Chapter 5: 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.

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.

[*] 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.

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, sets can only hold one type of element, so you cannot have a set that mixes and matches between ints and strings, 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 strings and other STL containers. However, you cannot store custom structs 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;
```

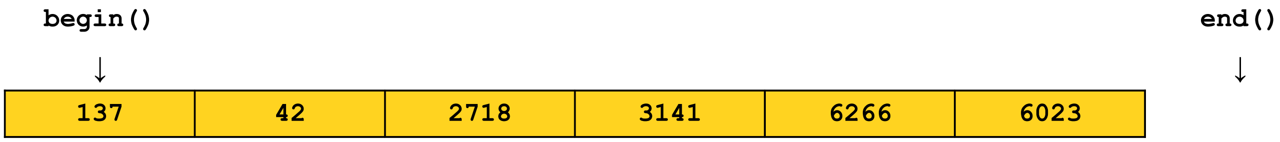
When applied to ints, the ++ operator is the increment operator; writing ++myInt means "increment the value of the myInt variable." When applied to iterators, the ++ operator means "advance the iterator one step forward." Because the step condition of the for loop is ++itr, this means that each iteration 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 strings 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 strings 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 strings 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 strings to ints, 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 structs as keys in an STL map. However, the *values* in the map needn't be comparable, so it's perfectly fine to map from strings 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 strings to ints. 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 strings 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_iterators`; 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
and_eq
asm
auto
bitand
bitor
bool
break
case
catch
char
class
compl
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
not
not_eq
operator
or
or_eq
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
xor
xor_eq
```

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 `string`s of the keywords to `size_t`s 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 maps and sets 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;
```

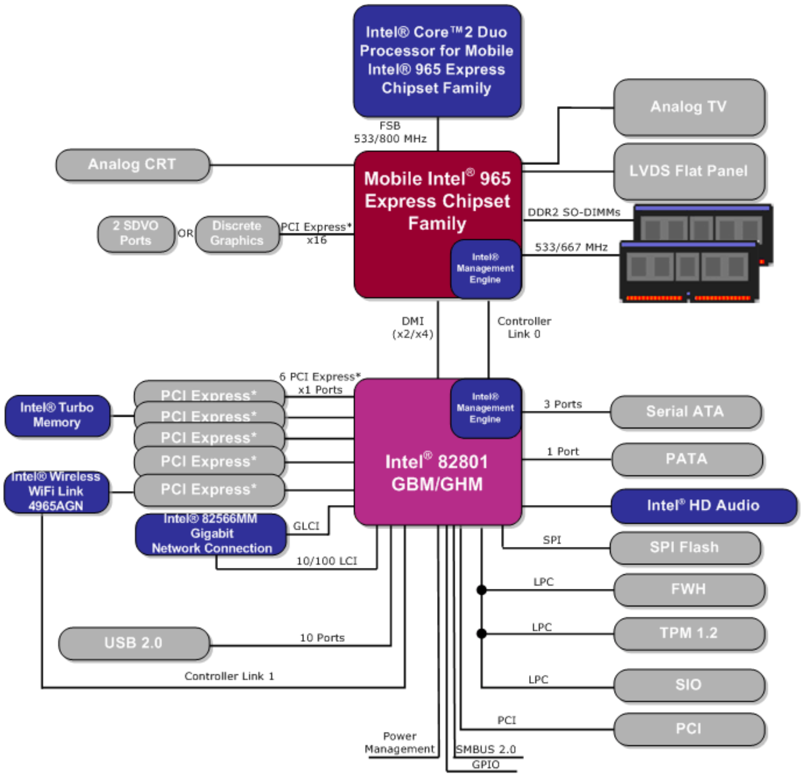
The multicontainers are fairly uncommon in practice partially because they can easily be emulated using the regular map or set. For example, a multimap<string, int> behaves similarly to a map<string, vector<int> > since both act as a map from strings to some number of ints. However, in many cases the 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]



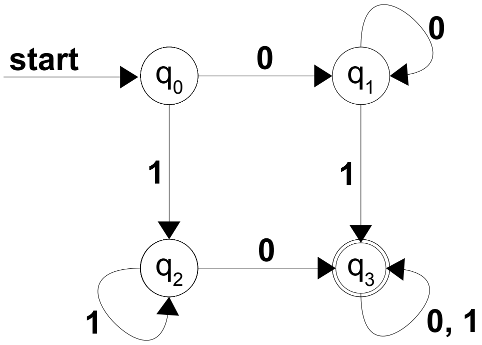
Modeling each of these components is exceptionally tricky, and trying to devise any sort of proof about the capabilities of such a machine would be all but impossible. Instead, one approach is to work with *automata*, abstract mathematical models of computing machines (the singular of *automata* is the 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:



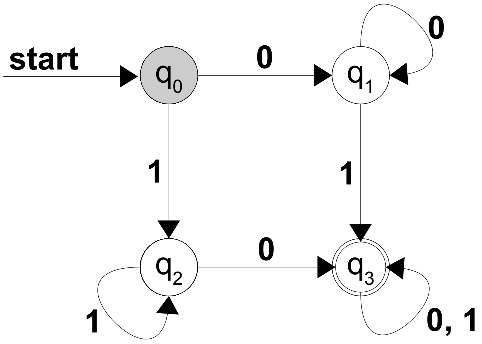
This DFA has four states, labeled q_0, q_1, q_2, and q_3, and a set of labeled transitions between those states. For example, the state q_0 has a transition labeled **0** to q_1 and a transition labeled **1** to q_2. Some states have transitions to themselves; for example, q_2 transitions to itself on 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.

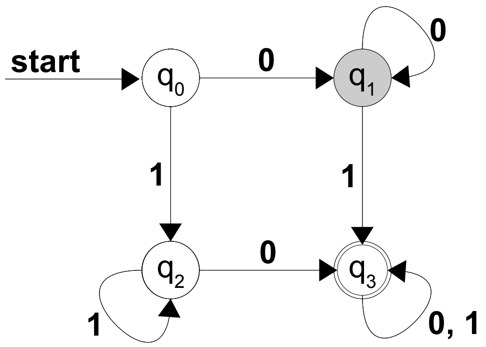
Since all of the transitions in this DFA are labeled either **0** or **1**, this DFA is said to have an *alphabet* of {**0**, **1**}. A DFA can use any nonempty set of symbols as an alphabet; for example, the Latin or Greek alphabets are perfectly acceptable for use as alphabets in a DFA, as is the set 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.

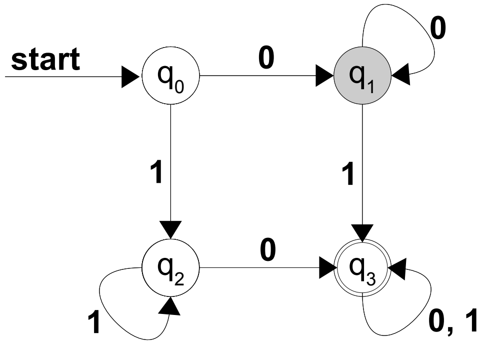
To see exactly how a DFA processes input, let us consider the above DFA simulated on the input **0011**. Initially, we begin in the start state, as shown here:



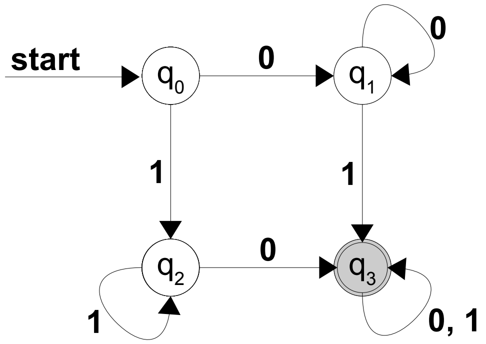
Since the first character of our string is a **0**, we follow the transition to state q_1, as shown here:



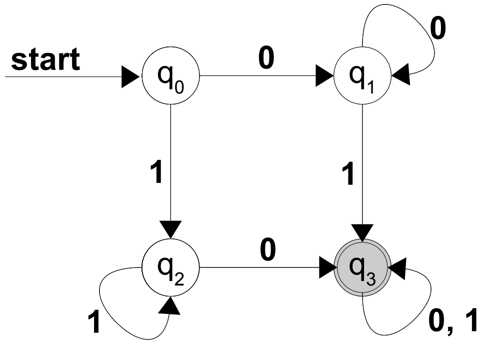
The second character of input is also a **0**, so we follow the transition labeled with a **0** and end up back in state q_1, leaving us in this state:



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



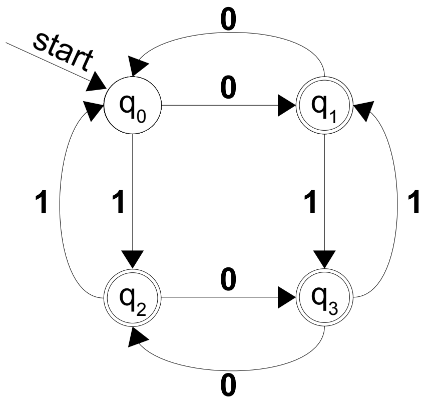
The final character of input is also a **1**, so we follow the transition labeled **1** and end back up in q_3:



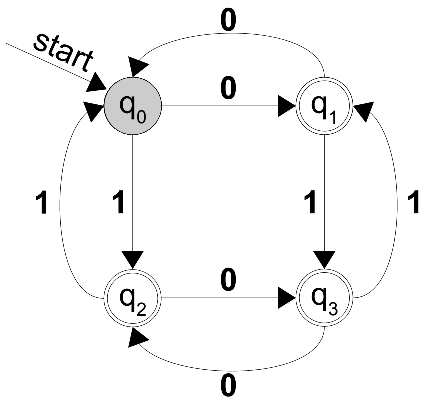
We are now done with our input, and since we have ended in an accepting state, the DFA accepts this input.

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**.

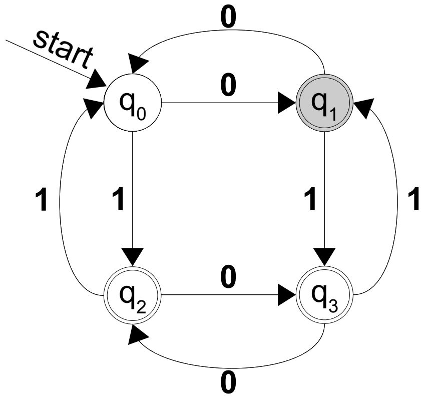
Two important details regarding DFAs deserve some mention. First, it is possible for a DFA to have multiple accepting states, as is the case in this DFA:



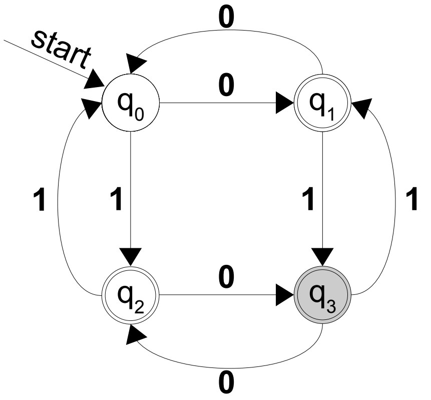
As with the previous DFA, this DFA has four states, but notice that three of them are marked as accepting. This leads into the second important detail regarding DFAs – the DFA only accepts its input if the DFA ends in an accepting state *when it runs out of input*. Simply transitioning into an accepting 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:



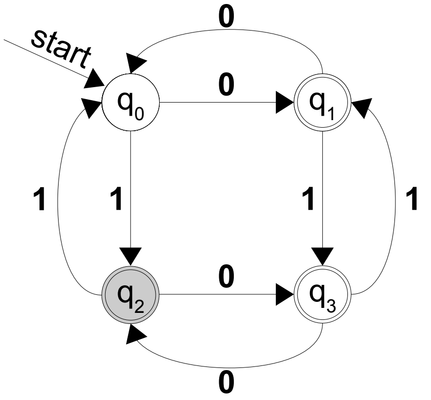
We first consume a **0**, sending us to state q_1:



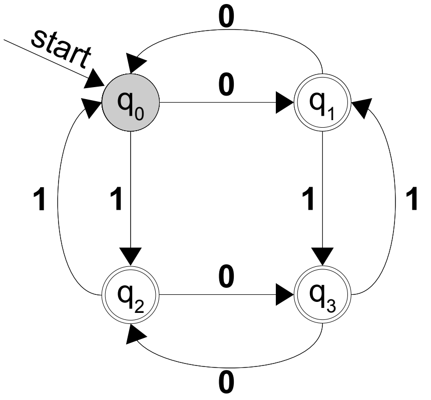
Next, we read a **1**, sending us to state q_3:



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:*

* 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.

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)
{
    /* ... */
}
```

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:[*]

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

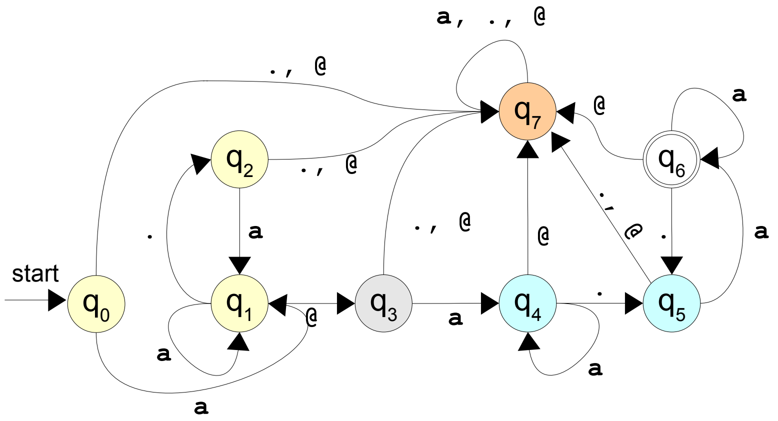
- The name field, which consists of nonempty alphanumeric strings separated by periods. Periods can only occur between alphanumeric strings, never before or after. Thus hello.world@example.com and cp-p.is.really.cool@example.com are legal but .oops@example.com, oops.@example.com, and oops..oops@example.com 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 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:

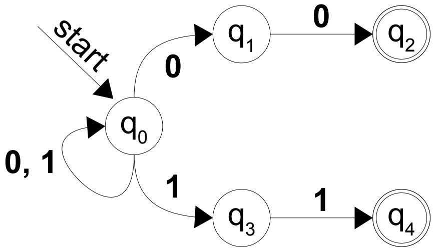
```
bool IsEmailAddress(string input)
{
    DFA emailChecker = LoadEmailDFA(); // Implemented elsewhere

    /* 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.

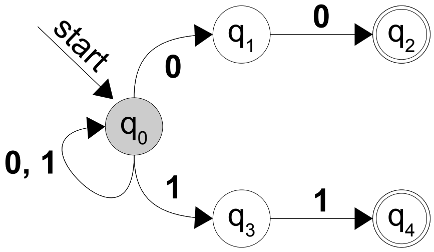
Nondeterministic Finite Automata

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:

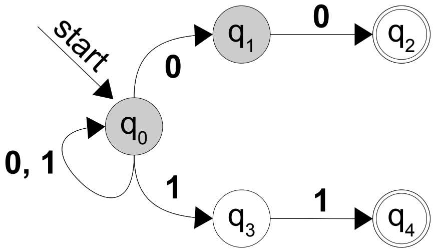


Here, the start state is q_0 and accepting states are q_2 and q_4. Notice that the start state q_0 has two transitions on **0** – one to q_1 and one to itself – and two transitions on **1**. Also, note that q_3 has no defined transitions on **0**, and states q_2 and q_4 have no transitions at all.

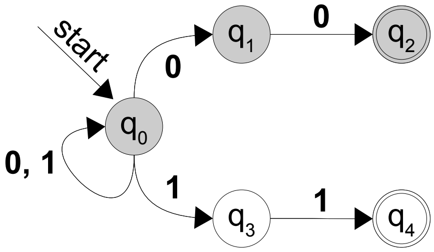
There are several ways to interpret a state having multiple transitions. The first is to view the automaton as choosing one of the paths nondeterministically (hence the name), then accepting the input if *some* set of choices results in the automaton ending in an accepting state. Another, more intuitive 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:



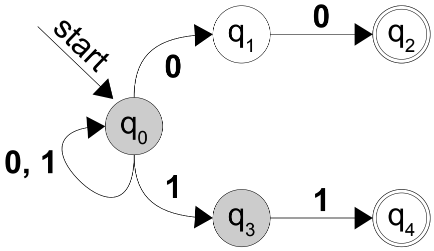
We now process the first character of input (**0**) and find that there are two transitions to follow – the first to q_0 and the second to q_1. The NFA thus ends up in both of these states simultaneously, as shown here:



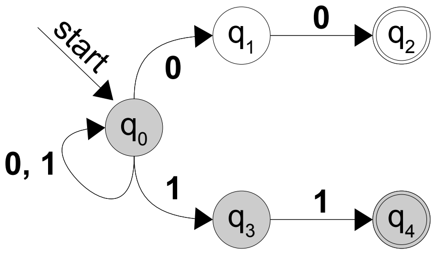
Next, we process the second character (**0**). From state q_0, we transition into q_0 and q_1, and from state q_1 we transition into q_2. We thus end up in states q_0, q_1, and q_2, as shown here:



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.")*