Chapter 6: STL Associative Containers and Iterators

In the previous chapter, we explored two of the STL's sequence containers, the `vector` and `deque`. These containers are ideally suited for situations where we need to keep track of an ordered list of elements, such as an itinerary, shopping list, or mathematical vector. However, representing data in ordered lists is not optimal in many applications. For example, when keeping track of what merchandise is sold in a particular store, it does not make sense to think of the products as an ordered list. Storing merchandise in a list would imply that the merchandise could be ordered as "this is the first item being sold, this is the second item being sold, etc." Instead, it makes more sense to treat the collection of merchandise as an *unordered collection*, where *membership* rather than *ordering* is the defining characteristic. That is, we are more interested in answers to the question "is item X being sold here?" than answers to the question "where in the sequence is the element X?" Another scenario in which ordered lists are suboptimal arises when trying to represent *relationships* between sets of data. For example, we may want to encode a mapping from street addresses to buildings, or from email addresses to names. In this setup, the main question we are interested in answering is "what value is associated with X?," not "where in the sequence is element X?"

In this chapter, we will explore four new STL container classes – `map`, `set`, `multimap`, and `multiset` – that provide new abstractions for storing data. These containers will represent allow us to ask different questions of our data sets and will make it possible to write programs to solve increasingly complex problems. As we explore those containers, we will introduce *STL iterators*, tools that will pave the way for more advanced STL techniques.

Storing Unordered Collections with `set`

To motivate the STL `set` container, let's consider a simple probability question. Recall from last chapter's *Snake* example that the C++ `rand()` function can be used to generate a pseudorandom integer in the range [0, `RAND_MAX`]. (Recall that the notation [a, b] represents all real numbers between a and b, inclusive). Commonly, we are interested not in values from zero to `RAND_MAX`, but instead values from 0 to some set upper bound k. To get values in this range, we can use the value of

```
    rand() % (k + 1)
```

This computes the remainder when dividing `rand()` by $k + 1$, which must be in the range [0, k].[*]

Now, consider the following question. Suppose that we have a six-sided die. We roll the die, then record what number we rolled. We then keep rolling the die and record what number came up, and keep repeating this process. The question is as follows: how many times, on average, will we roll the die before the same number comes up twice? This is actually a special case of a more general problem: if we continuously generate random integers in the range [0, k], how many numbers should we expect to generate before we generate some number twice? With some fairly advanced probability theory, this value can be calculated exactly. However, this is a textbook on C++ programming, not probability theory, and so we'll write a short program that will simulate this process and report the average number of die rolls.

[*] This process will not always yield uniformly-distributed values, because `RAND_MAX` will not always be a multiple of k. For a fun math exercise, think about why this is.

There are many ways that we can write this program. In the interest of simplicity, we'll break the program into two separate tasks. First, we'll write a function that rolls the die over and over again, then reports how many die rolls occurred before some number came up twice. Second, we'll write our `main` function to call this function multiple times to get a good sample, then will have it print out the average.

Let's think about how we can write a function that rolls a die until the same number comes up twice. At a high level, this function needs to generate a random number from 1 to 6, then check if it has been generated before. If so, it should stop and report the number of dice rolled. Otherwise, it should remember that this number has been rolled, then generate a new number. A key step of this process is remembering what numbers have come up before, and using the techniques we've covered so far we could do this using either a `vector` or a `deque`. For simplicity, we'll use a `vector`. One implementation of this function looks like this:

```
/* Rolls a six-sided die and returns the number that came up. */
int DieRoll() {
    /* rand() % 6 gives back a value between 0 and 5, inclusive.  Adding one to
     * this gives us a valid number for a die roll.
     */
    return (rand() % 6) + 1;
}

/* Rolls the dice until a number appears twice, then reports the number of die
 * rolls.
 */
size_t RunProcess() {
    vector<int> generated;

    while (true) {
        /* Roll the die. */
        int nextValue = DieRoll();

        /* See if this value has come up before.  If so, return the number of
         * rolls required.  This is equal to the number of dice that have been
         * rolled up to this point, plus one for this new roll.
         */
        for (size_t k = 0; k < generated.size(); ++k)
            if (generated[k] == nextValue)
                return generated.size() + 1;

        /* Otherwise, remember this die roll. */
        generated.push_back(nextValue);
    }
}
```

Now that we have the `RunProcess` function written, we can run through one simulation of this process. However, it would be silly to give an estimate based on just one iteration. To get a good estimate, we'll need to run this process multiple times to control for randomness. Consequently, we can write the following `main` function, which runs the process multiple times and reports the average value:

```
const size_t kNumIterations = 10000; // Number of iterations to run

int main() {
    /* Seed the randomizer.  See the last chapter for more information on this
     * line.
     */
    srand(static_cast<unsigned>(time(NULL)));

    size_t total = 0; // Total number of dice rolled

    /* Run the process kNumIterations times, accumulating the result into
     * total.
     */
    for (size_t k = 0; k < kNumIterations; ++k)
        total += RunProcess();

    /* Finally, report the result. */
    cout << "Average number of steps: "
         << double(total) / kNumIterations << endl;
}
```

If you compile and run this program, you'll see output that looks something like this:

```
Average number of steps: 3.7873
```

You might see a different number displayed on your system, since the program involves a fundamentally random process.

Now, let's make a small tweak to this program. Suppose that instead of rolling a six-sided die, we roll a twenty-sided die.* How many steps should we expect this to take now? If we change our implementation of `DieRoll` to the following:

```
int DieRoll() {
    return (rand() % 20) + 1;
}
```

Then running the program will produce output along the following lines:

```
Average number of steps: 6.2806
```

This is interesting – we more than tripled the number of sides on the die (from six to twenty), but the total number of expected rolls increased by less than a factor of two! Is this a coincidence, or is there some fundamental law of probability at work here? To find out, let's assume that we're now rolling a die with 365 sides (i.e. one side for every day of the year). This means our new implementation of `DieRoll` is

```
int DieRoll() {
    return (rand() % 365) + 1;
}
```

Running this program produces output that looks like this:

```
Average number of steps: 24.6795
```

* If you haven't seen a twenty-sided die (or *D20* in gamer-speak), you're really missing out. They're very fun to play with.

Now *that's* weird! In increasing the number of sides on the die from 20 to 365, we increased the number of sides on the die by a factor of (roughly) eighteen. However, the number of expected rolls went up only by a factor of four! But more importantly, think about what this result means. If you have a roomful of people with twenty-five people, then you should expect at least two people in that room to have the same birthday! This is sometimes called the *birthday paradox*, since it seems counterintuitive that such a small sample of people would cause this to occur. The more general result, for those of you who are interested, is that you will need to roll an n sided die roughly $\sqrt{n}$ times before the same number will come up twice.

This has been a fun diversion into the realm of probability theory, but what does it have to do with C++ programming? The answer lies in the implementation of the `RunProcess` function. The heart of this function is a `for` loop that checks whether a particular value is contained inside of a `vector`. This loop is reprinted here for simplicity:

```
for (size_t k = 0; k < generated.size(); ++k)
    if (generated[k] == nextValue)
        return generated.size() + 1;
```

Notice that there is a disparity between the high-level operation being modeled here ("check if the number has already been generated") and the actual implementation ("loop over the `vector`, checking, for each element, whether that element is equal to the most-recently generated number"). There is a tension here between what the code accomplishes and the way in which it accomplishes it. The reason for this is that we're using the *wrong abstraction*. Intuitively, a `vector` maintains an *ordered sequence* of elements. The main operations on a `vector` maintain that sequence by adding and removing elements from that sequence, looking up elements at particular positions in that sequence, etc. For this application, we want to store a collection of numbers that is *unordered*. We don't care when the elements were added to the `vector` or what position they occupy. Instead, we are interested *what* elements are in the `vector`, and in particular whether a given element is in the `vector` at all.

For situations like these, where the *contents* of a collection of elements are more important than the actual *sequence* those elements are in, the STL provides a special container called the `set`. The `set` container represents an arbitrary, unordered collection of elements and has good support for the following operations:

- Adding elements to the collection.
- Removing elements from the collection.
- Determining whether a particular element is in the collection.

To see the `set` in action, let's consider a modified version of the `RunProcess` function which uses a `set` instead of a `vector` to store its elements. This code is shown here (though you'll need to `#include <set>` for it to compile):

```
size_t RunProcess() {
    set<int> generated;

    while (true) {
        int nextValue = DieRoll();

        /* Check if this value has been rolled before. */
        if (generated.count(nextValue)) return generated.size() + 1;

        /* Otherwise, add this value to the set. */
        generated.insert(nextValue);
    }
}
```

Take a look at the changes we made to this code. To determine whether the most-recently-generated number has already been produced, we can use the simple syntax `generated.count(nextValue)` rather than the clunkier `for` loop from before. Also notice that to insert the new element into the `set`, we used the `insert` function rather than `push_back`.

The names of the functions on the `set` are indicative of the differences between the `set` and the `vector` and `deque`. When inserting an element into a `vector` or `deque`, we needed to specify where to put that element: at the end using `push_back`, at the beginning with `push_front`, or at some arbitrary position using `insert`. The `set` has only one function for adding elements – `insert` – which does not require us to specify where in the `set` the element should go. This makes sense, since the `set` is an inherently unordered collection of elements. Additionally, the `set` has no way to query elements at specific positions, since the elements of a `set` don't *have* positions. However, we can check whether an element exists in a `set` very simply using the `count` function, which returns `true` if the element exists and `false` otherwise.*

If you rerun this program using the updated code, you'll find that the program produces almost identical output (the randomness will mean that you're unlikely to get the same output twice). The only difference between the old code and the new code is the internal structure. Using the `set`, the code is easier to read and understand. In the next section, we'll probe the `set` in more detail and explore some of its other uses.

A Primer on `set`

The STL `set` container represents an unordered collection of elements that does not permit duplicates. Logically, a `set` is a collection of unique values that efficiently supports inserting and removing elements, as well as checking whether a particular element is contained in the `set`. Like the `vector` and `deque`, the `set` is a parameterized class. Thus we can speak of a `set<int>`, a `set<double>`, `set<string>`, etc. As with `vector` and `deque`, `set`s can only hold one type of element, so you cannot have a `set` that mixes and matches between `int`s and `string`s, for example. However, unlike the `vector` or `deque`, `set` can only store objects that can be compared using the < operator. This means that you can store all primitive types in a `set`, along with `string`s and other STL containers. However, you cannot store custom `struct`s inside of an STL `set`. For example, the following is illegal:

```
struct Point {
    double x, y;
};

set<Point> mySet; // Illegal, Point cannot be compared with <
```

This may seem like a somewhat arbitrary restriction. Logically, we could be able to gather up anything into an unordered collection. Why does it matter that those elements be comparable using <? The answer has to do with how the `set` is implemented behind the scenes. Internally, the `set` is layered on top of a *balanced binary tree*, a special data structure that naturally supports the `set`'s main operations. However, balanced binary trees can only be constructed on data sets where elements can be compared to one another, hence the restriction. Later in this text we'll see how to use a technique called *operator overloading* to make it possible to store objects of any type in an STL `set`, but for now you will need to confine yourself to primitives and other STL containers.

As we saw in the previous example, one of the most basic `set` operations is insertion using the `insert` function. Unlike the `deque` and `vector insert` functions, you do not need to specify a location for the new element. After all, a `set` represents an unordered collection, and specifying where an element should go in a `set` does not make any sense. Here is some sample code using `insert`:

* Technically speaking, `count` returns 1 if the element exists and 0 otherwise. For most purposes, though, it's safe to treat the function as though it returns a boolean true or false.

```
    set<int> mySet;
    mySet.insert(137);  // Now contains: 137
    mySet.insert(42);   // Now contains: 42 137
    mySet.insert(137);  // Now contains: 42 137
```

Notice in this last line that inserting a second copy of 137 into the `set` did not change the contents of the `set`. `set`s do not allow for duplicate elements.

To check whether a particular element is contained in an STL `set`, you can also use the `count` function, which returns 1 if the element is contained in the set and 0 otherwise. Using C++'s automatic conversion of nonzero values into `true` and zero values to `false`, you usually do not need to explicitly check whether `count` yields a one or zero and can rely on implicit conversions instead. For example:

```
    if(mySet.count(137))
        cout << "137 is in the set." << endl;  // Printed
    if(!mySet.count(500))
        cout << "500 is not in the set." << endl; // Printed
```

To remove an element from a `set`, you use the `erase` function. `erase` is a mirror to `insert`, and the two have very similar syntax. For example:

```
      mySet.erase(137); // Removes 137, if it exists.
```

The STL `set` also supports several operations common to all STL containers. You can remove all elements from a `set` using `clear`, check how many elements are present using `size`, etc. A full table of all `set` operations is presented later in this chapter.

Traversing Containers with Iterators

One of the most common operations we've seen in the course of working with the STL containers is *iteration*, traversing the contents of a container and performing some task on every element. For example, the following loop iterates over the contents of a `vector`, printing each element:

```
    for (size_t h = 0; h < myVector.size(); ++h)
        cout << myVector[h] << endl;
```

We can similarly iterate over a `deque` as follows:

```
    for (size_t h = 0; h < myDeque.size(); ++h)
        cout << myDeque[h] << endl;
```

The reason that we can use this convenient syntax to traverse the contents of the `vector` and `deque` is because the `vector` and `deque` represent linear sequences, and so it is possible to enumerate all possible indices in the container using the standard `for` loop. That is, we can iterate so easily over a `vector` or `deque` because we can look up the zeroth element, then the first element, then the second, etc. Unfortunately, this logic does not work on the STL `set`. Because the `set` does not have an ordering on its elements, it does not make sense to speak of the “zeroth element of a set,” nor the “first element of a set,” etc. To traverse the elements of a `set`, we will need to use a new concept, the *iterator*.

Every STL container presents a different means of storing data. `vector` and `deque` store data in an ordered list. `set` stores its data as an unordered collection. As you'll soon see, `map` encodes data as a collection of key/value pairs. But while each container stores its data in a different format, fundamentally, each container still stores data. Iterators provide a clean, consistent mechanism for accessing data stored in containers, irrespective of how that data may be stored. That is, the syntax for looking at `vector` data with iterators is almost identical to the syntax for examining `set` and `deque` data with iterators. This fact is extremely important. For starters, it implies that once you've learned how to use iterators to traverse *any* container, you can use them to traverse *all* containers. Also, as you'll see, because iterators can traverse data stored in any container, they can be used to specify a collection of values in a way that masks how those values are stored behind-the-scenes.

So what exactly is an iterator? At a high level, an iterator is like a cursor in a text editor. Like a cursor, an iterator has a well-defined position inside a container, and can move from one character to the next. Also like a cursor, an iterator can be used to read or write a range of data one element at a time.

It's difficult to get a good feel for how iterators work without having a sense of how all the pieces fit together. Therefore, we'll get our first taste of iterators by jumping head-first into the idiomatic "loop over the elements of a container" `for` loop, then will clarify all of the pieces individually. Here is a sample piece of code that will traverse the elements of a `vector<int>`, printing each element out on its own line:

```
vector<int> myVector = /* ... some initialization ... */
for (vector<int>::iterator itr = myVector.begin();
     itr != myVector.end(); ++itr)
    cout << *itr << endl;
```

This code is perhaps the densest C++ we've encountered yet, so let's take a few minutes to dissect exactly what's going on here. The first part of the `for` loop is the statement

```
vector<int>::iterator itr = myVector.begin();
```

This line of code creates an object of type `vector<int>::iterator`, an iterator variable named `itr` that can traverse a `vector<int>`. Note that a `vector<int>::iterator` can only iterate over a `vector<int>`. If we wanted to iterate over a `vector<string>`, we would need to use a `vector<string>::iterator`, and if we wanted to traverse a `set<int>` we would have to use a `set<int>::iterator`. We then initialize the iterator to `myVector.begin()`. Every STL container class exports a member function `begin()` which yields an iterator pointing to the first element of that container. By initializing the iterator to `myVector.begin()`, we indicate to the C++ compiler that the `itr` iterator will be traversing elements of the container `myVector`.

Inside the body of the `for` loop, we have the line

```
cout << *itr << endl;
```

The strange-looking entity `*itr` is known as an *iterator dereference* and means "the element being iterated over by `itr`." As `itr` traverses the elements of the `vector`, it will proceed from one element to the next in sequence until all of the elements of the `vector` have been visited. At each step, the element being iterated over can be yielded by prepending a star to the name of the iterator. In the above context, we dereference the iterator to yield the current element of `myVector` being traversed, then print it out. We will discuss the nuances of iterator dereferences in more detail shortly.

Returning up to the `for` loop itself, notice that after each iteration we execute

```
++itr;
```

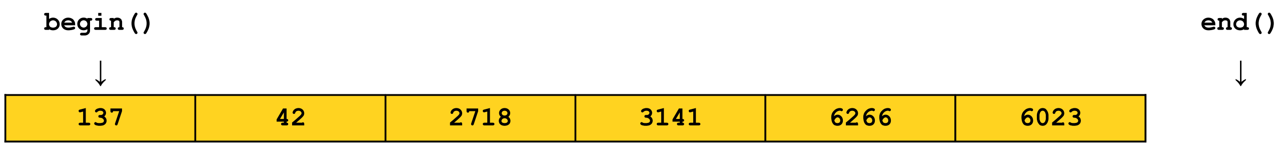
When applied to `int`s, the `++` operator is the increment operator; writing `++myInt` means "increment the value of the `myInt` variable." When applied to iterators, the `++` operator means "advance the iterator one step forward." Because the step condition of the `for` loop is `++itr`, this means that each iteration of the `for` loop will advance the iterator to the next element in the container, and eventually all elements will be visited. Of course, at some point, we will have visited all of the elements in the `vector` and will need to stop iterating. To detect when an iterator has visited all of the elements, we loop on the condition that

```
    itr != myVector.end();
```

Each STL container exports a special function called `end()` that returns an iterator to the element *one past the end of the container*. For example, consider the following `vector`:

137	**42**	**2718**	**3141**	**6266**	**6023**

In this case, the iterators returned by that `vector`'s `begin()` and `end()` functions would point to the following locations:



Notice that the `begin()` iterator points to the first element of the `vector`, while the `end()` iterator points to the slot one position past the end of the `vector`. This may seem strange at first, but is actually an excellent design decision. Recall the `for` loop from above, which iterates over the elements of a `vector`. This is reprinted below:

```
    for (vector<int>::iterator itr = myVector.begin();
         itr != myVector.end(); ++itr)
        cout << *itr << endl;
```

Compare this to the more traditional loop you're used to, which also iterates over a `vector`:

```
    for (size_t h = 0; h < myVector.size(); ++h)
        cout << myVector[h] << endl;
```

Because the `vector` is zero-indexed, if you were to look up the element in the `vector` at position `myVector.size()`, you would be reading a value not actually contained in the `vector`. For example, in a `vector` of five elements, the elements are stored at positions 0, 1, 2, 3, and 4. There is no element at position five, and trying to read an element there will result in undefined behavior. However, in the `for` loop to iterate over the contents of the `vector`, we still use the value of `myVector.size()` as the upper bound for the iteration, since the loop will cut off as soon as the iteration index reaches the value `myVector.size()`. This is identical to the behavior of the `end()` iterator in the iterator-based `for` loop. `myVector.end()` is never a valid iterator, but we use it as the loop upper bound because as soon as the `itr` iterator reaches `myVector.end()` the loop will terminate.

Part of the beauty of iterators is that the above `for` loop for iterating over the contents of a `vector` can trivially be adapted to iterate over just about any STL container class. For instance, if we want to iterate over the contents of a `deque<int>`, we could do so as follows:

```
    deque<int> myDeque = /* ... some initialization ... */
    for (deque<int>::iterator itr = myDeque.begin(); itr != myDeque.end(); ++itr)
        cout << *itr << endl;
```

This is *exactly* the same loop structure, though some of the types have changed (i.e. we've replaced `vector<int>::iterator` with `deque<int>::iterator`). However, the behavior is identical. This loop will traverse the contents of the `deque` in sequence, printing each element out as it goes.

Of course, at this point iterators may seem like a mere curiosity. Sure, we can use them to iterate over a `vector` or `deque`, but we already could do that using a more standard `for` loop. The beauty of iterators is that they work on any STL container, including the `set`. If we have a `set` of elements we wish to traverse, we can do so using the following syntax:

```
set<int> mySet = /* ... some initialization ... */
for (set<int>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << endl;
```

Again, notice that the structure of the loop is the same as before. Only the types have changed.

One crucial detail we've ignored up to this point is in what order the elements of a `set` will be traversed. When using the `vector` or `deque` there is a natural iteration order (from the start of the sequence to the end), but when using the STL `set` the idea of ordering is a bit more vague. However, iteration order over a `set` is well-specified. When traversing `set` elements via an iterator, the elements will be visited in sorted order, starting with the smallest element and ending with the largest. This is in part why the STL `set` can only store elements comparable using the less-than operator: there is no well-defined “smallest” or “biggest” element of a `set` if the elements cannot be compared. To see this in action, consider the following code snippet:

```
/* Generate ten random numbers */
set<int> randomNumbers;
for (size_t k = 0; k < 10; ++k)
    randomNumbers.insert(rand());

/* Print them in sorted order. */
for (set<int>::iterator itr = randomNumbers.begin();
     itr != randomNumbers.end(); ++itr)
    cout << *itr << endl;
```

This will print different outputs on each run, since the program generates and stores random numbers. However, the values will always be in sorted order. For example:

```
137 2718 3141 4103 5422 6321 8938 10299 12003 16554
```

Spotlight on Iterators

As you just saw, there are three major operations on iterators:

- *Dereferencing* the iterator to read a value.
- *Advancing* the iterator from one position to the next.
- *Comparing* two iterators for equality.

Iterator dereferencing is a particularly important operation, and so before moving on we'll take a few minutes to explore this in more detail.

As you've seen so far, iterators can be used to read the values of a container indirectly. However, iterators can also be used to *write* the values of a container indirectly as well. For example, here is a simple `for` loop to set all of the elements of a `vector<int>` to 137:

```
for (vector<int>::iterator itr = myVector.begin();
     itr != myVector.end(); ++itr)
    *itr = 137;
```

This is your first glimpse of the true power of iterators. Because iterators give a means for reading and writing container elements indirectly, it is possible to write functions that operate on data from any container class by manipulating iterators from that container class. These functions are called *STL algorithms* and will be discussed in more detail next chapter.

Up to this point, when working with iterators, we have restricted ourselves to STL containers that hold primitive types. That is, we've talked about `vector<int>` and `set<int>`, but not, say, `vector<string>`. All of the syntax that we have seen so far for containers holding primitive types are applicable to containers holding objects. For example, this loop will correctly print out all of the `string`s in a `set<string>`:

```
for (set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << endl;
```

However, let's suppose that we want to iterate over a `set<string>` printing out the *lengths* of the `string`s in that `set`. Unfortunately, the following syntax will *not* work:

```
for (set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr.length() << endl; // Error: Incorrect syntax!
```

The problem with this code is that the C++ compiler interprets it as

```
*(itr.length())
```

Instead of

```
(*itr).length()
```

That is, the compiler tries to call the nonexistent `length()` function on the iterator and to dereference *that*, rather than dereferencing the iterator and then invoking the `length()` function on the resulting value. This is a subtle yet important difference, so make sure that you take some time to think it through before moving on.

To fix this problem, all STL iterators support and operator called the *arrow operator* that allows you to invoke member functions on the element currently being iterated over. For example, to print out the lengths of all of the `string`s in a `set<string>`, the proper syntax is

```
for (set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << itr->length() << endl;
```

We will certainly encounter the arrow operator more as we continue our treatment of the material, so make sure that you understand its usage before moving on.

Defining Ranges with Iterators

Recall for a moment the standard "loop over a container" `for` loop:

```
set<int> mySet = /* ... some initialization ... */
for (set<int>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << endl;
```

If you'll notice, this loop is bounded by two iterators – mySet.begin(), which specifies the first element to iterate over, and mySet.end(), which defines the element one past the end of the iteration range. This raises an interesting point about the *duality* of iterators. A *single* iterator points to a single position in a container class and represents a way to read or write that value indirectly. A *pair* of iterators defines two positions and consequently defines a *range* of elements. In particular, given two iterators start and stop, these iterators define the range of elements beginning with start and ending one position before stop. Using mathematical notation, the range of elements defined by start and stop spans [start, stop).

So far, the only ranges we've considered have been those of the form [begin(), end()) consisting of all of the elements of a container. However, as we begin moving on to progressively more complicated programs, we will frequently work on ranges that do not span all of a container. For example, we might be interested in iterating over only the first half of a container, or perhaps just a slice of elements in a container meeting some property.

If you'll recall, the STL set stores its elements in sorted order, a property that guarantees efficient lookup and insertion. Serendipitously, this allows us to efficiently iterate over a slice out of a set whose values are bounded between some known limits. The set exports two functions, lower_bound and upper_bound, that can be used to iterate over the elements in a set that are within a certain range. lower_bound accepts a value, then returns an iterator to the first element in the set greater than or equal to that value. upper_bound similarly accepts a value and returns an iterator to the first element in the set that is strictly greater than the specified element. Given a closed range [lower, upper], we can iterate over that range by using lower_bound to get an iterator to the first element no less than lower and iterating until we reach the value returned by upper_bound, the first element strictly greater than upper. For example, the following loop iterates over all elements in the set in the range [10, 100]:

```
set<int>::iterator stop = mySet.upper_bound(100);
for(set<int>::iterator itr = mySet.lower_bound(10); itr != stop; ++itr)
     /* ... perform tasks... */
```

Part of the beauty of upper_bound and lower_bound is that it doesn't matter whether the elements specified as arguments to the functions actually exist in the set. For example, suppose that we run the above for loop on a set containing all the odd numbers between 3 and 137. In this case, neither 10 nor 100 are contained in the set. However, the code will still work correctly. The lower_bound function returns an iterator to the first element *at least as large* as its argument, and in the set of odd numbers would return an iterator to the element 11. Similarly, upper_bound returns an iterator to the first element *strictly greater* than its argument, and so would return an iterator to the element 101.

Summary of set

The following table lists some of the most important set functions. Again, we haven't covered const yet, so for now it's safe to ignore it. We also haven't covered const_iterators, but for now you can just treat them as iterators that can't write any values.

Constructor: set<T>()	set<int> mySet; Constructs an empty set.
Constructor: set<T>(const set<T>& other)	set<int> myOtherSet = mySet; Constructs a set that's a copy of another set.

Constructor: `set<T>(InputIterator start,` `InputIterator stop)`	`set<int> mySet(myVec.begin(), myVec.end());` Constructs a `set` containing copies of the elements in the range [`start`, `stop`). Any duplicates are discarded, and the elements are sorted. Note that this function accepts iterators from any source.
`size_type size() const`	`int numEntries = mySet.size();` Returns the number of elements contained in the `set`.
`bool empty() const`	`if(mySet.empty()) { ... }` Returns whether the `set` is empty.
`void clear()`	`mySet.clear();` Removes all elements from the `set`.

`iterator begin()`
`const_iterator begin() const`
	`set<int>::iterator itr = mySet.begin();`
	
	Returns an iterator to the start of the `set`. Be careful when modifying elements in-place.

`iterator end()`
`const_iterator end()`
	`while(itr != mySet.end()) { ... }`
	
	Returns an iterator to the element one past the end of the final element of the `set`.

`pair<iterator, bool>`
`insert(const T& value)`
`void insert(InputIterator begin,`
`InputIterator end)`
	`mySet.insert(4);`
	`mySet.insert(myVec.begin(), myVec.end());`
	
	The first version inserts the specified value into the `set`. The return type is a `pair` containing an iterator to the element and a `bool` indicating whether the element was inserted successfully (`true`) or if it already existed (`false`). The second version inserts the specified range of elements into the `set`, ignoring duplicates.

`iterator find(const T& element)`
`const_iterator`
`find(const T& element) const`
	`if(mySet.find(0) != mySet.end()) { ... }`
	
	Returns an iterator to the specified element if it exists, and `end` otherwise.

`size_type count(const T& item) const`	`if(mySet.count(0)) { ... }` Returns 1 if the specified element is contained in the `set`, and 0 otherwise.

`size_type erase(const T& element)`
`void erase(iterator itr);`
`void erase(iterator start,`
`iterator stop);`
	`if(mySet.erase(0)) {...} // 0 was erased`
	`mySet.erase(mySet.begin());`
	`mySet.erase(mySet.begin(), mySet.end());`
	
	Removes an element from the `set`. In the first version, the specified element is removed if found, and the function returns 1 if the element was removed and 0 if it wasn't in the `set`. The second version removes the element pointed to by `itr`. The final version erases elements in the range [`start`, `stop`).

`iterator lower_bound(const T& value)`	`itr = mySet.lower_bound(5);` Returns an iterator to the first element that is greater than or equal to the specified value. This function is useful for obtaining iterators to a range of elements, especially in conjunction with `upper_bound`.
`iterator upper_bound(const T& value)`	`itr = mySet.upper_bound(100);` Returns an iterator to the first element that is greater than the specified value. Because this element must be strictly greater than the specified value, you can iterate over a range until the iterator is equal to `upper_bound` to obtain all elements less than or equal to the parameter.

A Useful Helper: `pair`

We have just finished our treatment of the `set` and are about to move on to one of the STL's most useful containers, the `map`. However, before we can cover the `map` in any detail, we must first make a quick diversion to a useful helper class, the `pair`.

`pair` is a parameterized class that simply holds two values of arbitrary type. `pair`, defined in `<utility>`, accepts two template arguments and is declared as

```
    pair<TypeOne, TypeTwo>
```

`pair` has two fields, named `first` and `second`, which store the values of the two elements of the pair; `first` is a variable of type `TypeOne`, `second` of type `TypeTwo`. For example, to make a `pair` that can hold an `int` and a `string`, we could write

```
    pair<int, string> myPair;
```

We could then access the `pair`'s contents as follows

```
    pair<int, string> myPair;
    myPair.first  =  137;
    myPair.second = "C++ is awesome!";
```

In some instances, you will need to create a `pair` on-the-fly to pass as a parameter (especially to the `map`'s `insert`). You can therefore use the `make_pair` function as follows:

```
    pair<int, string> myPair = make_pair(137, "string!");
```

Interestingly, even though we didn't specify what type of `pair` to create, the `make_pair` function was able to deduce the type of the `pair` from the types of the elements. This has to do with how C++ handles function templates and we'll explore this in more detail later.

Representing Relationships with `map`

One of the most important data structures in modern computer programming is the *map*, a way of tagging information with some other piece of data. The inherent idea of a *mapping* should not come as a surprise to you. Almost any entity in the real world has extra information associated with it. For example, days of the year have associated events, items in your refrigerator have associated expiration dates, and people you know have associated titles and nicknames. The `map` STL container manages a relationship between a set of *keys* and a set of *values*. For example, the keys in the `map` might be email addresses, and the values the names of the people who own those email addresses. Alternatively, the keys might be longitude/latitude pairs, and the values the name of the city that resides at those coordinates.

Data in a `map` is stored in key/value pairs. Like the `set`, these elements are unordered. Also like the `set`, it is possible to query the map for whether a particular *key* exists in the `map` (note that the check is "does key X exist?" rather than "does *key/value pair* X exist?"). Unlike the `set`, however, the `map` also allows clients to ask "what is the value associated with key X?" For example, in a `map` from longitude/latitude pairs to city names, it is possible to give a properly-constructed pair of coordinates to the map, then get back which city is at the indicated location (if such a city exists).

The `map` is unusual as an STL container because unlike the `vector`, `deque`, and `set`, the `map` is parameterized over two types, the type of the key and the type of the value. For example, to create a `map` from `string`s to `int`s, you would use the syntax

```
map<string, int> myMap;
```

Like the STL `set`, behind the scenes the `map` is implemented using a balanced binary tree. This means that the *keys* in the `map` must be comparable using the less-than operator. Consequently, you won't be able to use your own custom `struct`s as keys in an STL `map`. However, the *values* in the map needn't be comparable, so it's perfectly fine to map from `string`s to custom `struct` types. Again, when we cover operator overloading later in this text, you will see how to store arbitrary types as keys in an STL `map`.

The `map` supports many different operations, of which four are key:

- Inserting a new key/value pair.
- Checking whether a particular key exists.
- Querying which value is associated with a given key.
- Removing an existing key/value pair.

We will address each of these in turn.

In order for a `map` to be useful, we will need to populate it with a collection of key/value pairs. There are two ways to insert key/value pairs into the map. The simplest way to insert key/value pairs into a `map` is to user the element selection operator (square brackets) to implicitly add the pair, as shown here:

```
map<string, int> numberMap;
numberMap["zero"] = 0;
numberMap["one"] = 1;
numberMap["two"] = 2;
/* ... etc. ... */
```

This code creates a new `map` from `string`s to `int`s. It then inserts the key `"zero"` which maps to the number zero, the key `"one"` which maps to the number one, etc. Notice that this is a major way in which the `map` differs from the `vector`. Indexing into a `vector` into a nonexistent position will cause undefined behavior, likely a full program crash. Indexing into a `map` into a nonexistent key implicitly creates a key/value pair.

The square brackets can be used both to insert new elements into the `map` and to query the `map` for the values associated with a particular key. For example, assuming that `numberMap` has been populated as above, consider the following code snippet:

```
cout << numberMap["zero"] << endl;
cout << numberMap["two"] * numberMap["two"] << endl;
```

The output of this program is

```
0
4
```

On the first line, we query the `numberMap` map for the value associated with the key `"zero"`, which is the number zero. The second line looks up the value associated with the key `"two"` and multiplies it with itself. Since `"two"` maps to the number two, the output is four.

Because the square brackets both query and create key/value pairs, you should use care when looking values up with square brackets. For example, given the above number map, consider this code:

```
cout << numberMap["xyzzy"] << endl;
```

Because `"xyzzy"` is not a key in the `map`, this implicitly creates a key/value pair with `"xyzzy"` as the key and zero as the value. (Like the `vector` and `deque`, the `map` will zero-initialize any primitive types used as values). Consequently, this code will output

```
0
```

and will change the `numberMap` map so that it now has `"xyzzy"` as a key. If you want to look up a key/value pair without accidentally adding a new key/value pair to the `map`, you can use the `map`'s `find` member function. `find` takes in a key, then returns an `iterator` that points to the key/value pair that has the specified key. If the key does not exist, `find` returns the `map`'s `end()` iterator as a sentinel. For example:

```
map<string, int>::iterator itr = numberMap.find("xyzzy");
if (itr == numberMap.end())
    cout << "Key does not exist." << endl;
else
    /* ... */
```

When working with an STL `vector`, `deque`, or `set`, iterators simply iterated over the contents of the container. That is, a `vector<int>::iterator` can be dereferenced to yield an `int`, while a `set<string>::iterator` dereferences to a `string`. `map` iterators are slightly more complicated because they dereference to a key/value pair. In particular, if you have a `map<KeyType, ValueType>`, then the iterator will dereference to a value of type

```
pair<const KeyType, ValueType>
```

This is a `pair` of an immutable key and a mutable value. We have not talked about the `const` keyword yet, but it means that keys in a `map` cannot be changed after they are set (though they can be removed). The values associated with a key, on the other hand, can be modified.

Because `map` iterators dereference to a `pair`, you can access the keys and values from an iterator as follows:

```
map<string, int>::iterator itr = numberMap.find("xyzzy");
if (itr == numberMap.end())
    cout << "Key does not exist." << endl;
else
    cout << "Key " << itr->first << " has value " << itr->second << endl;
```

That is, to access the key from a `map` iterator, you use the arrow operator to select the `first` field of the `pair`. The value is stored in the `second` field. This naturally segues into the stereotypical "iterate over the elements of a `map` loop," which looks like this:

```
for (map<string, int>::iterator itr = myMap.begin(); itr != myMap.end(); ++itr)
    cout << itr->first << ": " << itr->second << endl;
```

When iterating over a `map`, the key/value pairs will be produced sorted by key from lowest to highest. This means that if we were to iterate over the `numberMap` map from above printing out key/value pairs, the output would be

```
one: 1
two: 2
zero: 0
```

Since the keys are `string`s which are sorted in alphabetical order.

You've now seen how to insert, query, and iterate over key/value pairs. Removing key/value pairs from a `map` is also fairly straightforward. To do so, you use the `erase` function as follows:

```
myMap.erase("key");
```

That is, the erase function accepts a *key*, then removes the key/value pair from the `map` that has that key (if it exists).

As with all STL containers, you can remove all key/value pairs from a `map` using the `clear` function, determine the number of key/value pairs using the `size` function, etc. There are a few additional operations on a `map` beyond these basic operations, some of which are covered in the next section.

`insert` and How to Avoid It

As seen above, you can use the square brackets operator to insert and update key/value pairs in the `map`. However, there is another mechanism for inserting key/value pairs: `insert`. Like the `set`'s `insert` function, you need only specify what to insert, since the `map`, like the `set`, does not store values in a particular order. However, because the `map` stores elements as key/value pairs, the parameter to the `insert` function should be a `pair` object containing the key and the value. For example, the following code is an alternative means of populating the `numberMap` map:

```
map<string, int> numberMap;
numberMap.insert(make_pair("zero", 0));
numberMap.insert(make_pair("one", 1));
numberMap.insert(make_pair("two", 2));
/* ... */
```

There is one key difference between the `insert` function and the square brackets. Consider the following two code snippets:

```
/* Populate a map using [ ] */
map<string, string> one;
one["C++"] = "sad";
one["C++"] = "happy";

/* Populate a map using insert */
map<string, string> two;
two.insert(make_pair("C++", "sad"));
two.insert(make_pair("C++", "happy"));
```

In the first code snippet, we create a `map` from `string`s to `string`s called `one`. We first create a key/value pair mapping `"C++"` to `"sad"`, and then overwrite the value associated with `"C++"` to `"happy"`. After this code executes, the `map` will map the key "C++" to the value "happy", since in the second line the value was overwritten. In the second code snippet, we call `insert` twice, once inserting the key `"C++"` with the value `"sad"` and once inserting the key `"C++"` with the value `"happy"`. When this code executes, the `map` will end up holding one key/value pair: `"C++"` mapping to `"sad"`. Why is this the case?

Like the STL `set`, the `map` stores a unique set of keys. While multiple keys may map to the same value, there can only be one key/value pair for any given key. When inserting and updating keys with the square brackets, any updates made to the `map` are persistent; writing code to the effect of `myMap[key] = value` ensures that the map contains the key `key` mapping to value `value`. However, the `insert` function is not as forgiving. If you try to insert a key/value pair into a `map` using the `insert` function and the `key` already exists, the `map` will not insert the key/value pair, nor will it update the value associated with the existing key. To mitigate this, the `map`'s `insert` function returns a value of type `pair<iterator, bool>`. The `bool` value in the `pair` indicates whether the `insert` operation succeeded; a result of `true` means that the key/value pair was added, while a result of `false` means that the key already existed. The `iterator` returned by the `insert` function points to the key/value `pair` in the `map`. If the key/value pair was newly-added, this iterator points to the newly-inserted value, and if a key/value pair already exists the iterator points to the existing key/value pair that prevented the operation from succeeding. If you want to use `insert` to insert key/value pairs, you can write code to the following effect:

```
/* Try to insert normally. */
pair<map<string, int>::iterator, bool> result =
    myMap.insert(make_pair("STL", 137));

/* If insertion failed, manually set the value. */
if(!result.second)
    result.first->second = 137;
```

In the last line, the expression `result.first->second` is the value of the existing entry, since `result.first` yields an iterator pointing to the entry, so `result.first->second` is the value field of the iterator to the entry. As you can see, the `pair` can make for tricky, unintuitive code.

If `insert` is so inconvenient, why even bother with it? Usually, you won't, and will use the square brackets operator instead. However, when working on an existing codebase, you are extremely likely to run into the `insert` function, and being aware of its somewhat counterintuitive semantics will save you many hours of frustrating debugging.

`map` Summary

The following table summarizes the most important functions on the STL `map` container. Feel free to ignore `const` and `const_iterator`s; we haven't covered them yet.

Constructor: `map<K, V>()`	`map<int, string> myMap;` Constructs an empty `map`.
Constructor: `map<K, V>(const map<K, V>& other)`	`map<int, string> myOtherMap = myMap;` Constructs a `map` that's a copy of another `map`.

Constructor: `map<K, V>(InputIterator start,`
`InputIterator stop)`
	`map<string, int> myMap(myVec.begin(),`
	`myVec.end());`
	
	Constructs a `map` containing copies of the elements in the range [`start`, `stop`). Any duplicates are discarded, and the elements are sorted. Note that this function accepts iterators from any source, but they must be iterators over pairs of keys and values.

`size_type size() const`	`int numEntries = myMap.size();` Returns the number of elements contained in the `map`.
`bool empty() const`	`if(myMap.empty()) { ... }` Returns whether the `map` is empty.
`void clear()`	`myMap.clear();` Removes all elements from the `map`.

`iterator begin()`
`const_iterator begin() const`
	`map<int>::iterator itr = myMap.begin();`
	
	Returns an iterator to the start of the `map`. Remember that iterators iterate over pairs of keys and values.

`iterator end()`
`const_iterator end()`
	`while(itr != myMap.end()) { ... }`
	
	Returns an iterator to the element one past the end of the final element of the `map`.

`pair<iterator, bool>`
`insert(const pair<const K, V>& value)`
`void insert(InputIterator begin,`
`InputIterator end)`
	`myMap.insert(make_pair("STL", 137));`
	`myMap.insert(myVec.begin(), myVec.end());`
	
	The first version inserts the specified key/value pair into the `map`. The return type is a `pair` containing an iterator to the element and a `bool` indicating whether the element was inserted successfully (`true`) or if it already existed (`false`). The second version inserts the specified range of elements into the `map`, ignoring duplicates.

`V& operator[] (const K& key)`	`myMap["STL"] = "is awesome";` Returns the value associated with the specified key, if it exists. If not, a new key/value pair will be created and the value initialized to zero (if it is a primitive type) or the default value (for non-primitive types).

`iterator find(const K& element)`
`const_iterator find(const K& element) const`
	`if(myMap.find(0) != myMap.end()) { ... }`
	
	Returns an iterator to the key/value pair having the specified key if it exists, and `end` otherwise.

`size_type count(const K& item) const`	`if(myMap.count(0)) { ... }` Returns 1 if some key/value pair in the `map` has specified key and 0 otherwise.

`size_type erase(const K& element)`
`void erase(iterator itr);`
`void erase(iterator start, iterator stop);`
	`if(myMap .erase(0)) {...}`
	`myMap.erase(myMap.begin());`
	`myMap.erase(myMap.begin(), myMap.end());`
	
	Removes a key/value pair from the `map`. In the first version, the key/value pair having the specified key is removed if found, and the function returns 1 if a pair was removed and 0 otherwise. The second version removes the element pointed to by `itr`. The final version erases elements in the range [`start`, `stop`).

`iterator lower_bound(const K& value)`	`itr = myMap.lower_bound(5);` Returns an iterator to the first key/value pair whose key is greater than or equal to the specified value. This function is useful for obtaining iterators to a range of elements, especially in conjunction with `upper_bound`.
`iterator upper_bound(const K& value)`	`itr = myMap.upper_bound(100);` Returns an iterator to the first key/value pair whose key is greater than the specified value. Because this element must be strictly greater than the specified value, you can iterate over a range until the iterator is equal to `upper_bound` to obtain all elements less than or equal to the parameter.

Extended Example: Keyword Counter

To give you a better sense for how `map` and `set` can be used in practice, let's build a simple application that brings them together: a *keyword counter*. C++, like most programming languages, has a set of *reserved words*, keywords that have a specific meaning to the compiler. For example, the keywords for the primitive types `int` and `double` are reserved words, as are the `switch`, `for`, `while`, `do`, and `if` keywords used for control flow. For your edification, here's a complete list of the reserved words in C++:

and	continue	goto	public	try
and_eq	default	if	register	typedef
asm	delete	inline	reinterpret_cast	typeid
auto	do	int	return	typename
bitand	double	long	short	union
bitor	dynamic_cast	mutable	signed	unsigned
bool	else	namespace	sizeof	using
break	enum	new	static	virtual
case	explicit	not	static_cast	void
catch	export	not_eq	struct	volatile
char	extern	operator	switch	wchar_t
class	false	or	template	while
compl	float	or_eq	this	xor
const	for	private	throw	xor_eq
const_cast	friend	protected	true	

We are interested in answering the following question: given a C++ source file, how many times does each reserved word come up? This by itself might not be particularly enlightening, but in some cases it's interesting to see how often (or infrequently) the keywords come up in production code.

We will suppose that we are given a file called `keywords.txt` containing all of C++'s reserved words. This file is structured such that every line of the file contains one of C++'s reserved words. Here's the first few lines:

***File:* `keywords.txt`**

```
and
and_eq
asm
auto
bitand
bitor
bool
break
...
```

Given this file, let's write a program that prompts the user for a filename, loads the file, then reports the frequency of each keyword in that file. For readability, we'll only print out a report on the keywords that actually occurred in the file. To avoid a major parsing headache, we'll count keywords wherever they appear, even if they're in comments or contained inside of a string.

Let's begin writing this program. We'll use a top-down approach, breaking the task up into smaller subtasks which we will implement later on. Here is one possible implementation of the `main` function:

```
#include <iostream>
#include <string>
#include <fstream>
#include <map>
using namespace std;

/* Function: OpenUserFile(ifstream& fileStream);
 * Usage: OpenUserFile(myStream);
 * -------------------------------------------------
 * Prompts the user for a filename until a valid filename
 * is entered, then sets fileStream to read from that file.
 */
void OpenUserFile(ifstream& fileStream);

/* Function: GetFileContents(ifstream& file);
 * Usage: string contents = GetFileContents(ifstream& file);
 * -------------------------------------------------
 * Returns a string containing the contents of the file passed
 * in as a parameter.
 */
string GetFileContents(ifstream& file);

/* Function: GenerateKeywordReport(string text);
 * Usage: map<string, size_t> keywords = GenerateKeywordReport(contents);
 * -------------------------------------------------
 * Returns a map from keywords to the frequency at which those keywords
 * appear in the input text string.  Keywords not contained in the text will
 * not appear in the map.
 */
map<string, size_t> GenerateKeywordReport(string contents);
```

```
    int main() {
        /* Prompt the user for a valid file and open it as a stream. */
        ifstream input;
        OpenUserFile(input);

        /* Generate the report based on the contents of the file. */
        map<string, size_t> report = GenerateKeywordReport(GetFileContents(input));

        /* Print a summary. */
        for (map<string, size_t>::iterator itr = report.begin();
             itr != report.end(); ++itr)
            cout << "Keyword " << itr->first << " occurred "
                 << itr->second << " times." << endl;
}
```

The breakdown of this program is as follows. First, we prompt the user for a file using the OpenUserFile function. We then obtain the file contents as a string and pass it into GenerateKeywordReport, which builds us a map from strings of the keywords to size_ts representing the frequencies. Finally, we print out the contents of the map in a human-readable format. Of course, we haven't implemented any of the major functions that this program will use, so this program won't link, but at least it gives a sense of where the program is going.

Let's begin implementing this code by writing the OpenUserFile function. Fortunately, we've already written this function last chapter in the *Snake* example. The code for this function is reprinted below:

```
    void OpenUserFile(ifstream& input) {
        while(true) {
            cout << "Enter filename: ";
            string filename = GetLine();

            input.open(filename.c_str()); // See Chapter 3 for .c_str().
            if(input.is_open()) return;

            cout << "Sorry, I can't find the file " << filename << endl;
            input.clear();
        }
    }
```

Here, the GetLine() function is from the chapter on streams, and is implemented as

```
    string GetLine() {
        string line;
        getline(input, line);
        return line;
    }
```

Let's move on to the next task, reading the file contents into a string. This can be done in a few lines of code using the streams library. The idea is simple: we'll maintain a string containing all of the file contents encountered so far, and continuously concatenate on the next line of the file (which we'll read with the streams library's handy getline function). This is shown here:

```
string GetFileContents(ifstream& input) {
    /* String which will hold the file contents. */
    string result;

    /* Keep reading a line of the file until no data remains. *
    string line;
    while (getline(input, line))
        result += line + "\n"; // Add the newline character; getline removes it

    return result;
}
```

All that remains at this point is the `GenerateKeywordReport` function, which ends up being most of the work. The basic idea behind this function is as follows:

- Load in the list of keywords.
- For each word in the file:
    - If it's a keyword, increment the keyword count appropriately.
    - Otherwise, ignore it.

We'll take this one step at a time. First, let's load in the list of keywords. But how should we store those keywords? We'll be iterating over words from the user's file, checking at each step whether the given word is a keyword. This means that we will want to store the keywords in a way where we can easily query whether a string is or is not contained in the list of keywords. This is an ideal spot for a `set`, which is optimized for these operations. We can therefore write a function that looks like this to read the reserved words list into a `set`:

```
set<string> LoadKeywords() {
    ifstream input("keywords.txt"); // No error checking for brevity's sake
    set<string> result;

    /* Keep reading strings out of the file until we cannot read any more.
     * After reading each string, store it in the result set.  We can either
     * use getline or the stream extraction operator here, but the stream
     * extraction operator is a bit more general.
     */
    string keyword;
    while (input >> keyword)
        result.insert(keyword);

    return result;
}
```

We now have a way to read in the set of keywords, and can move on to our next task: reading all of the words out of the file and checking whether any of them are reserved words. This is surprisingly tricky. We are given a string, a continuous sequence of characters, and from this string want to identify where each "word" is. How are we to do this? There are many options at our disposal (we'll see a heavy-duty way to do this at the end of the chapter), but one particularly elegant method is to harness a `stringstream`. If you'll recall, the `stringstream` class is a stream object that can build and parse strings using standard stream operations. Further recall that when reading string data out of a stream using the stream extraction operator, the read operation will proceed up until it encounters whitespace or the end of the stream. That is, if we had a stream containing the text

```
This, dear reader, is a string.
```

If we were to read data from the stream one string at a time, we would get back the strings

```
This,
dear
reader,
is
a
string.
```

In that order. As you can see, the input is broken apart at whitespace boundaries, rather than word boundaries. However, whenever we encounter a word that does not have punctuation immediately on either side, the string is parsed correctly. This suggests a particularly clever trick. We will modify the full text of the file by replacing all punctuation characters with whitespace characters. Having performed this manipulation, if we parse the file contents using a `stringstream`, each string handed back to us will be a complete word.

Let's write a function, `PreprocessString`, which accepts as input a `string` by reference, then replaces all punctuation characters in that string with the space character. To help us out, we have the `<cctype>` header, which exports the `ispunct` function. `ispunct` takes in a single character, then returns whether or not it is a punctuation character. Unfortunately, `ispunct` treats underscores as punctuation, which will cause problems for some reserved words (for example, `static_cast`), and so we'll need to special-case it. The `PreprocessString` function is as follows:

```
void PreprocessString(string& text) {
    for (size_t k = 0; k < text.size(); ++k)
        if (ispunct(text[k]) && text[k] != '_') // If we need to change it...
            text[k] = ' '; // ... replace it with a space.
}
```

Combining this function with `LoadKeywords` gives us this partial implementation of `GenerateKeywordReport`:

```
map<string, size_t> GenerateKeywordReport(string fileContents) {
    /* Load the set of keywords from disk. */
    set<string> keywords = LoadKeywords();

    /* Preprocess the string to allow for easier parsing. */
    PreprocessString(fileContents);

    /* ... need to fill this in ... */
}
```

All that's left to do now is tokenize the string into individual words, then build up a frequency map of each keyword. To do this, we'll funnel the preprocessed file contents into a `stringstream` and use the prototypical stream reading loop to break it apart into individual words. This can be done as follows:

```
map<string, size_t> GenerateKeywordReport(string fileContents) {
    /* Load the set of keywords from disk. */
    set<string> keywords = LoadKeywords();

    /* Preprocess the string to allow for easier parsing. */
    PreprocessString(fileContents);

    /* Populate a stringstream with the file contents. */
    stringstream tokenizer;
    tokenizer << fileContents;

    /* Loop over the words in the file, building up the report. */
    map<string, size_t> result;

    string word;
    while (tokenizer >> word)
        /* ... process word here ... */
}
```

Now that we have a loop for extracting single words from the input, we simply need to check whether each word is a reserved word and, if so, to make a note of it. This is done here:

```
map<string, size_t> GenerateKeywordReport(string fileContents) {
    /* Load the set of keywords from disk. */
    set<string> keywords = LoadKeywords();

    /* Preprocess the string to allow for easier parsing. */
    PreprocessString(fileContents);

    /* Populate a stringstream with the file contents. */
    stringstream tokenizer;
    tokenizer << fileContents;

    /* Loop over the words in the file, building up the report. */
    map<string, size_t> result;

    string word;
    while (tokenizer >> word)
        if (keywords.count(word))
            ++ result[word];

    return result;
}
```

Let's take a closer look at what this code is doing. First, we check whether the current word is a keyword by using the `set`'s `count` function. If so, we increment the count of that keyword in the file by writing `++result[word]`. This is a surprisingly compact line of code. If the keyword has not been counted before, then `++result[word]` will implicitly create a new key/value pair using that keyword as the key and initializing the associated value to zero. The `++` operator then kicks in, incrementing the value by one. Otherwise, if the key already existed in the `map`, the line of code will retrieve the value, then increment it by one. Either way, the count is updated appropriately, and the `map` will be populated correctly.

We now have a working implementation of the `GenerateKeywordReport` function, and, combined with the rest of the code we've written, we now have a working implementation of the keyword counting program. As an amusing test, the result of running this program on itself is as follows:

```
Keyword for occurred 3 times.
Keyword if occurred 3 times.
Keyword int occurred 1 times.
Keyword namespace occurred 1 times.
Keyword return occurred 6 times.
Keyword true occurred 1 times.
Keyword using occurred 1 times.
Keyword void occurred 2 times.
Keyword while occurred 4 times.
```

How does this compare to production code? For reference, here is the output of the program when run on the monster source file `nsCSSFrameConstructor.cpp`, an 11,000+ line file from the Mozilla Firefox source code:[*]

```
Keyword and occurred 268 times.
Keyword auto occurred 2 times.
Keyword break occurred 58 times.
Keyword case occurred 66 times.
Keyword catch occurred 2 times.
Keyword char occurred 4 times.
Keyword class occurred 10 times.
Keyword const occurred 149 times.
Keyword continue occurred 11 times.
Keyword default occurred 8 times.
Keyword delete occurred 6 times.
Keyword do occurred 99 times.
Keyword else occurred 135 times.
Keyword enum occurred 1 times.
Keyword explicit occurred 4 times.
Keyword extern occurred 4 times.
Keyword false occurred 12 times.
Keyword float occurred 15 times.
Keyword for occurred 292 times.
Keyword friend occurred 3 times.
Keyword if occurred 983 times.
Keyword inline occurred 86 times.
Keyword long occurred 5 times.
Keyword namespace occurred 5 times.
Keyword new occurred 59 times.
Keyword not occurred 145 times.
Keyword operator occurred 1 times.
Keyword or occurred 108 times.
Keyword private occurred 2 times.
Keyword protected occurred 1 times.
Keyword public occurred 5 times.
Keyword return occurred 452 times.
Keyword sizeof occurred 3 times.
Keyword static occurred 118 times.
Keyword static_cast occurred 20 times.
Keyword struct occurred 8 times.
Keyword switch occurred 4 times.
Keyword this occurred 205 times.
Keyword true occurred 14 times.
Keyword try occurred 10 times.
Keyword using occurred 6 times.
Keyword virtual occurred 1 times.
Keyword void occurred 82 times.
Keyword while occurred 53 times.
```

As you can see, we have quite a lot of C++ ground to cover – just look at all those keywords we haven't covered yet!

[*] As of April 12, 2010

Multicontainers

The STL provides two special "multicontainer" classes, `multimap` and `multiset`, that act as `map`s and `set`s except that the values and keys they store are not necessarily unique. That is, a `multiset` could contain several copies of the same value, while a `multimap` might have duplicate keys associated with different values.

`multimap` and `multiset` (declared in `<map>` and `<set>`, respectively) have identical syntax to `map` and `set`, except that some of the functions work slightly differently. For example, the `count` function will return the number of copies of an element an a multicontainer, not just a binary zero or one. Also, while `find` will still return an iterator to an element if it exists, the element it points to is not guaranteed to be the only copy of that element in the multicontainer. Finally, the `erase` function will erase *all* copies of the specified key or element, not just the first it encounters.

One important distinction between the `multimap` and regular `map` is the lack of square brackets. On a standard STL `map`, you can use the syntax `myMap[key] = value` to add or update a key/value pair. However, this operation only makes sense because keys in a `map` are unique. When writing `myMap[key]`, there is only one possible key/value pair that could be meant. However, in a `multimap` this is not the case, because there may be multiple key/value pairs with the same key. Consequently, to insert key/value pairs into a `multimap`, you will need to use the `insert` function. Fortunately, the semantics of the `multimap` `insert` function are much simpler than the `map`'s `insert` function, since insertions never fail in a `multimap`. If you try to insert a key/value pair into a `multimap` for which the key already exists in the `multimap`, the new key/value pair will be inserted without any fuss. After all, `multimap` exists to allow single keys to map to multiple values!

One function that's quite useful for the multicontainers is `equal_range`. `equal_range` returns a `pair<iterator, iterator>` that represents the span of entries equal to the specified value. For example, given a `multimap<string, int>`, you could use the following code to iterate over all entries with key "STL":

```
/* Store the result of the equal_range */
pair<multimap<string, int>::iterator, multimap<string, int>::iterator>
   myPair = myMultiMap.equal_range("STL");

/* Iterate over it! */
for(multimap<string, int>::iterator itr = myPair.first;
    itr != myPair.second; ++itr)
   cout << itr->first << ": " << itr->second << endl;
```

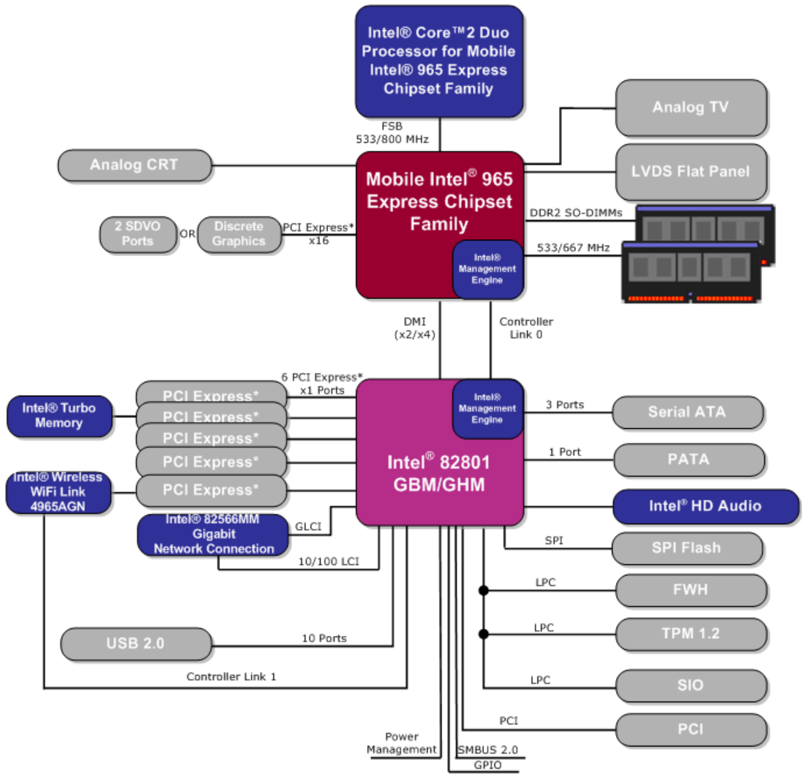
The multicontainers are fairly uncommon in practice partially because they can easily be emulated using the regular `map` or `set`. For example, a `multimap<string, int>` behaves similarly to a `map<string, vector<int> >` since both act as a map from `string`s to some number of `int`s. However, in many cases the multicontainers are exactly the tool for the job; we'll see them used later in this chapter.

Extended Example: Finite Automata

Computer science is often equated with programming and software engineering. Many a computer science student has to deal with the occasional "Oh, you're a computer science major! Can you make me a website?" or "Computer science, eh? Why isn't my Internet working?" This is hardly the case and computer science is a much broader discipline that encompasses many fields, such as artificial intelligence, graphics, and biocomputation.

One particular subdiscipline of computer science is *computability theory*. Since computer science involves so much programming, a good question is exactly *what* we can command a computer to do. What sorts of problems can we solve? How efficiently? What problems *can't* we solve and why not? Many of the most important results in computer science, such as the undecidability of the halting problem, arise from computability theory.

But how exactly can we determine what can be computed with a computer? Modern computers are phenomenally complex machines. For example, here is a high-level model of the chipset for a mobile Intel processor: [Intel]



Modeling each of these components is exceptionally tricky, and trying to devise any sort of proof about the capabilities of such a machine would be all but impossible. Instead, one approach is to work with *automata*, abstract mathematical models of computing machines (the singular of *automata* is the plural of *automaton*). Some types of automata are realizable in the physical world (for example, deterministic and nondeterministic finite automata, as you'll see below), while others are not. For example, the *Turing machine*, which computer scientists use as an overapproximation of modern computers, requires infinite storage space, as does the weaker *pushdown automaton*.

Although much of automata theory is purely theoretical, many automata have direct applications to software engineering. For example, most production compilers simulate two particular types of automata (called *pushdown automata* and *nondeterministic finite automata*) to analyze and convert source code into a form readable by the compiler's semantic analyzer and code generator. Regular expression matchers, which search through text strings in search of patterned input, are also frequently implemented using an automaton called a *deterministic finite automaton*.

In this extended example, we will introduce two types of automata, *deterministic finite automata* and *nondeterministic finite automata*, then explore how to represent them in C++. We'll also explore how these automata can be used to simplify difficult string-matching problems.

Deterministic Finite Automata

Perhaps the simplest form of an automaton is a *deterministic finite automaton*, or DFA. At a high-level, a DFA is similar to a flowchart – it has a collection of *states* joined by various *transitions* that represent how the DFA should react to a given input. For example, consider the following DFA:

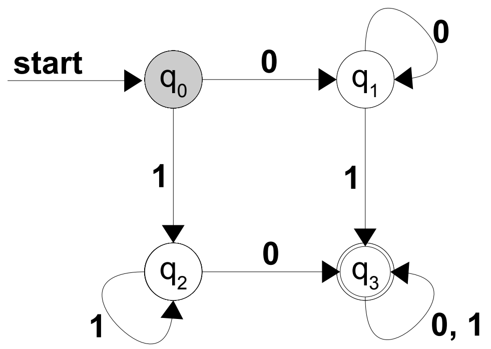
This DFA has four states, labeled q_0, q_1, q_2, and q_3, and a set of labeled transitions between those states. For example, the state q_0 has a transition labeled **0** to q_1 and a transition labeled **1** to q_2. Some states have transitions to themselves; for example, q_2 transitions to itself on a **1**, while q_3 transitions to itself on either a **0** or **1**. Note that as shorthand, the transition labeled **0, 1** indicates two different transitions, one labeled with a **0** and one labeled with a **1**. The DFA has a designated state state, in this case q_0, which is indicated by the arrow labeled **start**.

Notice that the state q_3 has two rings around it. This indicates that q_3 is an *accepting state*, which will have significance in a moment when we discuss how the DFA processes input.

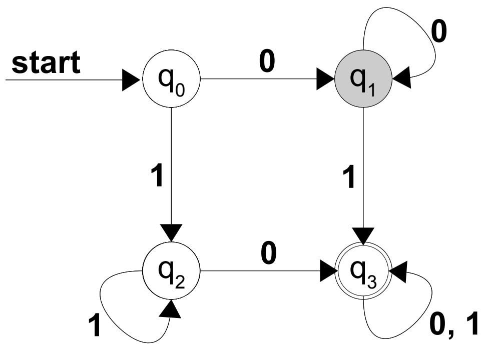
Since all of the transitions in this DFA are labeled either **0** or **1**, this DFA is said to have an *alphabet* of {**0**, **1**}. A DFA can use any nonempty set of symbols as an alphabet; for example, the Latin or Greek alphabets are perfectly acceptable for use as alphabets in a DFA, as is the set of integers between 42 and 137. By definition, every state in a DFA is required to have a transition for each symbol in its alphabet. For example, in the above DFA, each state has exactly two transitions, one labeled with a **0** and the other with a **1**. Notice that state q_3 has only one transition explicitly drawn, but because the transition is labeled with two symbols we treat it as two different transitions.

The DFA is a simple computing machine that accepts as input a string of characters formed from its alphabet, processes the string, and then halts by either *accepting* the string or *rejecting* it. In essence, the DFA is a device for discriminating between two types of input – input for which some criterion is true and input for which it is false. The DFA starts in its designated start state, then processes its input character-by-character by transitioning from its current state to the state indicated by the transition. Once the DFA has finished consuming its input, it accepts the string if it ends in an accepting state; otherwise it rejects the input.

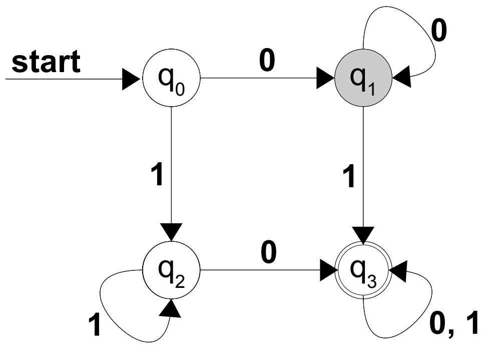
To see exactly how a DFA processes input, let us consider the above DFA simulated on the input **0011**. Initially, we begin in the start state, as shown here:



Since the first character of our string is a **0**, we follow the transition to state q_1, as shown here:

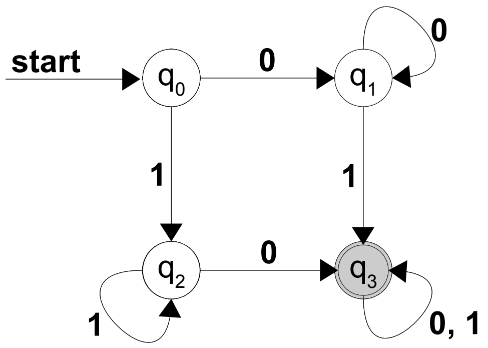


The second character of input is also a **0**, so we follow the transition labeled with a **0** and end up back in state q_1, leaving us in this state:



Next, we consume the next input character, a **1**, which causes us to follow the transition labeled **1** to state q_3:

The final character of input is also a **1**, so we follow the transition labeled **1** and end back up in q_3:



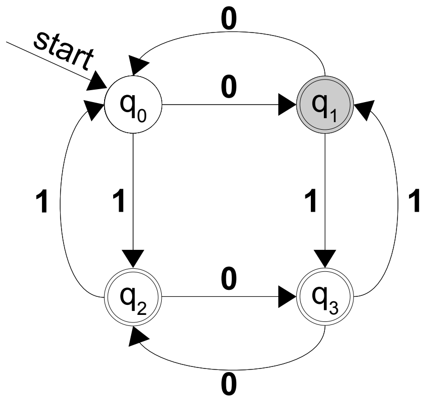
We are now done with our input, and since we have ended in an accepting state, the DFA accepts this input.

We can similarly consider the action of this DFA on the string **111**. Initially the machine will start in state q_0, then transition to state q_2 on the first input. The next two inputs each cause the DFA to transition back to state q_2, so once the input is exhausted the DFA ends in state q_2, so the DFA rejects the input. We will not prove it here, but this DFA accepts all strings that have at least one **0** and at least one **1**.

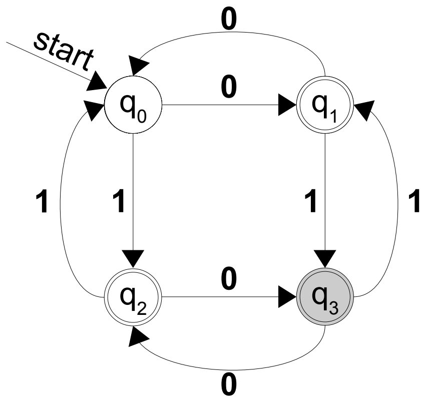
Two important details regarding DFAs deserve some mention. First, it is possible for a DFA to have multiple accepting states, as is the case in this DFA:

As with the previous DFA, this DFA has four states, but notice that three of them are marked as accepting. This leads into the second important detail regarding DFAs – the DFA only accepts its input if the DFA ends in an accepting state *when it runs out of input*. Simply transitioning into an accepting state does not cause the DFA to accept. For example, consider the effect of running this DFA on the input **0101**. We begin in the start state, as shown here:

We first consume a **0**, sending us to state q_1:

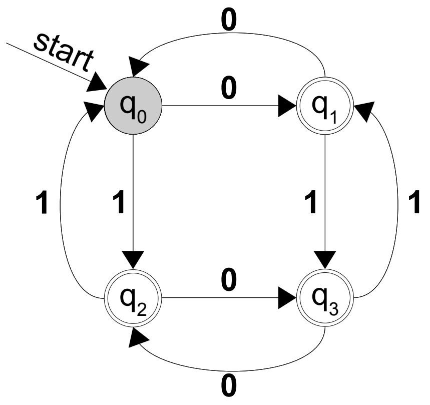


Next, we read a **1**, sending us to state q_3:



The next input is a **0**, sending us to q_2:

Finally, we read in a **1**, sending us back to q_0:



Since we are out of input and are not in an accepting state, this DFA rejects its input, even though we transitioned through every single accepting state. If you want a fun challenge, convince yourself that this DFA accepts all strings that contain an odd number of **0**s or an odd number of **1**s (inclusive OR).

Representing a DFA

A DFA is a simple model of computation that can easily be implemented in software or hardware. For any DFA, we need to store five pieces of information:[*]

1. The set of states used by the DFA.
2. The DFA's alphabet.
3. The start state.
4. The state transitions.
5. The set of accepting states.

Of these five, the one that deserves the most attention is the fourth, the set of state transitions. Visually, we have displayed these transitions as arrows between circles in the graph. However, another way to treat state transitions is as a table with states along one axis and symbols of the alphabet along the other. For example, here is a transition table for the DFA described above:

State	**0**	**1**
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

To determine the state to transition to given a current state and an input symbol, we look up the row for the current state, then look at the state specified in the column for the current input.

If we want to implement a program that simulates a DFA, we can represent almost all of the necessary information simply by storing the transition table. The two axes encode all of the states and alphabet symbols, and the entries of the table represent the transitions. The information not stored in this table is the set of accepting states and the designated start state, so provided that we bundle this information with the table we have a full description of a DFA.

To concretely model a DFA using the STL, we must think of an optimal way to model the transition table. Since transitions are associated with pairs of states and symbols, one option would be to model the table as an STL `map` mapping a state-symbol pair to a new state. If we represent each symbol as a `char` and each state as an `int` (i.e. q_0 is 0, q_1 is 1, etc.), this leads to a state transition table stored as a `map<pair<int, char>, int>`. If we also track the set of accepting states as a `set<int>`, we can encode a DFA as follows:

```
struct DFA {
    map<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

For the purposes of this example, assume that we have a function which fills this `DFA` struct will relevant data. Now, let's think about how we might go about simulating the DFA. To do this, we'll write a function `SimulateDFA` which accepts as input a `DFA struct` and a string representing the input, simulates the DFA when run on the given input, and then returns whether the input was accepted. We'll begin with the following:

```
bool SimulateDFA(DFA& d, string input) {
```

* In formal literature, a DFA is often characterized as a quintuple (Q, Σ, q_0, δ, F) of the states, alphabet, start state, transition table, and set of accepting states, respectively. Take CS154 if you're interested in learning more about these wonderful automata, or CS143 if you're interested in their applications.

```
        /* ... */
    }
```

We need to maintain the state we're currently in, which we can do by storing it in an `int`. We'll initialize the current state to the starting state, as shown here:

```
    bool SimulateDFA(DFA& d, string input) {
        int currState = d.startState;
        /* ... */
    }
```

Now, we need to iterate over the string, following transitions from state to state. Since the transition table is represented as a `map` from `pair<int, char>`s, we can look up the next state by using `make_pair` to construct a pair of the current state and the next input, then looking up its corresponding value in the `map`. As a simplifying assumption, we'll assume that the input string is composed only of characters from the DFA's alphabet.

This leads to the following code:

```
    bool SimulateDFA(DFA& d, string input) {
        int currState = d.startState;
        for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
            currState = d.transitions[make_pair(currState, *itr)];
        /* ... */
    }
```

You may be wondering how we're iterating over the contents of a `string` using iterators. Surprisingly, the `string` is specially designed like the STL container classes, and so it's possible to use all of the iterator tricks you've learned on the STL containers directly on the `string`.

Once we've consumed all the input, we need to check whether we ended in an accepting state. We can do this by looking up whether the `currState` variable is contained in the `acceptingStates set` in the `DFA` struct, as shown here:

```
    bool SimulateDFA(DFA& d, string input) {
        int currState = d.startState;
        for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
            currState = d.transitions[make_pair(currState, *itr)];
        return d.acceptingStates.find(currState) != d.acceptingStates.end();
    }
```

This function is remarkably simple but correctly simulates the DFA run on some input. As you'll see in the next section on applications of DFAs, the simplicity of this implementation lets us harness DFAs to solve a suite of problems surprisingly efficiently.

Applications of DFAs

The C++ `string` class exports a handful of searching functions (`find`, `find_first_of`, `find_last_not_of`, etc.) that are useful for locating specific strings or characters. However, it's surprisingly tricky to search strings for specific patterns of characters. The canonical example is searching for email addresses in a string of text. All email addresses have the same structure – a name field followed by an at sign (`@`) and a domain name. For example, htiek@cs.stanford.edu and this.is.not.my.real.address@example.com are valid email addresses. In general, we can specify the formatting of an email address as follows:*

- The name field, which consists of nonempty alphanumeric strings separated by periods. Periods can only occur between alphanumeric strings, never before or after. Thus hello.world@example.com and cpp.is.really.cool@example.com are legal but .oops@example.com, oops.@example.com, and oops..oops@example.com are not.

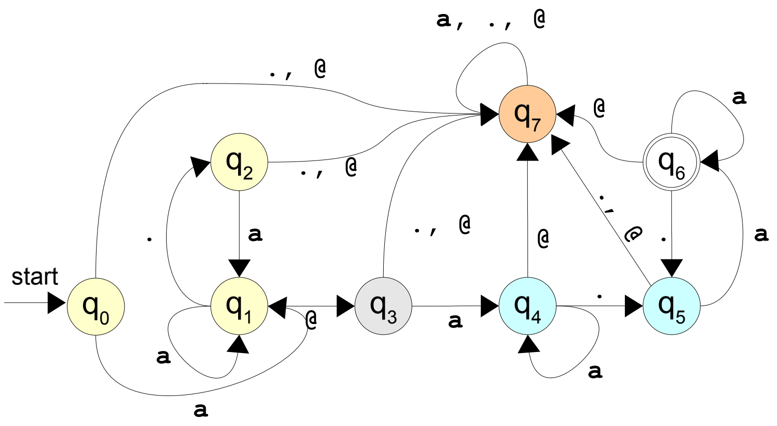
- The host field, which is structured similarly to the above except that there must be at least two sequences separated by a dot.

Now, suppose that we want to determine whether a string is a valid email address. Using the searching functions exported by the `string` class this would be difficult, but the problem is easily solved using a DFA. In particular, we can design a DFA over a suitable alphabet that accepts a string if and only if the string has the above formatting.

The first question to consider is what alphabet this DFA should be over. While we could potentially have the DFA operate over the entire ASCII alphabet, it's easier if we instead group together related characters and use a simplified alphabet. For example, since email addresses don't distinguish between letters and numbers, we can have a single symbol in our alphabet that encodes any alphanumeric character. We would need to maintain the period and at-sign in the alphabet since they have semantic significance. Thus our alphabet will be {**a**, `.`, `@`}, where **a** represents alphanumeric characters, `.` is the period character, and `@` is an at-sign.

Given this alphabet, we can represent all email addresses using the following DFA:

* This is a simplified version of the formatting of email addresses. For a full specification, refer to RFCs 5321 and 5322.



This DFA is considerably trickier than the ones we've encountered previously, so let's take some time to go over what's happening here. The machine starts in state q_0, which represents the beginning of input. Since all email addresses have to have a nonempty name field, this state represents the beginning of the first string in the name. The first character of an email address must be an alphanumeric character, which if read in state q_0 cause us to transition to state q_1. States q_1 and q_2 check that the start of the input is something appropriately formed from alphanumeric characters separated by periods. Reading an alphanumeric character while in state q_1 keeps the machine there (this represents a continuation of the current word), and reading a dot transitions the machine to q_2. In q_2, reading anything other than an alphanumeric character puts the machine into the "trap state," state q_7, which represents that the input is invalid. Note that once the machine reaches state q_7 no input can get the machine out of that state and that q_7 isn't accepting. Thus any input that gets the machine into state q_7 will be rejected.

State q_3 represents the state of having read the at-sign in the email address. Here reading anything other than an alphanumeric character causes the machine to enter the trap state.

States q_4 and q_5 are designed to help catch the name of the destination server. Like q_1, q_4 represents a state where we're reading a "word" of alphanumeric characters and q_5 is the state transitioned to on a dot. Finally, state q_6 represents the state where we've read at least one word followed by a dot, which is the accepting state. As an exercise, trace the action of this machine on the inputs valid.address@email.com and invalid@not.com@ouch.

Now, how can we use this DFA in code? Suppose that we have some way to populate a `DFA` struct with the information for this DFA. Then we could check if a string contains an email address by converting each character in the string into its appropriate character in the DFA alphabet, then simulating the DFA on the input. If the DFA rejects the input or the string contains an invalid character, we can signal that the string is invalid, but otherwise the string is a valid email address.

This can be implemented as follows:

```
bool IsEmailAddress(string input) {
    DFA emailChecker = LoadEmailDFA(); // Implemented elsewhere

    /* Transform the string one character at a time. */
    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        /* isalnum is exported by <cctype> and checks if the input is an
         * alphanumeric character.
         */
        if(isalnum(*itr))
            *itr = 'a';
        /* If we don't have alphanumeric data, we have to be a dot or at-sign
         * or the input is invalid.
         */
        else if(*itr != '.' && *itr != '@')
            return false;
    }
    return SimulateDFA(emailChecker, input);
}
```

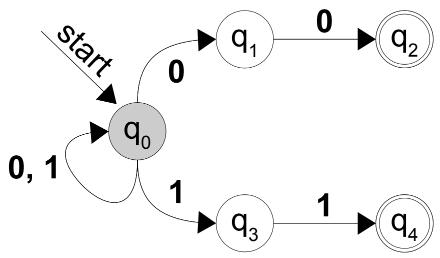
This code is remarkably concise, and provided that we have an implementation of `LoadEmailDFA` the function will work correctly. I've left out the implementation of `LoadEmailDFA` since it's somewhat tedious, but if you're determined to see that this actually works feel free to try your hand at implementing it.

Nondeterministic Finite Automata

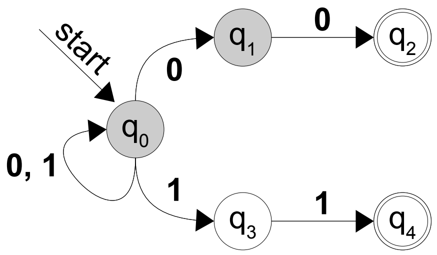
A generalization of the DFA is the *nondeterministic finite automaton*, or NFA. At a high level, DFAs and NFAs are quite similar – they both consist of a set of states connected by labeled transitions, of which some states are designated as accepting and others as rejecting. However, NFAs differ from DFAs in that a state in an NFA can have any number of transitions on a given input, including zero. For example, consider the following NFA:

Here, the start state is q_0 and accepting states are q_2 and q_4. Notice that the start state q_0 has two transitions on **0** – one to q_1 and one to itself – and two transitions on **1**. Also, note that q_3 has no defined transitions on **0**, and states q_2 and q_4 have no transitions at all.

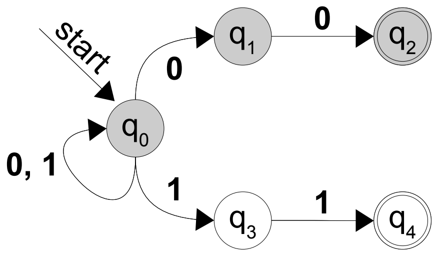
There are several ways to interpret a state having multiple transitions. The first is to view the automaton as choosing one of the paths nondeterministically (hence the name), then accepting the input if *some* set of choices results in the automaton ending in an accepting state. Another, more intuitive way for modeling multiple transitions is to view the NFA as being in several different states simultaneously, at each step following every transition with the appropriate label in each of its current states. To see this, let's consider what happens when we run the above NFA on the input **0011**. As with a DFA, we begin in the start state, as shown here:



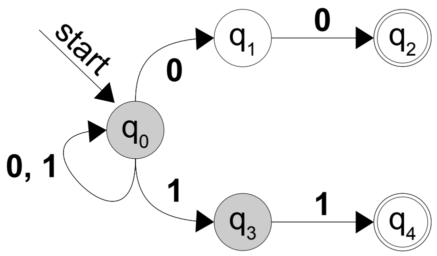
We now process the first character of input (**0**) and find that there are two transitions to follow – the first to q_0 and the second to q_1. The NFA thus ends up in both of these states simultaneously, as shown here:



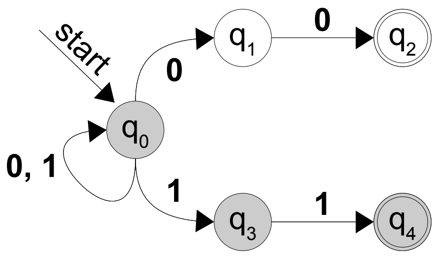
Next, we process the second character (**0**). From state q_0, we transition into q_0 and q_1, and from state q_1 we transition into q_2. We thus end up in states q_0, q_1, and q_2, as shown here:



We now process the third character of input, which is a **1**. From state q_0 we transition to states q_0 and q_3. We are also currently in states q_1 and q_2, but neither of these states has a transition on a **1**. When this happens, we simply drop the states from the set of current states. Consequently, we end up in states q_0 and q_3, leaving us in the following configuration:



Finally, we process the last character, a **1**. State q_0 transitions to q_0 and q_1, and state q_1 transitions to state q_4. We thus end up in this final configuration:



Since the NFA ends in a configuration where at least one of the active states is an accepting state (q_4), the NFA accepts this input. Again as an exercise, you might want to convince yourself that this NFA accepts all and only the strings that end in either **00** or **11**.

Implementing an NFA

Recall from above the definition of the `DFA` struct:

```
struct DFA {
    map<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

Here, the transition table was encoded as a `map<pair<int, char>, int>` since for every combination of a state and an alphabet symbol there was exactly one transition. To generalize this to represent an NFA, we need to be able to associate an arbitrary number of possible transitions. This is an ideal spot for an STL `multimap`, which allows for duplicate key/value pairs. This leaves us with the following definition for an NFA type:

```
struct NFA {
    multimap<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

How would we go about simulating this NFA? At any given time, we need to track the set of states that we are currently in, and on each input need to transition from the current set of states to some other set of states. A natural representation of the current set of states is (hopefully unsurprisingly) as a `set<int>`. Initially, we start with this set of states just containing the start state. This is shown here:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    /* ... */
}
```

Next, we need to iterate over the string we've received as input, following transitions where appropriate. This at least requires a simple `for` loop, which we'll write here:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        /* ... */
    }

    /* ... */
}
```

Now, for each character of input in the string, we need to compute the set of next states (if any) to which we should transition. To simplify the implementation of this function, we'll create a second `set<int>` corresponding to the next set of states the machine will be in. This eliminates problems caused by adding elements to our set of states as we're iterating over the set and updating it. We thus have

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        set<int> nextStates;
        /* ... */
    }
```

```
        /* ... */
    }
```

Now that we have space to put the next set of machine states, we need to figure out what states to transition to. Since we may be in multiple different states, we need to iterate over the set of current states, computing which states they transition into. This is shown here:

```
    bool SimulateNFA(NFA& nfa, string input) {
        /* Track our set of states.  We begin in the start state. */
        set<int> currStates;
        currStates.insert(nfa.startState);

        for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
            set<int> nextStates;
            for(set<int>::iterator state = currStates.begin();
                state != currStates.end(); ++state) {
                /* ... */
            }
        }

        /* ... */
    }
```

Given the state being iterated over by `state` and the current input character, we now want to transition to each state indicated by the `multimap` stored in the `NFA` struct. If you'll recall, the STL `multimap` exports a function called `equal_range` which returns a `pair` of iterators into the `multimap` that delineate the range of elements with the specified key. This function is exactly what we need to determine the set of new states we'll be entering for each given state – we simply query the `multimap` for all elements whose key is the pair of the specified state and the current input, then add all of the destination states to our next set of states. This is shown here:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state) {
            /* Get all states that we transition to from this current state. */
            pair<multimap<pair<int, char>, int>::iterator,
                 multimap<pair<int, char>, int>::iterator>
            transitions = nfa.transitions.equal_range(make_pair(*state, *itr));

            /* Add these new states to the nextStates set. */
            for(; transitions.first != transitions.second; ++transitions.first)
                /* transitions.first is the current iterator, and its second
                 * field is the value (new state) in the STL multimap.
                 */
                nextStates.insert(transitions.first->second);
        }
    }

    /* ... */
}
```

Finally, once we've consumed all input, we need to check whether the set of states contains any states that are also in the set of accepting states. We can do this by simply iterating over the set of current states, then checking if any of them are in the accepting set. This is shown here and completes the implementation of the function:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state) {
            /* Get all states that we transition to from this current state. */
            pair<multimap<pair<int, char>, int>::iterator,
                 multimap<pair<int, char>, int>::iterator>
            transitions = nfa.transitions.equal_range(make_pair(*state, *itr));

            /* Add these new states to the nextStates set. */
            for(; transitions.first != transitions.second; ++transitions.first)
                /* transitions.first is the current iterator, and its second
                 * field is the value (new state) in the STL multimap.
                 */
                nextStates.insert(transitions.first->second);
        }
    }

    for(set<int>::iterator itr = currStates.begin();
        itr != currStates.end(); ++itr)
        if(nfa.acceptingStates.count(*itr)) return true;
    return false;
}
```

Compare this function to the implementation of the DFA simulation. There is substantially more code here, since we have to track multiple different states rather than just a single state. However, this extra complexity is counterbalanced by the simplicity of designing NFAs compared to DFAs. Building a DFA to match a given pattern can be much trickier than building an equivalent NFA because it's difficult to model "guessing" behavior with a DFA. However, both functions are a useful addition to your programming arsenal, so it's good to see how they're implemented.

More to Explore

In this chapter we covered `map` and `set`, which combined with `vector` and `deque` are the most commonly-used STL containers. However, there are several others we didn't cover, a few of which might be worth looking into. Here are some topics you might want to read up on:

1. **`list`**: `vector` and `deque` are sequential containers that mimic built-in arrays. The `list` container, however, models a sequence of elements without indices. `list` supports several nifty operations, such as merging, sorting, and splicing, and has quick insertions at almost any point. If you're planning on using a linked list for an operation, the `list` container is perfect for you.

2. **The Boost Containers**: The Boost C++ Libraries are a collection of functions and classes developed to augment C++'s native library support. Boost offers several new container classes that might be worth looking into. For example, `multi_array` is a container class that acts as a `Grid` in any number of dimensions. Also, the `unordered_set` and `unordered_map` act as replacements to the `set` and `map` that use hashing instead of tree structures to store their data. If you're interested in exploring these containers, head on over to www.boost.org.

Practice Problems

1. How do you check whether an element is contained in an STL `set`?

2. What is the restriction on what types can be stored in an STL `set`? Do the `vector` or `deque` have this restriction?

3. How do you insert an element into a `set`? How do you remove an element from a `set`?

4. How many copies of a single element can exist in a `set`? How about a `multiset`?

5. How do you iterate over the contents of a `set`?

6. How do you check whether a key is contained in an STL `map`?

7. List two ways that you can insert key/value pairs into an STL `map`.

8. What happens if you look up the value associated with a nonexistent key in an STL `map` using square brackets? What if you use the `find` function?

9. Recall that when iterating over the contents of an STL `multiset`, the elements will be visited in sorted order. Using this property, rewrite the program from last chapter that reads a list of numbers from the user, then prints them in sorted order. Why is it necessary to use a `multiset` instead of a regular `set`?

10. The *union* of two sets is the collection of elements contained in at least one of the `set`s. For example, the union of {1, 2, 3, 5, 8} and {2, 3, 5, 7, 11} is {1, 2, 3, 5, 7, 8, 11}. Write a function `Union` which takes in two `set<int>`s and returns their union.

11. The *intersection* of two sets is the collection of elements contained in *both* of the `set`s. For example, the intersection of {1, 2, 3, 5, 8} and {2, 3, 5, 7, 11} is {2, 3, 5}. Write a function `Intersection` that takes in two `set<int>`s and returns their intersection.

12. Earlier in this chapter, we wrote a program that rolled dice until the same number was rolled twice, then printed out the number of rolls made. Rewrite this program so that the same number must be rolled *three* times before the process terminates. How many times do you expect this process to take when rolling twenty-sided dice? *(Hint: you will probably want to switch from using a `set` to using a `multiset`. Also, remember the difference between the `set`'s `count` function and the `multiset`'s `count` function).*

13. As mentioned in this chapter, you can use a combination of `lower_bound` and `upper_bound` to iterate over elements in the closed interval [min, max]. What combination of these two functions could you use to iterate over the interval [min, max)? What about (min, max] and (min, max)?

14. Write a function `NumberDuplicateEntries` that accepts a `map<string, string>` and returns the number of duplicate *values* in the `map` (that is, the number of key/value pairs in the map with the same value).

15. Write a function `InvertMap` that accepts as input a `map<string, string>` and returns a `multimap<string, string>` where each pair (key, value) in the source map is represented by (value, key) in the generated `multimap`. Why is it necessary to use a `multimap` here? How could you use the `NumberDuplicateEntries` function from the previous question to determine whether it is possible to invert the `map` into another `map`?

16. Suppose that we have two `map<string, string>`s called `one` and `two`. We can define the ***composition*** of `one` and `two` (denoted `two ○ one`) as follows: for any string `r`, if `one[r]` is `s` and `two[s]` is `t`, then (`two ○ one`)`[r] = t`. That is, looking up an element `x` in the composition of the maps is equivalent to looking up the value associated with `x` in `one` and then looking up its associated value in `two`. If `one` does not contain `r` as a key or if `one[r]` is not a key in `two`, then (`two ○ one`)`[r]` is undefined.

    Write a function `ComposeMaps` that takes in two `map<string, string>`s and returns a `map<string, string>` containing their composition.

17. (Challenge problem!) Write a function `PrintMatchingPrefixes` that accepts a `set<string>` and a `string` containing a prefix and prints out all of the entries of the `set` that begin with that prefix. Your function should only iterate over the entires it finally prints out. You can assume the prefix is nonempty, consists only of alphanumeric characters, and should treat prefixes case-sensitively. *(Hint: In a `set<string>`, strings are sorted lexicographically, so all strings that start with "abc" will come before all strings that start with "abd.")*

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.

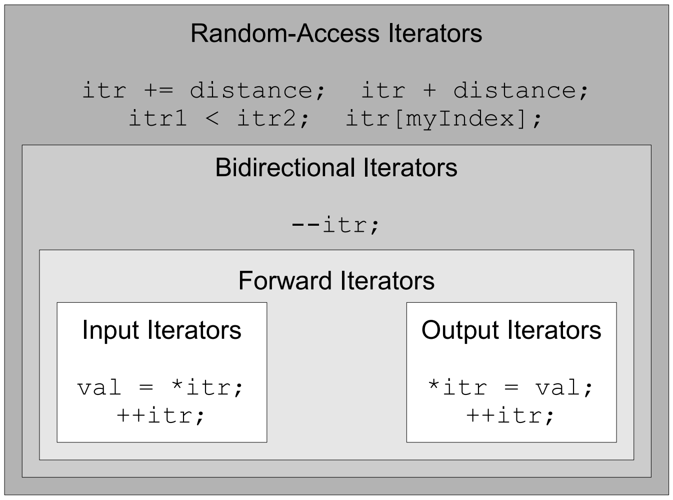
- **Input Iterators**. Input iterators are similar to output iterators except that they read values instead of writing them. That is, you can write code along the lines of `value = *myItr`, but not `*myItr = value`. Moreover, input iterators cannot iterate over the same range twice.

- **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 placeTs.* 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, vectors and deques 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:

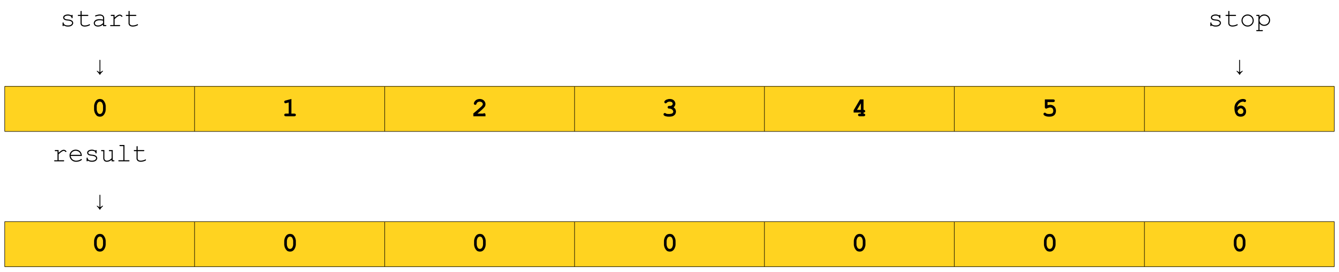
```
/* Assume myVector is sorted. */
if (binary_search(myVector.begin(), myVector.end(), 137)) {
   /* ... Found 137 ... */
}
```

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.

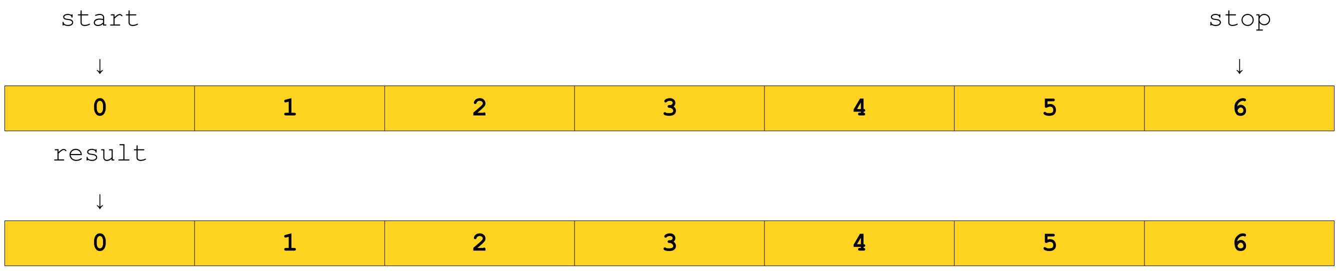
Note that binary_search doesn't return an iterator to the element – it simply checks to see if it's in the container. If you want to do a binary search in order to get an iterator to an element, you can use the lower_bound algorithm which, like the map and set lower_bound functions, returns an iterator 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 sets 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_iterators 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_iterators:

```
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_iterators` 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.

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

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 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.