The iterators must support random access. The `sort` algorithm does not guarantee that equal items retain their original order (if that is important, use `stable_sort` instead of `sort`).

As an example, in

```
std::sort( v.begin( ), v.end( ) );
std::sort( v.begin( ), v.end( ), greater<int>{ } );
std::sort( v.begin( ), v.begin( ) + ( v.end( ) - v.begin( ) ) / 2 );
```

the first call sorts the entire container, `v`, in nondecreasing order. The second call sorts the entire container in nonincreasing order. The third call sorts the first half of the container in nondecreasing order.

The sorting algorithm used is generally quicksort, which we describe in Section 7.7. In Section 7.2, we implement the simplest sorting algorithm using both our style of passing the array of comparable items, which yields the most straightforward code, and the interface supported by the STL, which requires more code.

7.2 Insertion Sort

One of the simplest sorting algorithms is the **insertion sort**.

7.2.1 The Algorithm

Insertion sort consists of $N-1$ **passes.** For pass $p=1$ through $N-1$, insertion sort ensures that the elements in positions 0 through p are in sorted order. Insertion sort makes use of the fact that elements in positions 0 through $p-1$ are already known to be in sorted order. Figure 7.1 shows a sample array after each pass of insertion sort.

Figure 7.1 shows the general strategy. In pass p, we move the element in position p left until its correct place is found among the first $p+1$ elements. The code in Figure 7.2 implements this strategy. Lines 11 to 14 implement that data movement without the explicit use of swaps. The element in position p is moved to `tmp`, and all larger elements (prior to position p) are moved one spot to the right. Then `tmp` is moved to the correct spot. This is the same technique that was used in the implementation of binary heaps.

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure 7.1 Insertion sort after each pass

```
1    /**
2     * Simple insertion sort.
3     */
4    template <typename Comparable>
5    void insertionSort( vector<Comparable> & a )
6    {
7        for( int p = 1; p < a.size( ); ++p )
8        {
9            Comparable tmp = std::move( a[ p ] );
10
11           int j;
12           for( j = p; j > 0 && tmp < a[ j - 1 ]; --j )
13               a[ j ] = std::move( a[ j - 1 ] );
14           a[ j ] = std::move( tmp );
15       }
16   }
```

Figure 7.2 Insertion sort routine

7.2.2 STL Implementation of Insertion Sort

In the STL, instead of having the sort routines take an array of comparable items as a single parameter, the sort routines receive a pair of iterators that represent the start and endmarker of a range. A two-parameter sort routine uses just that pair of iterators and presumes that the items can be ordered, while a three-parameter sort routine has a function object as a third parameter.

Converting the algorithm in Figure 7.2 to use the STL introduces several issues. The obvious issues are

1. We must write a two-parameter sort and a three-parameter sort. Presumably, the two-parameter sort invokes the three-parameter sort, with `less<Object>{ }` as the third parameter.
2. Array access must be converted to iterator access.
3. Line 11 of the original code requires that we create `tmp`, which in the new code will have type `Object`.

The first issue is the trickiest because the template type parameters (i.e., the generic types) for the two-parameter sort are both `Iterator`; however, `Object` is not one of the generic type parameters. Prior to C++11, one had to write extra routines to solve this problem. As shown in Figure 7.3, C++11 introduces `decltype` which cleanly expresses the intent.

Figure 7.4 shows the main sorting code that replaces array indexing with use of the iterator, and that replaces calls to `operator<` with calls to the `lessThan` function object.

Observe that once we actually code the `insertionSort` algorithm, every statement in the original code is replaced with a corresponding statement in the new code that makes

```
/*
 * The two-parameter version calls the three-parameter version,
 * using C++11 decltype
 */
template <typename Iterator>
void insertionSort( const Iterator & begin, const Iterator & end )
{
    insertionSort( begin, end, less<decltype(*begin)>{ } );
}
```

Figure 7.3 Two-parameter sort invokes three-parameter sort via C++11 `decltype`

```
template <typename Iterator, typename Comparator>
void insertionSort( const Iterator & begin, const Iterator & end,
                    Comparator lessThan )
{
    if( begin == end )
        return;

    Iterator j;

    for( Iterator p = begin+1; p != end; ++p )
    {
        auto tmp = std::move( *p );
        for( j = p; j != begin && lessThan( tmp, *( j-1 ) ); --j )
            *j = std::move( *(j-1) );
        *j = std::move( tmp );
    }
}
```

Figure 7.4 Three-parameter sort using iterators

straightforward use of iterators and the function object. The original code is arguably much simpler to read, which is why we use our simpler interface rather than the STL interface when coding our sorting algorithms.

7.2.3 Analysis of Insertion Sort

Because of the nested loops, each of which can take N iterations, insertion sort is $O(N^2)$. Furthermore, this bound is tight, because input in reverse order can achieve this bound. A precise calculation shows that the number of tests in the inner loop in Figure 7.2 is at most $p + 1$ for each value of p. Summing over all p gives a total of

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$

On the other hand, if the input is presorted, the running time is $O(N)$, because the test in the inner `for` loop always fails immediately. Indeed, if the input is almost sorted (this term will be more rigorously defined in the next section), insertion sort will run quickly. Because of this wide variation, it is worth analyzing the average-case behavior of this algorithm. It turns out that the average case is $\Theta(N^2)$ for insertion sort, as well as for a variety of other sorting algorithms, as the next section shows.

7.3 A Lower Bound for Simple Sorting Algorithms

An **inversion** in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$. In the example of the last section, the input list 34, 8, 64, 51, 32, 21 had nine inversions, namely $(34, 8)$, $(34, 32)$, $(34, 21)$, $(64, 51)$, $(64, 32)$, $(64, 21)$, $(51, 32)$, $(51, 21)$, and $(32, 21)$. Notice that this is exactly the number of swaps that needed to be (implicitly) performed by insertion sort. This is always the case, because swapping two adjacent elements that are out of place removes exactly one inversion, and a sorted array has no inversions. Since there is $O(N)$ other work involved in the algorithm, the running time of insertion sort is $O(I + N)$, where I is the number of inversions in the original array. Thus, insertion sort runs in linear time if the number of inversions is $O(N)$.

We can compute precise bounds on the average running time of insertion sort by computing the average number of inversions in a permutation. As usual, defining *average* is a difficult proposition. We will assume that there are no duplicate elements (if we allow duplicates, it is not even clear what the average number of duplicates is). Using this assumption, we can assume that the input is some permutation of the first N integers (since only relative ordering is important) and that all are equally likely. Under these assumptions, we have the following theorem:

Theorem 7.1

The average number of inversions in an array of N distinct elements is $N(N-1)/4$.

Proof

For any list, L, of elements, consider L_r, the list in reverse order. The reverse list of the example is $21, 32, 51, 64, 8, 34$. Consider any pair of two elements in the list (x, y) with $y > x$. Clearly, in exactly one of L and L_r this ordered pair represents an inversion. The total number of these pairs in a list L and its reverse L_r is $N(N-1)/2$. Thus, an average list has half this amount, or $N(N-1)/4$ inversions.

This theorem implies that insertion sort is quadratic on average. It also provides a very strong lower bound about any algorithm that only exchanges adjacent elements.

Theorem 7.2

Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average.