they can do they do fantastically. In fact, using the STL algorithms, it will be possible to rewrite the program that averages numbers in *four lines of code*. This chapter details many common STL algorithms, along with applications. Once you've finished this chapter, you'll have one of the most powerful standard libraries of any programming language at your disposal, and you'll be ready to take on increasingly bigger and more impressive software projects.

Your First Algorithm: `accumulate`

Let's begin our tour of the STL algorithms by jumping in head-first. If you'll recall, the second loop from the averaging program looks like this:

```
double total = 0.0;
for (multiset<int>::iterator itr = values.begin(); itr != values.end(); ++itr)
    total += *itr;
cout << "Average is: " << total / values.size() << endl;
```

This code is entirely equivalent to the following:

```
cout << accumulate(values.begin(), values.end(), 0.0) / values.size() << endl;
```

We've replaced the entire `for` loop with a single call to `accumulate`, eliminating about a third of the code from our original program.

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. It then computes the sum of all of the elements contained in the range of iterators, plus the base value.* What's beautiful about `accumulate` (and the STL algorithms in general) is that `accumulate` can take in iterators of any type. That is, we can sum up iterators from a `multiset`, a `vector`, or `deque`. This means that if you ever find yourself needing to compute the sum of the elements contained in a container, you can pass the `begin()` and `end()` iterators of that container into `accumulate` to get the sum. Moreover, `accumulate` can accept any valid iterator range, not just an iterator range spanning an entire container. For example, if we want to compute the sum of the elements of the `multiset` that are between 42 and 137, inclusive, we could write

```
accumulate(values.lower_bound(42), values.upper_bound(137), 0);
```

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 continuously advances 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 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 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 STL algorithm that performs a task, 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.

---

* There is also a version of `accumulate` that accepts four parameters, as you'll see in the chapter on functors.

Algorithm Naming Conventions

There are over fifty STL algorithms (defined either in `<algorithm>` or in `<numeric>`), 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. For example consider the `count` algorithm and its counterpart `count_if`. `count` accepts a range of iterators and a value, then returns the number of times that the value appears in that range. If we have a `vector<int>` of several integer values, we could print out the number of copies of the number 137 in that `vector` as follows:

```
cout << count(myVec.begin(), myVec.end(), 137) << endl;
```

`count_if`, on the other hand, accepts a range of iterators and a predicate function, then returns the number of times the predicate evaluates to `true` in that range. If we were interested in how number of even numbers are contained in a `vector<int>`, we could could obtain the value as follows. First, we write a predicate function that takes in an `int` and returns whether it's even, as shown here:

```
bool IsEven(int value) {
    return value % 2 == 0;
}
```

We could then use `count_if` as follows:

```
cout << count_if(myVec.begin(), myVec.end(), IsEven) << endl;
```

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` (`generate_n`, `search_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. To give you a better feel for what this means, consider the `fill` and `fill_n` algorithms. Each of these algorithms sets a range of elements to some specified value. For example, we could use `fill` as follows to set every element in a `deque` to have value 0:

```
fill(myDeque.begin(), myDeque.end(), 0);
```

The `fill_n` algorithm is similar to `fill`, except that instead of accepting a range of iterators, it takes in a start iterator and a number of elements to write. For instance, we could set the first ten elements of a `deque` to be zero by calling

```
fill_n(myDeque.begin(), 10, 0);
```

Iterator Categories

If you'll recall from the discussion of the `vector` and `deque` `insert` functions, to specify an iterator to the nth element of a `vector`, we used the syntax `myVector.begin() + n`. Although this syntax is legal in

conjunction with `vector` and `deque`, it is illegal to use + operator with iterators for other container classes like `map` and `set`. At first this may seem strange – after all, there's nothing intuitively wrong with moving a `set` iterator forward multiple steps, but when you consider how the `set` is internally structured the reasons become more obvious. Unlike `vector` and `deque`, the elements in a `map` or `set` are not stored sequentially (usually they're kept in a balanced binary tree). Consequently, to advance an iterator *n* steps forward, the `map` or `set` iterator must take *n* individual steps forward. Contrast this with a `vector` iterator, where advancing forward *n* steps is a simple addition (since all of the `vector`'s elements are stored contiguously). Since the runtime complexity of advancing a `map` or `set` iterator forward *n* steps is linear in the size of the jump, whereas advancing a `vector` iterator is a constant-time operation, the STL disallows the + operator for `map` and `set` iterators to prevent subtle sources of inefficiency.

Because not all STL iterators can efficiently or legally perform all of the functions of every other iterator, STL iterators are categorized based on their relative power. At the high end are *random-access iterators* that can perform all of the possible iterator functions, and at the bottom are the *input* and *output* iterators which guarantee only a minimum of functionality. There are five different types of iterators, each of which is discussed in short detail below.

- **Output Iterators**. Output iterators are one of the two weakest types of iterators. With an output iterator, you can write values using the syntax `*myItr = value` and can advance the iterator forward one step using the ++ operator. However, you cannot read a value from an output iterator using the syntax `value = *myItr`, nor can you use the += or – operators.

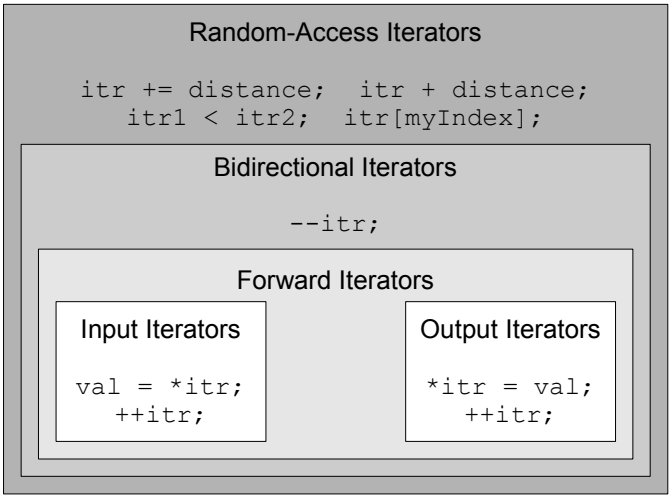
- **Input Iterators**. Input iterators are similar to output iterators except that they read values instead of writing them. That is, you can write code along the lines of `value = *myItr`, but not `*myItr = value`. Moreover, input iterators cannot iterate over the same range twice.

- **Forward Iterators**. Forward iterators combine the functionality of input and output iterators so that most intuitive operations are well-defined. With a forward iterator, you can write both `*myItr = value` and `value = *myItr`. Forward iterators, as their name suggests, can only move forward. Thus `++myItr` is legal, but `--myItr` is not.

- **Bidirectional Iterators**. Bidirectional iterators are the iterators exposed by `map` and `set` and encompass all of the functionality of forward iterators. Additionally, they can move backwards with the decrement operator. Thus it's possible to write `--myItr` to go back to the last element you visited, or even to traverse a list in reverse order. However, bidirectional iterators cannot respond to the + or += operators.

- **Random-Access Iterators**. Don't get tripped up by the name – random-access iterators don't move around randomly. Random-access iterators get their name from their ability to move forward and backward by arbitrary amounts at any point. These are the iterators employed by `vector` and `deque` and represent the maximum possible functionality, including iterator-from-iterator subtraction, bracket syntax, and incrementation with + and +=.

If you'll notice, each class of iterators is progressively more powerful than the previous one – that is, the iterators form a functionality hierarchy. This means that when a library function requires a certain class of iterator, you can provide it any iterator that's at least as powerful. For example, if a function requires a forward iterator, you can provide either a forward, bidirectional, or random-access iterator. The iterator hierarchy is illustrated below:



Why categorize iterators this way? Why not make them all equally powerful? There are several reasons. First, in some cases, certain iterator operations cannot be performed efficiently. For instance, the STL `map` and `set` are layered on top of balanced binary trees, a structure in which it is simple to move from one element to the next but significantly more complex to jump from one position to another arbitrarily. By disallowing the `+` operator on `map` and `set` iterators, the STL designers prevent subtle sources of inefficiency where simple code like `itr + 5` is unreasonably inefficient. Second, iterator categorization allows for better classification of the STL algorithms. For example, suppose that an algorithm takes as input a pair of input iterators. From this, we can tell that the algorithm will not modify the elements being iterated over, and so can feel free to pass in iterators to data that must not be modified under any circumstance. Similarly, if an algorithm has a parameter that is labeled as an output iterator, it should be clear from context that the iterator parameter defines where data generated by the algorithm should be written.

Reordering Algorithms

There are a large assortment of STL algorithms at your disposal, so for this chapter 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 preserve the elements in a container.

Perhaps the most useful of the reordering algorithms is `sort`, which sorts elements in a range 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. In other words, your callback should mimic the < operator. 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.* For example:

```
bool ComparePlaces(placeT one, placeT two) {
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}

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

You can also use custom comparison functions even if a default already exists. 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 haven't covered `const` yet, 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 that uses `random_shuffle` to scramble a `vector`'s elements:

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

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 shouldn't want to do this in the first place.

Internally, `random_shuffle` uses the built-in `rand()` function to generate random numbers. Accordingly, you should use the `srand` function to seed the randomizer before using `random_shuffle`.

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 3 would result in the container (2, 3, 4, 5, 0, 1). The syntax for `rotate` is anomalous in that it accepts three iterators delineating the range and the new front, but in the order *begin*, *middle*, *end*. For example, to rotate a `vector` around its third position, we would write

```
rotate(v.begin(), v.begin() + 2, v.end());
```

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.

---

* When we cover operator overloading in the second half of this text, you'll see how to create functions that `sort` will use automatically.

To search for an element in a container, you can use the `find` function. `find` accepts two iterators delineating a range and a value, then returns an iterator to the first element in the range with that value. If nothing in the range matches, `find` returns the second iterator as a sentinel. For example:

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

Although you can legally pass `map` and `set` iterators as parameters to `find`, you should avoid doing 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 linear 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:

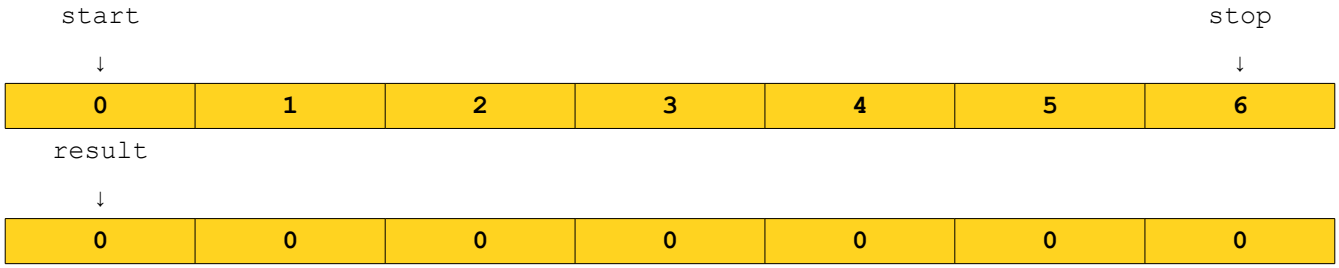
```
/* Assume myVector is sorted. */
if (binary_search(myVector.begin(), myVector.end(), 137)) {
   /* ... Found 137 ... */
}
```

Also, as with `sort`, if the container is sorted using a special comparison 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.

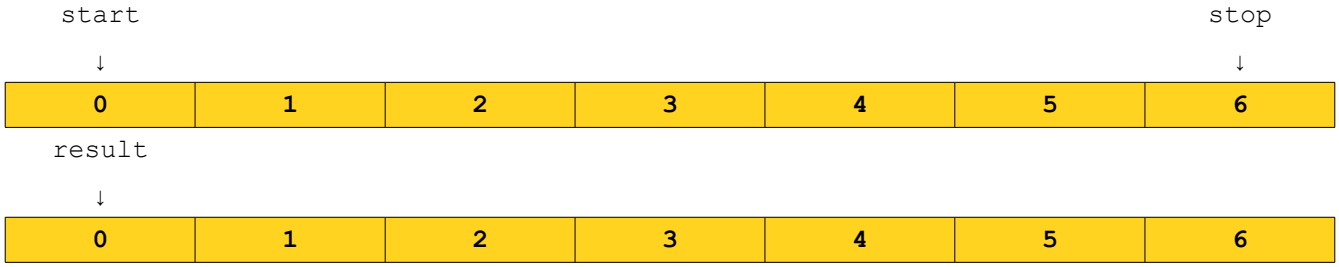
Note that `binary_search` doesn't return an iterator to the element – it simply checks to see if it's in the container. If you want to do a binary search in order to get an iterator to an element, you can use the `lower_bound` algorithm which, like the `map` and `set` `lower_bound` functions, returns an iterator to the first element greater than or equal to the specified value. Note that `lower_bound` might hand back an iterator to a different element than the one you searched for if the element isn't in the range, so be sure to check the return value before using it. As with `binary_search`, the container must be in sorted order for `lower_bound` algorithm to work correctly.

Iterator Adaptors

The algorithms that we've encountered so far do not produce any new data ranges. The `sort` algorithm rearranges data without generating new values. `binary_search` and `accumulate` scan over data ranges, but yield only a single value. However, there are a great many STL algorithms that take in ranges of data and produce new data ranges at output. As a simple example, consider the `copy` algorithm. At a high level, `copy` takes in a range of data, then duplicates the values in that range at another location. Concretely, `copy` takes in three parameters – two input iterators defining a range of values to copy, and an output iterator indicating where the data should be written. For example, given the following setup:



After calling `copy(start, stop, result)`, the result is as follows:



When using algorithms like `copy` that generate a range of data, you *must* make sure that the destination has enough space to hold the result. Algorithms that generate data ranges work by *overwriting* elements in the range beginning with the specified iterator, and if your output iterator points to a range that doesn't have enough space the algorithms will write off past the end of the range, resulting in undefined behavior. But here we reach a wonderful paradox. When running an algorithm that generates a range of data, you must make sure that sufficient space exists to hold the result. However, in some cases you can't tell how much data is going to be generated until you actually run the algorithm. That is, the only way to determine how much space you'll need is to run the algorithm, which might result in undefined behavior because you didn't allocate enough space.

To break this cycle, we'll need a special set of tools called *iterator adaptors*. Iterator adaptors (defined in the `<iterator>` header) are objects that act like iterators – they can be dereferenced with `*` and advanced forward with `++` – but which don't actually point to elements of a container. To give a concrete example, let's consider the `ostream_iterator`. `ostream_iterator`s are objects that look like output iterators. That is, you can dereference them using the `*` operator, advance them forward with the `++` operator, etc. However, `ostream_iterator`s don't actually point to elements in a container. Whenever you dereference an `ostream_iterator` and assign a value to it, that value is printed to a specified output stream, such as `cout` or an `ofstream`. Here's some code showing off an `ostream_iterator`; the paragraph after it explores how it works in a bit more detail:

```
/* Declare an ostream_iterator that writes ints to cout. */
ostream_iterator<int> myItr(cout, " ");

/* Write values to the iterator.  These values will be printed to cout. */
*myItr = 137; // Prints 137 to cout
++myItr;

*myItr = 42;  // Prints 42 to cout
++myItr
```

If you compile and run this code, you will notice that the numbers 137 and 42 get written to the console, separated by spaces. Although it *looks* like you're manipulating the contents of a container, you're actually writing characters to the `cout` stream.

Let's consider this code in a bit more detail. If you'll notice, we declared the `ostream_iterator` by writing

```
ostream_iterator<int> myItr(cout, " ");
```

There are three important pieces of data in this line of code. First, notice that `ostream_iterator` is a parameterized type, much like the `vector` or `set`. In the case of `ostream_iterator`, the template argument indicates what sorts of value will be written to this iterator. That is, an `ostream_iterator<int>` writes `int`s into a stream, while an `ostream_iterator<string>` would write strings. Second, notice that when we created the `ostream_iterator`, we passed it two pieces of information. First, we gave the `ostream_iterator` a stream to write to, in this case `cout`. Second, we gave it a *separator string*, in our case a string holding a single space. Whenever a value is written to an `ostream_iterator`, that value is pushed into the specified stream, followed by the separator string.

At this point, iterator adaptors might seem like little more than a curiosity. Sure, we can use an `ostream_iterator` to write values to `cout`, but we could already do that directly with `cout`. So what makes the iterator adaptors so useful? The key point is that iterator adaptors are *iterators*, and so they can be used in conjunction with the STL algorithms. Whenever an STL algorithm expects a regular iterator, you can supply an iterator adaptor instead to "trick" the algorithm into performing some complex task when it believes it's just writing values to a range. For example, let's revisit the `copy` algorithm now that we have `ostream_iterator`s. What happens if we use `copy` to copy values from a container to an `ostream_iterator`? That is, what is the output of the following code:

```
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
```

This code copies all of the elements from the `myVector` container to the range specified by the `ostream_iterator`. Normally, `copy` would duplicate the values from `myVector` at another location, but since we've written the values to an `ostream_iterator`, this code will instead print all of the values from the `vector` to `cout`, separated by spaces. This means that this single line of code prints out `myVector`!

Of course, this is just one of many iterator adaptors. We initially discussed iterator adaptors as a way to break the "vicious cycle" where algorithms need space to hold their results, but the amount of space needed can only be calculated by running the algorithm. To resolve this issue, the standard library provides a collection of special iterator adapters called *insert iterators*. These are output iterators that, when written to, insert the value into a container using one of the `insert`, `push_back`, or `push_front` functions. As a simple example, let's consider the `back_insert_iterator`. `back_insert_iterator` is an iterator that, when written to, calls `push_back` on a specified STL sequence containers (i.e. `vector` or `deque`) to store the value. For example, consider the following code snippet:

```
vector<int> myVector; /* Initially empty */

/* Create a back_insert_iterator that inserts values into myVector. */
back_insert_iterator< vector<int> > itr(myVector);

for (int i = 0; i < 10; ++i) {
    *itr = i; // "Write" to the back_insert_iterator, appending the value.
    ++itr;
}

/* Print the vector contents; this displays 0 1 2 3 4 5 6 7 8 9 */
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
```

This code is fairly dense, so let's go over it in some more detail. The first line simply creates an empty `vector<int>`. The next line is

```
back_insert_iterator< vector<int> > itr(myVector);
```

This code creates a `back_insert_iterator` which inserts into a `vector<int>`. This syntax might be a bit strange, since the iterator type is parameterized over the type of the *container* it inserts into, not the type of the elements stored in that container. Moreover, notice that we indicated to the iterator that it should insert into the `myVector` container by surrounding the container name in parentheses. From this point, any values written to the `back_insert_iterator` will be stored inside of `myVector` by calling `push_back`.

We then have the following loop, which indirectly adds elements to the `vector`:

```
for (int i = 0; i < 10; ++i) {
    *itr = i; // "Write" to the back_insert_iterator, appending the value.
    ++itr;
}
```

Here, the line `*itr = i` will implicitly call `myVector.push_back(i)`, adding the value to the `vector`. Thus, when we encounter the final line:

```
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
```

the call to `copy` will print out the numbers 0 through 9, inclusive, since they've been stored in the `vector`.

In practice, it is rare to see `back_insert_iterator` used like this. This type of iterator is almost exclusively used as a parameter to STL algorithms that need a place to store a result. For example, consider the `reverse_copy` algorithm. Like `copy`, `reverse_copy` takes in three iterators, two delineating an input range and one specifying a destination, then copies the elements from the input range to the destination. However, unlike the regular `copy` algorithm, `reverse_copy` copies the elements in reverse order. For example, using `reverse_copy` to copy the sequence 0, 1, 2, 3, 4 to a destination would cause the destination range to hold the sequence 4, 3, 2, 1, 0. Suppose that we are interested in using the `reverse_copy` algorithm to make a copy of a `vector` with the elements in reverse order as the original. Then we could do so as follows:

```
vector<int> original = /* ... */
vector<int> destination;
reverse_copy(original.begin(), original.end(),
             back_insert_iterator< vector<int> >(destination));
```

The syntax `back_insert_iterator<vector<int> >` is admittedly bit clunky, and fortunately there's a shorthand. To create a `back_insert_iterator` that inserts elements into a particular container, you can write

```
back_inserter(container);
```

Thus the above code with `reverse_copy` could be rewritten as

```
vector<int> original = /* ... */
vector<int> destination;
reverse_copy(original.begin(), original.end(), back_inserter(destination));
```

This is much cleaner than the original and is likely to be what you'll see in practice.

The `back_inserter` is a particularly useful container when you wish to store the result of an operation in a `vector` or `deque`, but cannot be used in conjunction with `map` or `set` because those containers do not

support the `push_back` member function. For those containers, you can use the more general `insert_iterator`, which insert elements into arbitrary positions in a container. A great example of `insert_iterator` in action arises when computing the union, intersection, or difference of two sets. Mathematically speaking, the *union* of two sets is the set of elements contained in *either* of the sets, the *intersection* of two sets is the set of elements contained in *both* of the sets, and the *difference* of two sets is the set of elements contained in the first set but not in the second. These operations are exported by the STL algorithms as `set_union`, `set_intersection`, and `set_difference`. These algorithms take in five parameters – two pairs of iterator ranges defining what ranges to use as the input sets, along with one final iterator indicating where the result should be written. As with all STL algorithms, the set algorithms assume that the destination range has enough space to store the result of the operation, and again we run into a problem because we cannot tell how many elements will be produced by the algorithm. This is an ideal spot for an `insert_iterator`. Given two sets `one` and `two`, we can compute the union of those two `set`s as follows:

```
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.
```

Notice that the last parameter is `inserter(result, result.begin())`. This is an insert iterator that inserts its elements into the `result` set. For somewhat technical reasons, when inserting elements into a `set`, you must specify both the container and the container's `begin` iterator as parameters, though the generated elements will be stored in sorted order.

All of the iterator adaptors we've encountered so far have been used to channel the output of an algorithm to a location other than an existing range of elements. `ostream_iterator` writes values to streams, `back_insert_iterator` invokes `push_back` to make space for its elements, etc. However, there is a particularly useful iterator adapter, the `istream_iterator`, which is an *input* iterator. That is, `istream_iterator`s can be used to provide data as inputs to particular STL algorithms. As its name suggests, `istream_iterator` can be used to read values from a stream as if it were a container of elements. To illustrate `istream_iterator`, let's return to the example from the start of this chapter. If you'll recall, we wrote a program that read in a list of numbers from a file, then computed their average. In this program, we read in the list of numbers using the following `while` loop:

```
int currValue;
while (input >> currValue)
    values.insert(currValue);
```

Here, `values` is a `multiset<int>`. This code is equivalent to the following, which uses the STL `copy` algorithm in conjunction with an `inserter` and two `istream_iterator`s:

```
copy(istream_iterator<int>(input), istream_iterator<int>(),
     inserter(values, values.begin()));
```

This is perhaps the densest single line of code we've encountered yet, so let's dissect it to see how it works. Recall that the `copy` algorithm copies the values from an iterator range and stores them in the range specified by the destination iterator. Here, our destination is an `inserter` that adds elements into the `values` `multiset`. Our input is the pair of iterators

```
istream_iterator<int>(input), istream_iterator<int>()
```

What exactly does this mean? Whenever a value is read from an `istream_iterator`, the iterator uses the stream extraction operator `>>` to read a value of the proper type from the input stream, then returns it.

Consequently, the iterator `istream_iterator<int>(input)` is an iterator that reads `int` values out of the stream `input`. The second iterator, `istream_iterator<int>()`, is a bit stranger. This is a special `istream_iterator` called the *end-of-stream iterator*. When defining ranges with STL iterators, it is always necessary to specify two iterators, one for the beginning of the range and one that is one past the end of it. When working with STL containers this is perfectly fine, since the size of the container is known. However, when working with streams, it's unclear exactly how many elements that stream will contain. If the stream is an `ifstream`, the number of elements that can be read depends on the contents of the file. If the stream is `cin`, the number of elements that can be read depends on how many values the user decides to enter. To get around this, the STL designers used a bit of a hack. When reading values from a stream with an `istream_iterator`, whenever no more data is available in the stream (either because the stream entered a fail state, or because the end of the file was reached), the `istream_iterator` takes on a special value which indicates "there is no more data in the stream." This value can be formed by constructing an `istream_iterator` without specifying what stream to read from. Thus in the code

```
copy(istream_iterator<int>(input), istream_iterator<int>(),
     inserter(values, values.begin()));
```

the two `istream_iterator`s define the range from the beginning of the input stream up until no more values can be read from the stream.

The following table lists some of the more common iterator adapters and provides some useful context. You'll likely refer to this table most when writing code that uses algorithms.

`back_insert_iterator<Container>`	`back_insert_iterator<vector<int> > itr(myVector);`
	`back_insert_iterator<deque<char> > itr = back_inserter(myDeque);`
	
	An output iterator that stores elements by calling `push_back` on the specified container. You can declare `back_insert_iterator`s explicitly, or can create them with the function `back_inserter`.

`front_insert_iterator<Container>`	`front_insert_iterator<deque<int> > itr(myIntDeque);`
	`front_insert_iterator<deque<char> > itr = front_inserter(myDeque);`
	
	An output iterator that stores elements by calling `push_front` on the specified container. Since the container must have a `push_front` member function, you cannot use a `front_insert_iterator` with a `vector`. As with `back_insert_iterator`, you can create `front_insert_iterator`s with the the `front_inserter` function.

`insert_iterator<Container>`	`insert_iterator<set<int> > itr(mySet, mySet.begin());`
	`insert_iterator<set<int> > itr = inserter(mySet, mySet.begin());`
	
	An output iterator that stores its elements by calling `insert` on the specified container to insert elements at the indicated position. You can use this iterator type to insert into any container, especially `set`. The special function `inserter` generates `insert_iterator`s for you.

`ostream_iterator<Type>`	`ostream_iterator<int> itr(cout, " ");`
	`ostream_iterator<char> itr(cout);`
	`ostream_iterator<double> itr(myStream, "\n");`
	
	An output iterator that writes elements into an output stream. In the constructor, you must initialize the `ostream_iterator` to point to an `ostream`, and can optionally provide a separator string written after every element.

`istream_iterator<Type>`	`istream_iterator<int> itr(cin);  // Reads from cin`
	`istream_iterator<int> endItr;  // Special end value`
	
	An input iterator that reads values from the specified `istream` when dereferenced. When `istream_iterator`s reach the end of their streams (for example, when reading from a file), they take on a special "end" value that you can get by creating an `istream_iterator` with no parameters. `istream_iterator`s are susceptible to stream failures and should be used with care.

`ostreambuf_iterator<char>`	`ostreambuf_iterator<char> itr(cout); // Write to cout` An output iterator that writes raw character data to an output stream. Unlike `ostream_iterator`, which can print values of any type, `ostreambuf_iterator` can only write individual characters. `ostreambuf_iterator` is usually used in conjunction with `istreambuf_iterator`.

`istreambuf_iterator<char>`	`istreambuf_iterator<char> itr(cin); // Read data from cin`
	`istreambuf_iterator<char> endItr;  // Special end value`
	
	An input iterator that reads unformatted data from an input stream. `istreambuf_iterator` always reads in character data and will not skip over whitespace. Like `istream_iterator`, `istreambuf_iterator`s have a special iterator constructed with no parameters which indicates "end of stream." `istreambuf_iterator` is used primarily to read raw data from a file for processing with the STL algorithms.

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.

Despite their name, removal algorithms **do not** actually remove elements from containers. This is somewhat counterintuitive but makes sense when you think about how algorithms work. Algorithms accept *iterators*, not *containers*, and thus do not know how to erase elements from containers. 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 class 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 [*, end), where * 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.

Other Noteworthy Algorithms

The past few sections have focused on common genera of algorithms, picking out representatives that illustrate the behavior of particular algorithm classes. However, there are many noteworthy algorithms that we have not discussed yet. This section covers several of these algorithms, including useful examples.

A surprisingly useful algorithm is transform, which applies a function to a range of elements and stores the result in the specified destination. transform accepts four parameters – two iterators delineating an input range, an output iterator specifying a destination, and a callback function, then stores in the output destination the result of applying the function to each element in the input range. As with other algorithms, transform assumes that there is sufficient storage space in the range pointed at by the destination iterator, so make sure that you have sufficient space before transforming a range.

transform is particularly 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 specify text.begin() another time to indicate that transform should start writing elements to the beginning of the string as it generates them.

---

* On some compilers, this code will not compile as written. See the later section on compatibility issues for more information.

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 `string`s into a set of `double`s, for example.

Most of the algorithms we've seen so far operate on entire ranges of data, but not all algorithms have this property. One of the most useful (and innocuous-seeming) algorithms is `swap`, which exchanges the values of two variables. We first encountered `swap` two chapters ago when discussing sorting algorithms, but it's worth repeating. Several advanced C++ techniques hinge on `swap`'s existence, and you will almost certainly encounter it in your day-to-day programming even if you eschew the rest of the STL.

Two last algorithms worthy of mention are the `min_element` and `max_element` algorithms. These algorithms accept as input a range of iterators and return an iterator to the largest element in the range. As with other algorithms, by default the elements are compared by `<`, but you can provide a binary comparison function to the algorithms as a final parameter to change the default comparison order.

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)`
	Returns whether elements contained in the range defined by [`start1`, `stop1`) and the range beginning with start2 are `equal`. If you have a special comparison function to compare two elements, you can specify it as the final parameter.

`pair<RandomItr, RandomItr>`
`equal_range(RandomItr start,`
`RandomItr 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(ForwardItr start,`
`ForwardItr stop,`
`const Type& value)`
	Sets every element in the range [`start`, `stop`) to `value`.

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

`InputItr 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.

`InputItr 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`).
`void generate(ForwardItr start, ForwardItr stop, Generator fn);`	Calls the zero-parameter function `fn` once for each element in the range [`start`, `stop`), storing the return values in the range.
`void generate_n(OutputItr start, size_t n, Generator fn);`	Calls the zero-parameter function `fn` `n` times, storing the results in the range beginning with `start`.
`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.
`Type inner_product(InputItr start1, InputItr stop1, InputItr start2, Type initialValue)`	Computes the inner product of the values in the range [start1, stop1) and [start2, start2 + (stop1 – start1)). The inner product is the value $\sum_{i=1}^{n} a_i b_i + initialValue$, where a_i and b_i denote the ith elements of the first and second range.
`bool lexicographical_compare(InputItr s1, InputItr s2, InputItr t1, InputItr t2)`	Returns whether the range of elements defined by [`s1`, `s2`) is lexicographically less than [`t1`, `t2`); that is, if the first range precedes the second in a "dictionary ordering."
`InputItr 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.
`InputItr 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.
`InputItr 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.

`bool next_permutation(BidirItr start, BidirItr stop)`	Given a range of elements [`start`, `stop`), modifies the range to contain the next lexicographically higher permutation of those elements. The function then returns whether such a permutation could be found. It is common to use this algorithm in a `do ... while` loop to iterate over all permutations of a range of data, as shown here:
	
	`sort(range.begin(), range.end());`
	`do {`
	`  /* ... process ... */`
	`}while(next_permutation(range.begin(), range.end()));`

`bool prev_permutation(BidirItr start, BidirItr stop)`	Given a range of elements [`start`, `stop`), modifies the range to contain the next lexicographically lower permutation of those elements. The function then returns whether such a permutation could be found.
`void random_shuffle(RandomItr start, RandomItr stop)`	Randomly reorders the elements in the range [`start`, `stop`).

`ForwardItr remove(ForwardItr start, ForwardItr 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 [`retValue`, `end()`), where `retValue` is the return value of `remove`.
`ForwardItr remove_if(ForwardItr start, ForwardItr 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(ForwardItr start, ForwardItr 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(ForwardItr start, ForwardItr stop, PredicateFunction fn, const Type& with)`	Replaces all elements in the range [`start`, `stop`) for which `fn` returns true with the value `with`.
`ForwardItr rotate(ForwardItr start, ForwardItr middle, ForwardItr 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`.
`ForwardItr search(ForwardItr start1, ForwardItr stop1, ForwardItr start2, ForwardItr stop2)`	Returns whether the sequence [`start2`, `stop2`) is a subsequence of the range [`start1`, `stop1`). To compare elements by a special comparison function, specify it as a final parameter.
`InputItr 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.
`InputItr 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.
`InputItr 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.
`InputItr 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.
`void swap(Value& one, Value& two)`	Swaps the values of `one` and `two`.
`ForwardItr swap_ranges(ForwardItr start1, ForwardItr stop1, ForwardItr start2)`	Swaps each element in the range [`start1`, `stop1`) with the correspond elements in the range starting with `start2`.

`OutputItr 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.
`RandomItr upper_bound(RandomItr start, RandomItr 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.

A Word on Compatibility

The STL is ISO-standardized along with the rest of C++. Ideally, this would mean that all STL implementations are uniform and that C++ code that works on one compiler should work on any other compiler. Unfortunately, this is not the case. No compilers on the market fully adhere to the standard, and almost universally compiler writers will make minor changes to the standard that decrease portability.

Consider, for example, the `ConvertToLowerCase` function from earlier in the section:

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

This code will compile in Microsoft Visual Studio, but not in Xcode or the popular Linux compiler g++. The reason is that there are *two* `tolower` functions – the original C `tolower` function exported by `<cctype>` and a more modern `tolower` function exported by the `<locale>` header. Unfortunately, Xcode and g++ cannot differentiate between the two functions, so the call to `transform` will result in a compiler error. To fix the problem, you must explicitly tell C++ which version of `tolower` you want to call as follows:

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

Here, the strange-looking `::` syntax is the *scope-resolution operator* and tells C++ that the `tolower` function is the original C function rather than the one exported by the `<locale>` header. Thus, if you're using Xcode or g++ and want to use the functions from `<cctype>`, you'll need to add the `::`.

Another spot where compatibility issues can lead to trouble arises when using STL algorithms with the STL `set`. Consider the following code snippet, which uses `fill` to overwrite all of the elements in an STL `set` with the value 137:

```
fill(mySet.begin(), mySet.end(), 137);
```

This code will compile in Visual Studio, but will not under g++. Recall from the second chapter on STL containers that manipulating the contents of an STL `set` in-place can destroy the set's internal ordering. Visual Studio's implementation of `set` will nonetheless let you modify `set` contents, even in situations like the above where doing so is unsafe. g++, however, uses an STL implementation that treats all `set` iterators as read-only. Consequently, this code won't compile, and in fact will cause some particularly nasty compiler errors.

When porting C++ code from one compiler to another, you might end up with inexplicable compiler errors. If you find some interesting C++ code online that doesn't work on your compiler, it doesn't necessarily

mean that the code is invalid; rather, you might have an overly strict compiler or the online code might use an overly lenient one.

Extended Example: Palindromes

> *A man, a plan, a caret, a ban, a myriad, a sum, a lac, a liar, a hoop, a pint, a catalpa, a gas, an oil, a bird, a yell, a vat, a caw, a pax, a wag, a tax, a nay, a ram, a cap, a yam, a gay, a tsar, a wall, a car, a luger, a ward, a bin, a woman, a vassal, a wolf, a tuna, a nit, a pall, a fret, a watt, a bay, a daub, a tan, a cab, a datum, a gall, a hat, a tag, a zap, a say, a jaw, a lay, a wet, a gallop, a tug, a trot, a trap, a tram, a torr, a caper, a top, a tonk, a toll, a ball, a fair, a sax, a minim, a tenor, a bass, a passer, a capital, a rut, an amen, a ted, a cabal, a tang, a sun, an ass, a maw, a sag, a jam, a dam, a sub, a salt, an axon, a sail, an ad, a wadi, a radian, a room, a rood, a rip, a tad, a pariah, a revel, a reel, a reed, a pool, a plug, a pin, a peek, a parabola, a dog, a pat, a cud, a nu, a fan, a pal, a rum, a nod, an eta, a lag, an eel, a batik, a mug, a mot, a nap, a maxim, a mood, a leek, a grub, a gob, a gel, a drab, a citadel, a total, a cedar, a tap, a gag, a rat, a manor, a bar, a gal, a cola, a pap, a yaw, a tab, a raj, a gab, a nag, a pagan, a bag, a jar, a bat, a way, a papa, a local, a gar, a baron, a mat, a rag, a gap, a tar, a decal, a tot, a led, a tic, a bard, a leg, a bog, a burg, a keel, a doom, a mix, a map, an atom, a gum, a kit, a baleen, a gala, a ten, a don, a mural, a pan, a faun, a ducat, a pagoda, a lob, a rap, a keep, a nip, a gulp, a loop, a deer, a leer, a lever, a hair, a pad, a tapir, a door, a moor, an aid, a raid, a wad, an alias, an ox, an atlas, a bus, a madam, a jag, a saw, a mass, an anus, a gnat, a lab, a cadet, an em, a natural, a tip, a caress, a pass, a baronet, a minimax, a sari, a fall, a ballot, a knot, a pot, a rep, a carrot, a mart, a part, a tort, a gut, a poll, a gateway, a law, a jay, a sap, a zag, a tat, a hall, a gamut, a dab, a can, a tabu, a day, a batt, a waterfall, a patina, a nut, a flow, a lass, a van, a mow, a nib, a draw, a regular, a call, a war, a stay, a gam, a yap, a cam, a ray, an ax, a tag, a wax, a paw, a cat, a valley, a drib, a lion, a saga, a plat, a catnip, a pooh, a rail, a calamus, a dairyman, a bater, a canal – Panama!*
>
> – Dan Hoey [Pic96]

It is fitting to conclude our whirlwind tour of the STL with an example showcasing exactly how concise and powerful well-written STL code can be. This example is shorter than the others in this book, but should nonetheless illustrate how the different library pieces all fit together. Once you've finished reading this chapter, you should have a solid understanding of how the STL and streams libraries can come together beautifully to elegantly solve a problem.

Palindromes

A *palindrome* is a word or phrase that is the same when read forwards or backwards, such as "racecar" or "Malayalam." It is customary to ignore spaces, punctuation, and capitalization when reading palindromes, so the phrase "Mr. Owl ate my metal worm" would count as a palindrome, as would "Go hang a salami! I'm a lasagna hog."

Suppose that we want to write a function `IsPalindrome` that accepts a `string` and returns whether or not the string is a palindrome. Initially, we'll assume that spaces, punctuation, and capitalization are all significant in the string, so "Party trap" would not be considered a palindrome, though "Part y traP" would. Don't worry – we'll loosen this restriction in a bit. Now, we want to verify that the string is the same when read forwards and backwards. There are many possible ways to do this. Prior to learning the STL, we might have written this function as follows:

```
bool IsPalindrome(string input) {
    for(int k = 0; k < input.size() / 2; ++k)
        if(input[k] != input[input.length() - 1 - k])
            return false;
    return true;
}
```

That is, we simply iterate over the first half of the string checking to see if each character is equal to its respective character on the other half of the string. There's nothing wrong with the approach, but it feels too *mechanical*. The high-level operation we're modeling asks whether the first half of the string is the same forwards as the second half is backwards. The code we've written accomplishes this task, but has to explicitly walk over the characters from start to finish, manually checking each pair. Using the STL, we can accomplish the same result as above without explicitly spelling out the details of how to check each character.

There are several ways we can harness the STL to solve this problem. For example, we could use the STL `reverse` algorithm to create a copy of the string in reverse order, then check if the string is equal to its reverse. This is shown here:

```
bool IsPalindrome(string input) {
    string reversed = input;
    reverse(input.begin(), input.end());
    return reversed == input;
}
```

This approach works, but requires us to create a copy of the string and is therefore less efficient than our original implementation. Can we somehow emulate the functionality of the initial `for` loop using iterators? The answer is yes, thanks to `reverse_iterator`s. Every STL container class exports a type `reverse_iterator` which is similar to an iterator except that it traverses the container backwards. Just as the `begin` and `end` functions define an iterator range over a container, the `rbegin` and `rend` functions define a `reverse_iterator` range spanning a container.

Let's also consider the the STL `equal` algorithm. `equal` accepts three inputs – two iterators delineating a range and a third iterator indicating the start of a second range – then returns whether the two ranges are equal. Combined with `reverse_iterator`s, this yields the following *one-line implementation* of `IsPalindrome`:

```
bool IsPalindrome(string input) {
    return equal(input.begin(), input.begin() + input.size() / 2,
                 input.rbegin());
}
```

This is a remarkably simple approach that is identical to what we've written earlier but much less verbose. Of course, it doesn't correctly handle capitalization, spaces, or punctuation, but we can take care of that with only a few more lines of code. Let's begin by stripping out everything from the string except for alphabetic characters. For this task, we can use the STL `remove_if` algorithm, which accepts as input a range of iterators and a predicate, then modifies the range by removing all elements for which the predicate returns true. Like its partner algorithm `remove`, `remove_if` doesn't actually remove the elements from the sequence (see the last chapter for more details), so we'll need to `erase` the remaining elements afterwards.

Because we want to eliminate all characters from the string that are not alphabetic, we need to create a predicate function that accepts a character and returns whether it is not a letter. The header file

<cctype> exports a helpful function called isalpha that returns whether a character *is* a letter. This is the opposite what we want, so we'll create our own function which returns the negation of isalpha:*

```
bool IsNotAlpha(char ch) {
    return !isalpha(ch);
}
```

We can now strip out nonalphabetic characters from our input string as follows:

```
bool IsPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlpha),
                input.end());
    return equal(input.begin(), input.begin() + input.size() / 2,
                 input.rbegin());
}
```

Finally, we need to make sure that the string is treated case-insensitively, so inputs like "RACEcar" are accepted as palindromes. Using the code developed in the chapter on algorithms, we can convert the string to uppercase after stripping out everything except characters, yielding this final version of IsPalindrome:

```
bool IsPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlpha),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);
    return equal(input.begin(), input.begin() + input.size() / 2,
                 input.rbegin());
}
```

This function is remarkable in its elegance and terseness. In *three lines of code* we've stripped out all of the characters in a string that aren't letters, converted what's left to upper case, and returned whether the string is the same forwards and backwards. This is the STL in action, and I hope that you're beginning to appreciate the power of the techniques you've learned over the past few chapters.

Before concluding this example, let's consider a variant on a palindrome where we check whether the *words* in a phrase are the same forwards and backwards. For example, "Did mom pop? Mom did!" is a palindrome both with respect to its letters and its words, while "This is this" is a phrase that is not a palindrome but is a word-palindrome. As with regular palindromes, we'll ignore spaces and punctuation, so "It's an its" counts as a word-palindrome even though it uses two different forms of the word its/it's. The machinery we've developed above works well for entire strings; can we modify it to work on a word-by-word basis?

In some aspects this new problem is similar to the original. We still to ignore spaces, punctuation, and capitalization, but now need to treat words rather than letters as meaningful units. There are many possible algorithms for checking this property, but one solution stands out as particularly good. The idea is as follows:

1. Clean up the input: strip out everything except letters *and spaces*, then convert the result to upper case.
2. Break up the input into a list of words.
3. Return whether the list is the same forwards and backwards.

In the first step, it's important that we preserve the spaces in the original input so that we don't lose track of word boundaries. For example, we would convert the string "Hello? Hello!? HELLO?" into "HELLO

* When we cover the <functional> library in the second half of this book, you'll see a simpler way to do this.

HELLO HELLO" instead of "HELLOHELLOHELLO" so that we can recover the individual words in the second step. Using a combination of the `isalpha` and `isspace` functions from `<cctype>` and the convert-to-upper-case code used above, we can preprocess the input as shown here:

```
bool IsNotAlphaOrSpace(char ch) {
    return !isalpha(ch) && !isspace(ch);
}

bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);
    /* ... */
}
```

At this point the string `input` consists of whitespace-delimited strings of uniform capitalization. We now need to tokenize the input into individual words. This would be tricky were it not for `stringstream`. Recall that when reading a `string` out of a stream using the stream extraction operator (`>>`), the stream treats whitespace as a delimiter. Thus if we funnel our string into a `stringstream` and then read back individual strings, we'll end up with a tokenized version of the input. Since we'll be dealing with an arbitrarily-long list of strings, we'll store the resulting list in a `vector<string>`, as shown here:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    /* ... */
}
```

Now, what is the easiest way to read strings out of the stream until no strings remain? We could do this manually, as shown here:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    string token;
    while(tokenizer >> token)
        tokens.push_back(token);
}
```

This code is correct, but it's bulky and unsightly. The problem is that it's just too *mechanical*. We want to insert all of the tokens from the `stringstream` into the `vector`, but as written it's not clear that this is what's happening. Fortunately, there is a much, *much* easier way to solve this problem thanks to `istream_iterator`. Recall that `istream_iterator` is an iterator adapter that lets you iterate over an input stream as if it were a range of data. Using `istream_iterator` to wrap the stream operations and the `vector`'s `insert` function to insert a range of data, we can rewrite this entire loop in one line as follows:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    tokens.insert(tokens.begin(),
                  istream_iterator<string>(tokenizer),
                  istream_iterator<string>());
}
```

Recall that two `istream_iterator`s are necessary to define a range, and that an `istream_iterator` constructed with no arguments is a special "end of stream" iterator. This one line of code replaces the entire loop from the previous implementation, and provided that you have some familiarity with the STL this second version is also easier to read.

The last step in this process is to check if the sequence of strings is the same forwards and backwards. But we already know how to do this – we just use `equal` and a `reverse_iterator`. Even though the original implementation applied this technique to a `string`, we can use the same pattern here on a `vector<string>` because all the container classes are designed with a similar interface. Remarkable, isn't it?

The final version of `IsWordPalindrome` is shown here:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    tokens.insert(tokens.begin(),
                  istream_iterator<string>(tokenizer),
                  istream_iterator<string>());
    return equal(tokens.begin(), tokens.begin() + tokens.size() / 2,
                 tokens.rbegin());
}
```

More to Explore

While this chapter 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 chapter 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. **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. Give three reasons why STL algorithms are preferable over hand-written loops.

2. What does the `_if` suffix on an STL algorithm indicate? What about `_n`?

3. What are the five iterator categories?

4. Can an input iterator be used wherever a forward iterator is expected? That is, if an algorithm requires a forward iterator, is it legal to provide it an input iterator instead? What about the other way around?

5. Why do we need `back_insert_iterator` and the like? That is, what would happen with the STL algorithms if these iterator adaptors didn't exist?

6. The `distance` function, defined in the `<iterator>` header, takes in two iterators and returns the number of elements spanned by that iterator range. For example, given a `vector<int>`, calling

   ```
   distance(v.begin(), v.end());
   ```

   returns the number of elements in the container.

   Modify the code from this chapter that prints the average of the values in a file so that it instead prints the average of the values in the file between 25 and 75. If no elements are in this range, you should print a message to this effect. You will need to use a combination of `accumulate` and `distance`.

7. Using `remove_if` and a custom callback function, write a function `RemoveShortWords` that accepts a `vector<string>` and removes all strings of length 3 or less from it. This function can be written in two lines of code if you harness the algorithms correctly.

8. In n-dimensional space, the distance from a point (x_1, x_2, x_3, ..., x_n) to the origin is $\sqrt{x_1^2 + x_2^2 + x_3^2 + \ldots + x_n^2}$. 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 `inner_product` algorithm to compute the expression under the square root.)*

9. 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. This may seem like a silly problem, but can come up in some circumstances. For example, if you have a list of songs in a music library, you might want songs with the title "Untitled" to always appear at the top.

10. 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 string in the `map`. *(Hint: Remember that all elements in a `map<string, double>` are stored internally as `pair<string, double>`)*

11. Implement the `count` algorithm for `vector<int>`s. Your function should have the prototype `int count(vector<int>::iterator start, vector<int>::iterator stop, int element)` and should return the number of elements in the range [`start`, `stop`) that are equal to `element`.

12. Using the `generate_n` algorithm, the `rand` function, and a `back_insert_iterator`, show how to populate a `vector` with a specified number of random values. Then use `accumulate` to compute the average of the range.

13. The *median* of a range of data is the value that is bigger than half the elements in the range and smaller than half the elements in a range. For data sets with odd numbers of elements, this is the middle element when the elements are sorted, and for data sets with an even number of elements it is the average of the two middle elements. Using the `nth_element` algorithm, write a function that computes the median of a set of data.

14. Show how to use a combination of `copy`, `istreambuf_iterator`, and `ostreambuf_iterator` to open a file and print its contents to `cout`.

15. Show how to use a combination of `copy` and iterator adapters to write the contents of an STL container to a file, where each element is stored on its own line.

16. Suppose that you are given two `vector<int>`s with their elements stored in sorted order. Show how to print out the elements those `vector`s have in common in one line of code using the `set_intersection` algorithm and an appropriate iterator adaptor.

17. A *monoalphabetic substitution cipher* is a simple form of encryption. We begin with the letters of the alphabet, as shown here:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

We then scramble these letters randomly, yielding a new ordering of the alphabet. One possibility is as follows:

K	V	D	Q	J	W	A	Y	N	E	F	C	L	R	H	U	X	I	O	G	T	Z	P	M	S	B

This new ordering thus defines a mapping from each letter in the alphabet to some other letter in the alphabet, as shown here:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
K	V	D	Q	J	W	A	Y	N	E	F	C	L	R	H	U	X	I	O	G	T	Z	P	M	S	B

To encrypt a source string, we simply replace each character in the string with its corresponding encrypted character. For example, the string "The cookies are in the fridge" would be encoded as follows:

T	H	E	C	O	O	K	I	E	S	A	R	E	I	N	T	H	E	F	R	I	D	G	E
G	**Y**	**J**	**D**	**H**	**H**	**F**	**N**	**J**	**O**	**K**	**I**	**J**	**N**	**R**	**G**	**Y**	**J**	**W**	**I**	**N**	**Q**	**A**	**J**

Monoalphabetic substitution ciphers are surprisingly easy to break – in fact, most daily newspapers include a daily puzzle that involves deciphering a monoalphabetic substitution cipher – but they are still useful for low-level encryption tasks such as posting spoilers to websites (where viewing the spoiler explicitly requires the reader to decrypt the text).

Using the `random_shuffle` algorithm, implement a function `MonoalphabeticSubstitutionEncrypt` that accepts a source string and encrypts it with a random monoalphabetic substitution cipher.