

Sorting, Templates, and Generic Programming

11

No transcendent ability is required in order to make useful discoveries in science; the edifice of science needs its masons, bricklayers, and common labourers as well as its foremen, master-builders, and architects. In art nothing worth doing can be done without genius; in science even a very moderate capacity can contribute to a supreme achievement.

Bertrand Russell
Mysticism and Logic

Many human activities require collections of items to be put into some particular order. The post office sorts mail by ZIP code for efficient delivery; telephone books are sorted by name to facilitate finding phone numbers; and hands of cards are sorted by suit to make it easier to go fish. **Sorting** is a task performed well by computers; the study of different methods of sorting is also intrinsically interesting from a theoretical standpoint. In Chapter 8 we saw how fast binary search is, but binary search requires a sorted list as well as random access. In this chapter we'll study different sorting algorithms and methods for comparing these algorithms. We'll extend the idea of conforming interfaces we studied in Chapter 7 (see Section 7.2) and see how conforming interfaces are used in **template** classes and functions. We'll study sorting from a theoretical perspective, but we'll also emphasize techniques for making sorting and other algorithmic programming more efficient in practice using **generic programming** and **function objects**.

11.1 Sorting an Array

There are several elementary sorting algorithms, details of which can be found in books on algorithms; Knuth [Knu98b] is an encyclopedic reference. Some of the elementary algorithms are much better than others, both in terms of performance and in terms of ease of coding. Contrary to what some books on computer programming claim, there are large differences between these elementary algorithms. In addition, these elementary algorithms are more than good enough for sorting reasonably large vectors,¹ *provided that the good elementary algorithms are used*.

In particular, there are three “classic” simple sorting algorithms. Each of these sorts is a **quadratic** algorithm. We'll define quadratic more carefully later, but quadratic algorithms behave like a quadratic curve $y = x^2$ in mathematics: if a vector of 1000 elements is sorted in 4 seconds, a vector of 2000 elements will be sorted in 16 seconds using a quadratic algorithm. We'll study sorting algorithms that are more efficient than these quadratic sorts, but the quadratic sorts are a good place to start.

The three basic sorting algorithms are:

¹What is “reasonably large”? The answer, as it often is, is “It depends”—on the kind of element sorted, the kind of computer being used, and on how fast “pretty fast” is.

- Selection sort
- Insertion sort
- Bubble sort

We'll develop selection sort in this section. You've already seen insertion sort in Section 8.3.4 where an element is inserted into an already-sorted vector and the vector is kept in sorted order. However, a few words are needed about bubble sort.

Program Tip 11.1: Under no circumstances should you use bubble sort.

Bubble sort is the slowest of the elementary sorts, for reasons we'll explore as an exercise. Bubble sort is worth knowing about only so that you can tell your friends what a poor sort it is. Although interesting from a theoretical perspective, bubble sort has no practical use in programming on a computer with a single processor.

11.1.1 Selection Sort

The basic algorithm behind selection sort is quite simple and is similar to the method used in shuffling tracks of a CD explored and programmed in *shuffle.cpp*, Program 8.4. To sort from smallest to largest in a vector named `A`, the following method is used:

1. Find the smallest entry in `A`. Swap it with the first element `A[0]`. Now the smallest entry is in the first location of the vector.
2. Considering only vector locations `A[1]`, `A[2]`, `A[3]`, ...; find the smallest of these and swap it with `A[1]`. Now the first two entries of `A` are in order.
3. Continue this process by finding the smallest element of the remaining vector elements and swapping it appropriately.

This algorithm is outlined in code the function `SelectSort` of Program 11.1 which sorts an `int` vector. Each time through the loop in the function `SelectSort`, the index of the smallest entry of those not yet in place (from `k` to the end of the vector) is determined by calling the function `MinIndex`. This function (which will be shown shortly) returns the index, or location, of the smallest element, which is then stored/swapped into location `k`. This process is diagrammed in Figure 11.1. The shaded boxes represent vector elements that are in their final position. Although only five elements are shaded in the last "snapshot," if five out of six elements are in the correct position, the sixth element must be in the correct position as well.

23 18 42 7 57 38 0 1 2 3 4 5	MinIndex = 3	Swap(a[0],a[3]);
7 18 42 23 57 38 0 1 2 3 4 5	MinIndex = 1	Swap(a[1],a[1]);
7 18 42 23 57 38 0 1 2 3 4 5	MinIndex = 3	Swap(a[2],a[3]);
7 18 23 42 57 38 0 1 2 3 4 5	MinIndex = 5	Swap(a[3],a[5]);
7 18 23 38 57 42 0 1 2 3 4 5	MinIndex = 5	Swap(a[4],a[5]);
7 18 23 38 42 57 0 1 2 3 4 5		

Figure 11.1 Selection sort.

Program 11.1 selectsort1.cpp

```

#include "tvector.h"
int MinIndex(tvector<int> & a, int first, int last);
// precondition: 0 <= first, first <= last
// postcondition: returns k such that a[k] <= a[j], j in [first..last]
//                i.e., index of minimal element in a

void Swap(int & a, int & b);
// postcondition: a and b interchanged/swapped

void SelectSort(tvector<int> & a)
// precondition: a contains a.size() elements
// postcondition: elements of a are sorted in non-decreasing order
{
    int k, index, numElts = a.size();

    // invariant: a[0]..a[k-1] in final position
    for(k=0; k < numElts - 1; k+=1)
    {
        index = MinIndex(a,k,numElts - 1); // find min element
        Swap(a[k],a[index]);
    }
}

```

selectsort1.cpp

Each time the loop test `k < numElts - 1` is evaluated, the statement “elements `a[0]..a[k-1]` are in their final position” is true. Recall that any statement that is true each time a loop test is evaluated is called a **loop invariant**. In this case the statement

is true because the first time the loop test is evaluated, the range $0 \dots k-1$ is $[0 \dots -1]$, which is an empty range, consisting of no vector elements. As shown in Figure 11.1, the shaded vector elements indicate that the statement holds after each iteration of the loop. The final time the loop test is evaluated, the value of k will be `numElts - 1`, the last valid vector index. Since the statement holds (it holds each time the test is evaluated), the vector must be sorted. The function `MinIndex` is straightforward to write:

```
int MinIndex(const tvector<int> & a, int first, int last)
// pre: 0 <= first, first <= last
// post: returns index of minimal element in a[first..last]
{
    int smallIndex = first;
    int k;
    for(k=first+1; k <= last; k++)
    {   if (a[k] < a[smallIndex] )
        {   smallIndex = k;
            }
    }
    return smallIndex;
}
```

`MinIndex` finds the minimal element in an array; it's similar to code discussed in Section 6.4 for finding largest and smallest values. The first location of the vector `a` is the initial value of `smallIndex`, then all other locations are examined. If a smaller entry is found, the value of `smallIndex` is changed to record the location of the new smallest item.

Program 11.2 `selectsort2.cpp`

```
void SelectSort(tvector<int> & a)
// pre: a contains a.size() elements
// post: elements of a are sorted in non-decreasing order
{
    int j,k,temp,minIndex,numElts = a.size();

    // invariant: a[0]..a[k-1] in final position
    for(k=0; k < numElts - 1; k++)
    {   minIndex = k;           // minimal element index
        for(j=k+1; j < numElts; j++)
        {   if (a[j] < a[minIndex])
            {   minIndex = j;           // new min, store index
                }
            }
        temp = a[k];           // swap min and k-th elements
        a[k] = a[minIndex];
        a[minIndex] = temp;
    }
}
```

`selectsort2.cpp`

The function `MinIndex`, combined with `Swap` and `SelectSort`, yields a complete implementation of selection sort. Sometimes it's convenient to have all the code in one function rather than spread over three functions. This is certainly possible and leads to the code shown in *selectsort2.cpp*, Program 11.2. However, as you develop code, it's often easier to test and debug when separate functions are used. This allows each piece of code to be tested separately.

The code in Program 11.2 works well for sorting a vector of numbers, but what about sorting vectors of strings or some other kind of element? If two vector elements can be compared, then the vector can be sorted based on such comparisons. A vector of strings can be sorted using the same code provided in the function `SelectSort`; the only difference in the functions is the type of the first parameter and the type of the local variable `temp`, as follows:

```
void SelectSort(tvector<string> & a)
// pre: a contains a.size() elements
// post: elements of a are sorted in nondecreasing order
{
    int j, k, minIndex, numElts = a.size();
    string temp;
    // code here doesn't change
}
```

Both this function and `SelectSort` in *selectsort2.cpp* could be used in the same program since the parameter lists are different. In previous chapters we overloaded the `+` operator so that we could use it both to add numbers and to concatenate strings. We've also used the function `toString` from *strutils.h* (see How to G) to convert both doubles and ints to strings; there are two functions with the same name but different parameters. In the same way, we can overload function names. Different functions with the same name can be used in the same program provided that the parameter lists of the functions are different. In these examples the function `Sort` is overloaded using three different



Syntax: Function overloading

```
void Sort(tvector<string>& a);
void Sort(tvector<double>& a);
void Sort(tvector<int>& a);
int DoStuff(int a, int b);
int DoStuff(int a, int b, int c);
```

kinds of vectors. `DoStuff` is overloaded, since there are two versions with different parameter lists. The names of the parameters do not matter; only the types of the parameters are important in resolving which overloaded function is actually called. It is *not* possible, for example, to use the two versions of

`FindRoots` below in the same program, because the parameter lists are the same. The different return types are not sufficient to distinguish the functions:

```
int FindRoots(double one, double two);
double FindRoots(double first, double second);
```

Shafi Goldwasser (b. 1958)

Shafi Goldwasser is Professor of Computer Science at MIT. She works in the area of computer science known as **theory**, but her work has practical implications in the area of secure cryptographic protocols—methods that ensure that information can be reliably transmitted between two parties without electronic eavesdropping. In particular, she is interested in using randomness in designing algorithms. She was awarded the first Gödel prize in theoretical computer science for her work. So-called **randomized algorithms** involve (simulated) coin flips in making decisions. In [Wei94] a randomized method of giving quizzes is described. Suppose a teacher wants to ensure that students do a take-home quiz, but does not want to grade quizzes every day. A teacher can give out quizzes in one class, then in the next class flip a coin to determine whether the quizzes are handed in. In the long run, this results in quizzes being graded 50 percent of the time, but students will need to do all the quizzes. Goldwasser is a coinventor of **zero-knowledge interactive proof protocols**. This mouthful is described in [Har92] as follows:

Suppose Alice wants to convince Bob that she knows a certain secret, but she does not want Bob to end up knowing the secret himself. This sounds impossible: How do you convince someone that you know, say, what color tie the president of the United States is wearing right now, without somehow divulging that priceless piece of information to the other person or to some third party?

Using zero-knowledge interactive proofs it is possible to do this. The same concepts make it possible to develop smart cards that would let people be admitted to a secure environment without letting anyone know exactly who has entered. In some colleges, cards are used to gain admittance to dormitories. Smart cards could be used to gain admittance without allowing student movement to be tracked.

Goldwasser has this to say about choosing what area to work in:

Choosing a research area, like most things in life, is not the same as solving an optimization problem. Work on what you like, what feels right. I know of no other way to end up doing creative work.

For more information see [EL94].

11.1.2 Insertion Sort

We've already discussed the code for insertion sort and used it in `stocks2.cpp`, a program discussed in Section 8.3.4; the code is shown in that section. The invariant for selection sort states that elements with indexes $0..k - 1$ are in their final position. In contrast, the insertion sort invariant states that elements with indexes $0..k - 1$ are sorted relative to each other, but are not (necessarily) in their final position. Program 11.3 shows the code for insertion sort and Figure 11.2 shows a vector being sorted during each iteration of

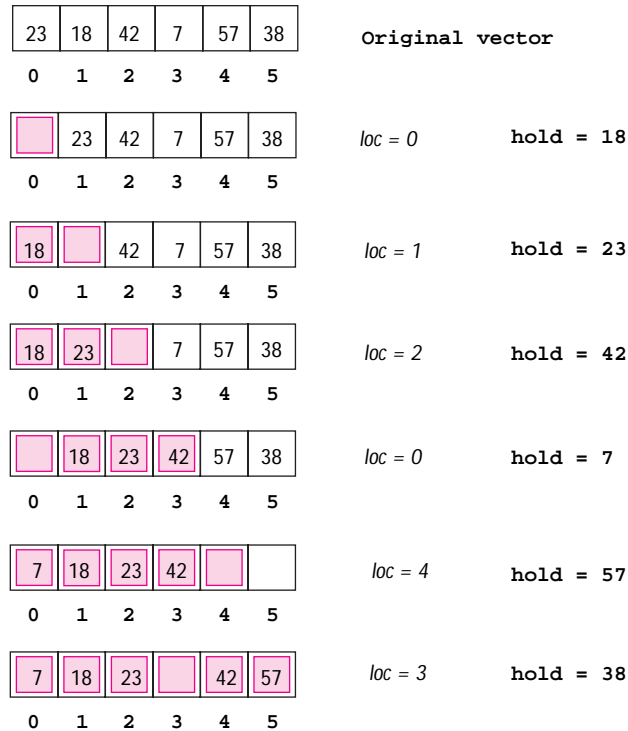


Figure 11.2 Insertion sort.

the outer for loop.

To re-establish the invariant, the inner while loop shifts elements until the location where `hold` (originally `a[k]`) belongs is determined. If `hold` is already in its correct position relative to the elements that precede it, that is, it's larger than the element to its left, then the inner while loop doesn't iterate at all. This is illustrated in Figure 11.2 when `hold` has the values 23, 42, and 57. In those cases no vector elements are shifted. When `hold` has the value 7, all the elements that precede it in the vector are larger, so all are shifted and the element 7 is stored in first (index zero) vector location. Although the outer loops in Program 11.2 and Program 11.3 iterate the same number of times, it's possible for the inner loop of insertion sort (Program 11.3) to iterate fewer times than the corresponding inner loop of selection sort (Program 11.2.)

Program 11.3 `insertsort.cpp`

```
void InsertSort(tvector<string> & a)
// precondition: a contains a.size() elements
// postcondition: elements of a are sorted in non-decreasing order
```

532

Chapter 11 Sorting, Templates, and Generic Programming

```

{
    int k, loc, numElts = a.size();

    // invariant: a[0]..a[k-1] sorted
    for(k=1; k < numElts; k++)
    {
        string hold = a[k]; // insert this element
        loc = k;           // location for insertion

        // shift elements to make room for hold/a[k]
        while (0 < loc && hold < a[loc-1])
        {
            a[loc] = a[loc-1];
            loc--;
        }
        a[loc] = hold;
    }
}

```

insertsort.cpp

We'll discuss why both insertion sort and selection sort are called quadratic sorts in more detail in Section 11.4. However, the graph of execution times for the quadratic sorts given in Figure 11.3 provides a clue; the shape of each curve is quadratic. These timings are from a single run of *timequadsorts.cpp*, Program 11.4, shown below. For more accurate empirical results you would need to run the program with different vectors, that is, for more than one trial at each vector size. A more thorough empirical analysis of sorts is explored in the exercises for this chapter.

Program 11.4 timequadsorts.cpp

```

#include <iostream>
#include <string>
using namespace std;
#include "ctimer.h"
#include "tvector.h"
#include "sortall.h"
#include "randgen.h"
#include "prompt.h"

// compare running times of quadratic sorts

void Shuffle(tvector<int> & a, int count)
// precondition: a has space for count elements
// postcondition: a contains 0..count-1 randomly shuffled
{
    RandGen gen; // for random # generator
    int randIndex, k;

    // fill with values 0..count-1
    for(k=0; k < count; k++)
    {
        a[k] = k;
    }
    // choose random index from k..count-1 and interchange with k
    for(k=0; k < count - 1; k++)
    {
        randIndex = gen.RandInt(k, count-1); // random index
    }
}

```

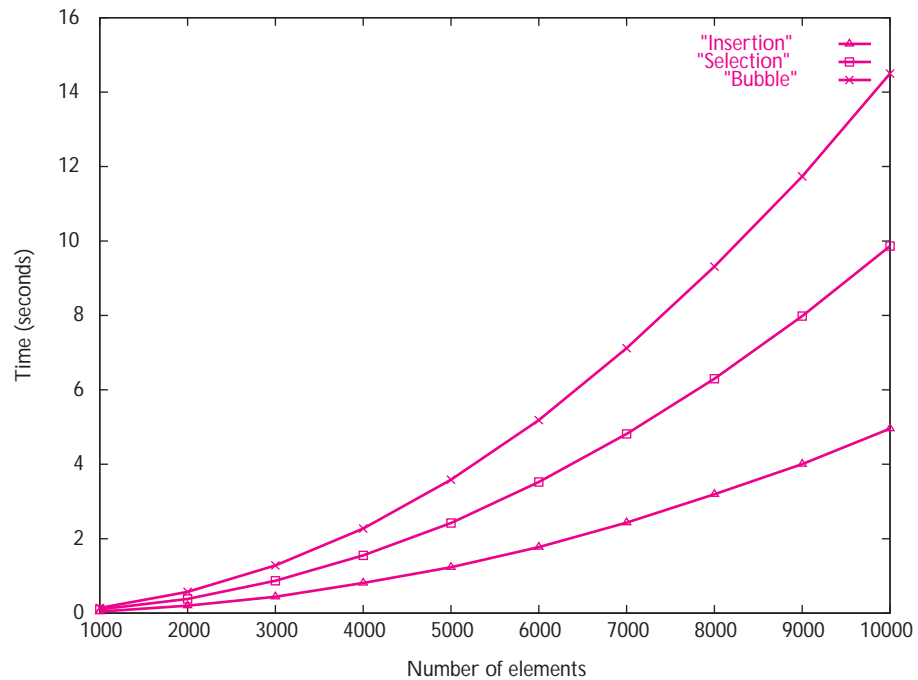


Figure 11.3 Execution times for quadratic sorts of int vectors on a Pentium II/300 running Windows NT.

```

        Swap(a,k,randIndex);           // swap in sortall.h
    }
}

int main()
{
    int size,minSize,maxSize,incr; // sort minSize, minSize+incr, ... maxSize
    CTimer timer;

    cout << "min and max size of vector: ";
    cin >> minSize >> maxSize;
    incr = PromptRange("increment in vector size",1,1000);

    cout << endl << "n\tinsert\tselect\tbubble" << endl << endl;
    for(size=minSize; size <= maxSize; size += incr)
    {
        tvector<int> copy(size), original(size);
        cout << size << "\t";
        Shuffle(original,size);

        copy = original; // sort using insertion sort
        timer.Start();
    }
}

```

534

Chapter 11 Sorting, Templates, and Generic Programming

```

InsertSort(copy, copy.size());
timer.Stop();
cout << timer.ElapsedTime() << "\t";

copy = original;    // sort using selection sort
timer.Start();
SelectSort(copy, copy.size());
timer.Stop();
cout << timer.ElapsedTime() << "\t";

copy = original;    // sort using bubble sort
timer.Start();
BubbleSort(copy, copy.size());
timer.Stop();
cout << timer.ElapsedTime() << endl;
}
return 0;
}

```

timequadsorts.cpp

O U T P U T

```

prompt> timequadsorts
min and max size of vector: 1000 10000
increment in vector size between 1 and 1000: 1000

n          insert  select  bubble
1000      0.04    0.1    0.14
2000      0.2    0.381  0.571
3000      0.44    0.871  1.282
4000      0.811  1.553  2.273
5000      1.232  2.423  3.585
6000      1.773  3.525  5.188
7000      2.433  4.817  7.12
8000      3.195  6.299  9.313
9000      4.006  7.982  11.736
10000     4.958  9.864  14.501

```

Pause to Reflect



- 11.1** What changes are necessary in Program 11.2, *selectsort2.cpp*, so that the vector *a* is sorted into decreasing order rather than into increasing order? For example, why is it a good idea to change the name of the identifier *minIndex*, although the names of variables don't influence how a program executes?
- 11.2** Why is $k < \text{numElts} - 1$ the test of the outer for loop in the selection sort code instead of $k < \text{numElts}$? Could the test be changed to the latter?

- 11.3** How many swaps are made when selection sort is used to sort an n -element vector? How many times is the statement `if (a[j] < a[minIndex])` executed when selection sort is used to sort a 5-element vector, a 10-element vector, and an n -element vector?
- 11.4** How can you use the sorting functions to “sort” a number so that its digits are in increasing order? For example, 7216 becomes 1267 when sorted. Describe what to do and then write a function that sorts a number (you can call one of the sorting functions from this section if that helps.)
- 11.5** If insertion sort is used to sort a vector that’s already sorted, how many times is the statement `a[loc] = a[loc-1];` executed?
- 11.6** If the vector `counts` from Program 8.3, *letters.cpp*, is passed to the function `SelectSort`, why won’t the output of the program be correct?
- 11.7** In the output from *timequadsorts.cpp*, Program 11.4 the ratio of the timings when the size of the vector doubles from 4000 elements to 8000 is given in Table 11.1.

Table 11.1 Timing Quadratic Sorts

Sort	4000 elts.	8000 elts.	ratio
insertion	0.811	3.195	3.939
select	1.553	6.299	4.056
bubble	3.585	9.313	4.097

Assuming the ratio holds consistently (rounded to 4) how long will it take each quadratic algorithm to sort a vector of 16,000 elements? 32,000 elements? 1,000,000 elements?

11.2 Function Templates

Although it is possible to overload a function name, the sorting functions in the previous sections are not ideal candidates for function overloading. If we write a separate function to use selection sort with a vector of ints, a vector of strings, and a vector of doubles the code that would appear in each function is nearly identical. The only differences in the code would be in the definition of the variable `temp` and in the kind of `tvector` passed to the function. Consider what might happen if a more efficient sorting algorithm is required. The code in each of the three functions must be removed, and the code for the more efficient algorithm inserted. Maintaining three versions of the function makes it much more likely that errors will eventually creep into the code, because it is difficult to ensure that whenever a modification is made to one sorting function, it is made to all of the sorting functions (see Program Tip. 4.1.)

Fortunately, a mechanism exists in C++ that allows code to be reused rather than replicated. We have already used this mechanism behind the scenes in the implementation of the `tvector` class and the `CList` class.

A **function template**, sometimes called a **templated function**, can be used when different types are part of the parameter list and the types conform to an interface used in the function body. For example, to sort a vector of a type `T` using selection sort we must be able to compare values of type `T` using the relational operator `<` since that's how elements are compared. We wouldn't expect to be able to sort `Dice` objects since they're not comparable using relational operators. We should be able to sort `ClockTime` objects (see *clockt.h*, Program 9.9 in How to G) since they're comparable using operator `<`. The sorting functions require objects that conform to an interface of being comparable using operator `<`. A templated function allows us to capture this interface in code so that we can write one function that works with any type that can be compared. We'll study templates in detail, building towards them with a series of examples.



11.2.1 Printing a `tvector` with a Function Template

Program 11.5, *sortwlen.cpp* reads a files and tracks all words and word lengths. Both words and lengths are sorted using the templated function `SelectSort` from *sortall.h*. A templated function `Print` that prints both string and int vectors is shown in *sortwlen.cpp*.

Program 11.5 *sortwlen.cpp*

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
#include "tvector.h"
#include "prompt.h"
#include "sortall.h"

// illustrates templates, read words, sort words, print words

template <class Type>
void Print(const tvector<Type>& list, int first, int last)
// post: list[first]..list[last] printed, one per line
{
    int k;
    for(k=first; k <= last; k++)
    {    cout << list[k] << endl;
    }
}

int main()
{
    tvector<string> wordList;
    tvector<int>    lengthList;
    string filename = PromptString("filename: ");
    string word;
    ifstream input(filename.c_str());
```

```

while (input >> word)
{
    wordList.push_back(word);
    lengthList.push_back(word.length());
}
SelectSort(wordList,wordList.size());
SelectSort(lengthList,lengthList.size());

Print(wordList,wordList.size()-5,wordList.size()-1);
Print(lengthList,lengthList.size()-5,lengthList.size()-1);

return 0;
}

```

sortwlen.cpp

OUTPUT

```

prompt > sortwlen
filename: poet.txt
your
your
your
your
your
16
16
18
18
19

```

Just as not all types of vectors can be sorted, not all types of vectors can be printed. The function `Print` in `sortwlen.cpp` expects a vector whose type conforms to the interface of being insertable onto a stream using operator `<<`. Consider an attempt to print a vector of vectors.

```

tvector<tvector<int> > ivlist;
ivlist.push_back(lengthList);
Print(ivlist,0,1);

```

The first two lines compile without trouble, but the attempt to pass `ivlist` to `Print` fails because the type `tvector<int>` does not conform to the expected interface: there is no overloaded operator `<<` for `tvector<int>` objects. The error messages generated by different compilers vary from informative to incomprehensible.

The error message from Visual C++ 6.0 is very informative.

```
sortwlen.cpp(17) : error C2679: binary '<<' : no operator
defined which takes a right-hand operand of type
'const class tvector<int>'
```

The error message from the Metrowerks Codewarrior compiler is less informative.

```
Error      : illegal operand
sortwlen.cpp line 17 { cout << list[k] << endl;
```

The error message from the Linux g++ compiler is informative, though difficult to comprehend.

```
'void Print<tvector<int>>(const class
tvector<tvector<int> >&)' : sortwlen.cpp:17:
no match for '_IO_ostream_withassign
& << const tvector<int> &'
```

In general, a call of a templated function fails if the type of argument passed can't be used in the function because the expected conforming interface doesn't apply. What that means in the example of `Print` is spelled out clearly by the error messages: the type `const tvector<int>` can't be inserted onto a stream and stream insertion using operator `<<` is the interface expected of vector elements when `Print` is called.

One reason the error messages can be hard to understand is that the compiler *catches* the error and indicates its source in the templated function: line 17 in the example of `Print` above (see the error message). However, the error *is caused* by a call of the templated function, what's termed the template function **instantiation**, and you must determine what call or instantiation causes the error. In a small program this can be straightforward, but in a large program you may not even be aware that a function (or class) is templated. Since the error messages don't show the call, finding the real source can be difficult.

Program Tip 11.2: If the compiler indicates an error in a templated function or class, look carefully at the error message and try to find the template instantiation that causes the problem. The instantiation that causes the problem may be difficult to find, but searching for the name of the class or function in which the error occurs using automated Find/Search capabilities may help.

A function is declared as a templated function when it is preceded by the word **template** followed by an angle bracket delimited list of class identifiers that serve as type parameters. At least one of the type parameters must be used in the function's parameter list. Any name can be the template parameter, such as `Type`, `T`, and `U` as

shown in the syntax diagram.

This name can be used for the return type, as the type (or part of a type) in the parameter list, or as the type of a local variable in the function body. It is possible to have more than one template class parameter. The function `doThat` in the syntax box has two template

Syntax: Function Template

```
template <class T> void doIt(T& t);

template <class Type>
Type minElt(const tvector<Type>);

template <class T, class U>
void doThat(CList<T> t, CList<U> u);
```

parameters, `T` and `U`. `doThat` could be called as follows:

```
doThat(cons(3, CList<int>()),
       cons(string("help"), CList<string>()));
```

Here the template parameter `T` is **bound to** or **unified with** the type `int`, and the template type `U` is bound to `string`. If the template instantiation succeeds, all uses of `T` and `U` in the body of the function `doThat` will be supported by the types `int` and `string`, respectively.

11.2.2 Function Templates and Iterators

We've seen many different iterator classes: `WordStreamIterator` for iterating over each word in a file, `CListIterator` for iterating over the contents of a list, `RandomWalk` for iterating over a simulated random walk, and even `Permuter` for generating permutations of an `int` vector. Because all these classes adhere to the same naming convention for the iterating methods, namely `Init`, `HasMore`, `Next`, and `Current`, we can use the iterators in a general way in templated functions.

We'll use a simple example to illustrate how a templated function can count the number of elements in any iterator. We're not using this example because it's powerful, but it's the first step towards a very powerful technique of writing and using templated functions and classes.

The function `CountIter` in Program 11.6, *countiter.cpp*, counts the number of elements in an iterator. We'll call the function in two ways: to count the number of words in a file and to calculate $n!$ by generating every permutation of an n -element vector and counting the number of permutations. Counting words in a file this way is reasonably efficient; calculating $n!$ is not efficient at all.

Program 11.6 countiter.cpp

```
#include <iostream>
#include <string>
```

540 Chapter 11 Sorting, Templates, and Generic Programming

```

using namespace std;

#include "permuter.h"
#include "worditer.h"
#include "tvector.h"
#include "prompt.h"

template <class Type>
int CountIter(Type& it)
{
    int count = 0;
    for(it.Init(); it.HasMore(); it.Next())
    {
        count++;
    }
    return count;
}

int main()
{
    string filename = PromptString("filename: ");
    int k,num = PromptRange("factorial: ",1,8);
    tvector<int> vec;
    WordStreamIterator witer;

    for(k=0; k < num; k++)
    {
        vec.push_back(k);
    }
    witer.Open(filename);

    cout << "# words = " << CountIter(witer) << endl;
    cout << num << " factorial = " << CountIter(Permuter(vec)) << endl;

    return 0;
}

```

countiter.cpp

O U T P U T

```

prompt> countiter
filename: poe.txt
factorial: between 1 and 8: 6
# words = 2324
6 factorial = 720

```

The parameter `it` used in the function `CountIter` must conform to the iterator interface we use in this book. The variable `it` is used as an object that supports methods `Init`, `HasMore`, and `Next`. Since the function `Current` isn't used in `CountIter`, we could pass an object to `CountIter` that has a method named `GetCurrent` instead of `Current`. Since there is no call `it.Current()`, `Current` is not part of the

interface that the compiler expects to find when processing an argument passed in a call to `CountIter`.²

The method `Current` is used in `UniqueStrings` in Program 11.7 to count the number of unique strings in an iterator. We use it to determine the number of different strings in a file, and in a list constructed from the words in the file.

Program 11.7 `uniqueiter.cpp`

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

#include "clist.h"
#include "worditer.h"
#include "stringset.h"
#include "prompt.h"

template <class Type>
int UniqueStrings(Type& iter)
// post: return # unique strings in iter
{
    StringSet uni;
    for(iter.Init(); iter.HasMore(); iter.Next())
    {
        uni.insert(iter.Current());
    }
    return uni.size();
}

int main()
{
    string filename = PromptString("filename: ");
    WordStreamIterator witer;
    witer.Open(filename);
    CList<string> slist;

    for(witer.Init(); witer.HasMore(); witer.Next())
    {
        slist = cons(witer.Current(),slist);
    }

    cout << "unique from WordIterator = "
         << UniqueStrings(witer) << endl;
    cout << "unique from CList = "
         << UniqueStrings(CListIterator<string>(slist)) << endl;
    return 0;
}
```

`uniqueiter.cpp`

²The function `Permuter::Current` is a void function; it returns a vector as a reference parameter. It doesn't have the same interface as other `Current` functions which aren't void, but return a value.

```


O U T P U T



```

prompt> uniqueiter.cpp
filename: poe.txt
unique from WordIterator = 1039
unique from CList = 1039

prompt> uniqueiter.cpp
filename: hamlet.txt
unique from WordIterator = 7807
unique from CList = 7807

```


```

Although only standard iterator methods are used with parameter `iter`, the object returned by `iter.Current()` is inserted into a `StringSet`. This means that any iterator passed to `UniqueStrings` must conform to the expected interface of returning a string value from `Current`. For example, if we try to pass an iterator in which `Current` returns an `int`, as in the call below that uses a `CListIterator` for an `int` list, an error message will be generated by the compiler when it tries to instantiate the templated function.

```

cout << UniqueStrings(
    CListIterator<int>(CList<int>::EMPTY)
) << endl;

```

The error message generated by Visual C++ 6.0 is reasonably informative in telling us where to look for a problem.

```

itertempdemo.cpp(16) : error C2664: 'insert' :
cannot convert parameter 1 from 'int' to
'const class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > &'

```

The call to `insert` fails and the error message says something about converting an `int` to something related to a “`basic_string`”. The `basic_string` class is used to implement the standard class `string`; the class `basic_string` is templated to make it simpler to change from an underlying use of `char` to a type that uses Unicode, for example. This is a great idea in practice, but leads to error messages that are very difficult to understand if you don't know that identifier `string` is actually a typedef for a complicated templated class.³

³The actual typedef for `string` is `typedef basic_string<char, char_traits<char>, allocator<char> > string;` which isn't worth trying to understand completely.

ProgramTip 11.3: The name `string` is actually a typedef for a templated class called `basic_string`, and the template instantiation is somewhat complicated. If you see an error message that you can't understand, generated by either the compiler or a debugger, look carefully to see if the error is about the class `basic_string`; such errors are almost always caused by a problem with `string` objects or functions.

The function templates we've seen for sorting, printing, and iterating are powerful because they generalize an interface. Using function templates we can write functions that use an interface without knowing what class will actually be used to satisfy the interface. We'll explore an important use of templated functions in the Section 11.3 where we'll see how it's possible to sort by different criteria with one function.

11.2.3 Function Templates, Reuse, and Code Bloat

Function templates help avoid duplicated code since a templated function is written once, but can be instantiated many times. The template function `Print` in *sortwlen.cpp*, Program 11.5 is not compiled into object code as other functions and classes are when the program is compiled. Instead, when `Print` is instantiated, the compiler generates object code for the specific instantiation. If a templated function is never called/instantiated in a program, then it doesn't generate any code. A nontemplated function is always compiled into code regardless of whether it's called. The word *template* makes sense here; a templated function works as a function-generator, generating different versions of the function when the template is instantiated. However, different code is generated each time the function is instantiated with a different type. For example, if the templated `Print` function is called from three different statements using `int` vectors, two statements using `string` vectors, and one statement using `Date` vectors, then three different functions will be instantiated and emit compiled object code: one each for `int`, `string`, and `Date`.

ProgramTip 11.4: Function templates can generate larger than expected object files when they're instantiated with several types in the same program. A function template saves programmer resources and time since one function is written and maintained rather than several. However, a function template can lead to **code bloat** where a program is very large because of the number of template instantiations. Smart compilers may be able to do some code sharing, but code bloat can be a real problem.

11.3 Function Objects

In the program *sortwlen.cpp*, Program 11.5 we used a templated function `SelectSort` to sort vectors of strings and ints. The vector of strings was sorted alphabetically.

Suppose we need to generate a list of words in order by length of word, with shortest words like “a” coming first, and longer words like “acknowledgement” coming last. We can certainly do this by changing the comparison of vector elements to use a string’s length. In the function `SelectSort`, for example, we would change the comparison:

```
if (a[j] < a[minIndex])
```

to a comparison using string lengths.

```
if (a[j].length() < a[minIndex].length())
```

This solution does not generalize to other sorting methods. We may want to sort in reverse order, “zebra” before “aardvark”; or to ignore case when sorting so that “Zebra” comes after “aardvark” instead of before it as it does when ASCII values are used to compare letters. We can, of course, implement any of these sorts by modifying the code, but in general we don’t want to modify existing code, we want to re-use and extend it.

Program Tip 11.5: Classes, functions, and code should be open for extension, but closed to modification. This is called the *open-closed principle*.

This design heuristic will be easier to realize when we’ve studied templates and inheritance (see Chapter 13.) Ideally we want to adapt programs without breaking existing applications, so modifying code isn’t a good idea if it can be avoided.

In all these different sorts, we want to change the method used for comparing vector elements. Ideally we’d like to make the comparison method a parameter to the sort functions so that we can pass different comparisons to sort by different criteria. We’ve already discussed how vector elements must conform to the expected interface of being comparable using the relational operator `<` if the sorting functions in `sortall.h` are used. We need to extend the conforming interface in some way so that in addition to using operator `<`, a parameter is used to compare elements. Since all objects we’ve passed are instances of classes or built-in types, we need to encapsulate a comparison function in a class, and pass an object that’s an instance of a class. A class that encapsulates a function is called a **function object** or a **functor**. We’ll use functors to sort on several criteria.⁴

11.3.1 The Function Object Comparer

In the language C, there is no class `string`. Instead, special arrays of characters are used to represent strings. We won’t study these kinds of strings, but we’ll use the same convention in creating a class to compare objects that’s used in C to compare strings. Just as certain method names are expected with any iterators used in this book (by convention)

⁴In this section we’ll use classes with a function named `compare` to sort. In more advanced uses of C++, functors use an overloaded `operator()` so that an object can be used syntactically like a function, (e.g., `foo(x)` might be an object named `foo` with an overloaded `operator()` applied to `x`). My experience is that using an overloaded function application operator is hard for beginning programmers to understand, so I’ll use named functions like `compare` instead.

and with iterators used by templated functions like `UniqueStrings` in Program 11.7, *uniqueiter.cpp* (enforced by the compiler), we'll expect any class that encapsulates a comparison function to use the name `compare` for the function. We'll use this name in writing sorting functions and we'll expect functions with the name to conform to specific behavior. If a client uses a class with a name other than `compare`, the program will not compile, because the templated sorting function will fail to be instantiated. However, if a client uses the name `compare`, but doesn't adhere to the behavior convention we'll discuss, the program will compile and run, but the vector that results from calling a sort with such a function object will most likely not be in the order the client expects. Using a conforming interface is an example of **generic programming** which [Aus98] says is a "set of requirements on data types."

The sorting functions expect the conforming interface of `StrLenComp::compare` below. The method is `const` since no state is modified — there is no state.

```
class StrLenComp
{
public:
    int compare(const string& a, const string& b) const
    // post: return -1/+1/0 as a.length() < b.length()
    {
        if (a.length() < b.length()) return -1;
        if (a.length() > b.length()) return 1;
        return 0;
    }
};
```

The conforming interface is illustrated by the function prototype: it is a `const` function, returns an `int`, and expects two `const`-reference parameters that have the same type (which is `string` in the example above). The expected behavior is based on determining how `a` compares to `b`. Any function object used with the sorts in this book must have the following behavior.

- if `a` is less than `b` then `-1` is returned
- if `a` is greater than `b` then `+1` is returned
- otherwise, `a == b` and `0` is returned.

As shown in `StrLenComp::compare`, the meaning of "less than" is completely determined by returning `-1`, similarly for "greater than" and a return value of `+1`. Program 11.8 shows how this function object sorts by word length.

Program 11.8 `strlensort.cpp`

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

```

#include "tvector.h"
#include "sortall.h"
#include "prompt.h"

class StrLenComp
{
public:
    int compare(const string& a, const string& b) const
        // post: return -1/+1/0 as a.length() < b.length()
        {
            if (a.length() < b.length()) return -1;
            if (a.length() > b.length()) return 1;
            return 0;
        }
};

int main()
{
    string word, filename = PromptString("filename: ");
    tvector<string> wvec;
    StrLenComp slencomp;
    int k;
    ifstream input(filename.c_str());

    while (input >> word)
    {
        wvec.push_back(word);
    }
    InsertSort(wvec, wvec.size(), slencomp);

    for(k=0; k < 5; k++)
    {
        cout << wvec[k] << endl;
    }
    cout << "----" << endl << "last words" << endl;
    cout << "----" << endl;
    for(k=wvec.size()-5; k < wvec.size(); k++)
    {
        cout << wvec[k] << endl;
    }
    return 0;
}

```

strlensort.cpp

The sorts declared in *sortall.h* and implemented in *sortall.cpp* have two forms: one that expects a comparison function object as the third parameter and one that uses operator < so doesn't require the third parameter. The headers for the two versions of `InsertSort` are reproduced below.

```

template <class Type>
void InsertSort(tvector<Type> & a, int size);
// post: a[0] <= a[1] <= ... <= a[size-1]

template <class Type, class Comparer>
void InsertSort(tvector<Type> & a, int size,
               const Comparer & comp);
// post: first size entries sorted by criteria in comp

```

The third parameter to the function has a type specified by the second template parameter `Comparer`. Any object can be passed as the third parameter if it has a method named `compare`. In the code from Program 11.8 the type `StrLenComp` is bound to the type `Comparer` when the templated function `InsertSort` is instantiated.

O U T P U T

```
prompt> strlensort
filename: twain.txt
a
I
I
I
a
-----
last words
-----
shoulder--so--at
discouraged-like,
indifferent-like,
shoulders--so--like
"One--two--three-git!"
```

As another example, suppose we want to sort a vector of stocks, where the struct `Stock` from *stocks.cpp*, Program 8.6 is used to store stock information (see *stock.h* in on-line materials or Program 8.6 for details, or Program 11.9 below). We might want to sort by the symbol of the stock, the price of the stock, or the volume of shares traded. If we were the implementers of the class we could overload the relational operator `<` for `Stock` objects, but not in three different ways. In many cases, we'll be client programmers, using classes we purchase "off-the-shelf" for creating software. We won't have access to implementations so using function objects provides a good solution to the problem of sorting a class whose implementation we cannot access, and sorting by more than one criteria. In *sortstocks.cpp*, Program 11.9, we sort a vector of stocks by two different criteria: price and shares traded. We've used a struct for the comparer objects, but a class in which the `compare` function is public works just as well.

Program 11.9 *sortstocks.cpp*

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

```
#include "tvector.h"
#include "strutils.h" // for atoi and atof
#include "prompt.h"
#include "sortall.h"
#include "stock.h"

struct PriceComparer // compares using price
{
    int compare(const Stock& lhs, const Stock& rhs) const
    {
        if (lhs.price < rhs.price) return -1;
        if (lhs.price > rhs.price) return +1;
        return 0;
    }
};

struct VolumeComparer // compares using volume of shares traded
{
    int compare(const Stock& lhs, const Stock& rhs) const
    {
        if (lhs.shares < rhs.shares) return -1;
        if (lhs.shares > rhs.shares) return +1;
        return 0;
    }
};

void Read(tvector<Stock>& list, const string& filename)
// post: stocks from filename read into list
{
    ifstream input(filename.c_str());
    string symbol, exchange, price, shares;
    while (input >> symbol >> exchange >> price >> shares)
    {
        list.push_back(Stock(symbol,exchange,atof(price),atoi(shares)));
    }
}

Print(const tvector<Stock>& list, ostream& out)
// post: stocks in list printed to out, one per line
{
    int k,len = list.size();
    out.precision(3); // show 3 decimal places
    out.setf(ios::fixed);
    for(k=0; k < len; k++)
    {
        out << list[k].symbol << "\t" << list[k].exchange << "\t"
            << setw(8) << list[k].price << "\t" << setw(12)
            << list[k].shares << endl;
    }
}

int main()
{
    string filesymbol = PromptString("stock file ");
    tvector<Stock> stocks;
    Read(stocks,filesymbol);
    Print(stocks,cout);
    cout << endl << "---" << endl << "# stocks: " << stocks.size() << endl;
    cout << "--sorted by price--" << endl;
    InsertSort(stocks,stocks.size(), PriceComparer());
}
```

```

Print(stocks,cout);
cout << "--sorted by volume--" << endl;
InsertSort(stocks,stocks.size(), VolumeComparer());
Print(stocks,cout);
return 0;
}

```

 sortstocks.cpp

O U T P U T

```

prompt> sortstocks
filename: stocksmall.dat
KO      N      50.500      735000
DIS     N      64.125      282200
ABPCA   T       5.688       49700
NSCP    T      42.813      385900
F       N      32.125      798900

----
# stocks: 5
----sorted by price----
ABPCA   T       5.688       49700
F       N      32.125      798900
NSCP    T      42.813      385900
KO      N      50.500      735000
DIS     N      64.125      282200
----sorted by volume----
ABPCA   T       5.688       49700
DIS     N      64.125      282200
NSCP    T      42.813      385900
KO      N      50.500      735000
F       N      32.125      798900

```

11.3.2 Predicate Function Objects

As a final example, we'll consider the problem of finding all the files in a directory that are larger than a size specified by the user or that were last modified recently, (e.g., within three days of today). We'll use a function templated on three different arguments: every entry (one template parameter) in an iterator (another template parameter) is checked and those entries that satisfy a criterion (the last template parameter) are stored in a vector.

```

template <class Iter, class Pred, class Kind>
void IterToVectorIf(Iter& it, const Pred& p, tvector<Kind>& list)
// post: all items in Iter that satisfy Pred are added to list
{
    for(it.Init(); it.HasMore(); it.Next())
    {
        if (p.Satisfies(it.Current()))
        {
            list.push_back(it.Current());
        }
    }
}

```

The parameter `it` is used as in iterator in the function body, so we could pass a `DirStream` object or a `CListIterator` object among the many kinds of iterators we've studied. Since `it` has type `Iter`, when the function is instantiated/called, the first argument should be an iterator type. The second parameter of the function, `p`, has type `Pred`. If you look at the function body, you'll see that the only use of `p` is in the `if` statement. The object passed as a second parameter to `p` must have a member function named `Satisfies` that returns a `bool` value for the template instantiation to work correctly. Finally, the third parameter to the function is a vector that stores `Kind` elements. Elements are stored in the vector by calling `push_back` with `it.Current()` as an argument. The vector passed as the third argument to `IterToVectorIf` must store the same type of object returned by the iterator passed as the first argument. I ran the program on May 16, 1999 which should help explain the output.

Program 11.10 `dirvecfun.cpp`

```

#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;
#include "directory.h"
#include "prompt.h"
#include "tvector.h"

// illustrates templated functions, function objects

// find large files
struct SizePred // satisfies if DirEntry::Size() >= size
{
    SizePred(int size)
        : mySize(size)
    {}
    bool Satisfies(const DirEntry& de) const
    {
        return de.Size() >= mySize;
    }
    int mySize;
};

```

```

// find recent files
struct DatePred    // satisfies if DirEntry::GetDate() >= date
{
    DatePred(const Date& d)
        : myDate(d)
    { }
    bool Satisfies(const DirEntry& de) const
    {
        return !de.IsDir() && de.GetDate() >= myDate;
    }
    Date myDate;
};

void Print(const tvector<DirEntry>& list)
// post: all entries in list printed, one per line
{
    int k;
    DirEntry de;
    for(k=0; k < list.size(); k++)
    {
        de = list[k];
        cout << setw(10) << de.Size() << "\t" << setw(12)
            << de.Name() << "\t" << de.GetDate() << endl;
    }
    cout << "-\n# entries = " << list.size() << endl;
};

template <class Iter, class Pred, class Kind>
void IterToVectorIf(Iter& it, const Pred& p, tvector<Kind>& list)
// post: all items in Iter that satisfy Pred are added to list
{
    for(it.Init(); it.HasMore(); it.Next())
    {
        if (p.Satisfies(it.Current()))
        {
            list.push_back(it.Current());
        }
    }
}

int main()
{
    Date today;
    string dirname = PromptString("directory ");
    int size = PromptRange("min file size",1,300000);
    int before = PromptRange("# days before today",0,300);

    DatePred datePred(today-before); // find files within before days of today
    SizePred sizePred(size); // find files larger than size
    DirStream dirs(dirname); // iterate over directory entries
    tvector<DirEntry> dirvec; // store satisfying entries here

    IterToVectorIf(dirs,datePred,dirvec);
    cout << "date satisfying" << endl << "-" << endl;
    Print(dirvec);

    dirvec.resize(0); // remove old entries
}

```

552

Chapter 11 Sorting, Templates, and Generic Programming

```

cout << endl << "size satisfying"<< endl << "-" << endl;
IterToVectorIf(dirs,sizePred,dirvec);
Print(dirvec);
return 0;
}

```

dirvecfun.cpp

O U T P U T

```

prompt> dirvecfun
directory c:\book\ed2\code
min file size between 1 and 300000: 50000
# days before today between 0 and 300: 0
date satisfying
---
      4267      directory.cpp   May 16 1999
      4814      directory.h     May 16 1999
      2251      dirvecfun.cpp    May 16 1999
      2316      nqueens.cpp      May 16 1999
---
# entries = 4

size satisfying
---
      99991      foot.exe       April 14 1999
      64408      mult.exe       March 10 1999
      53760      mult.opt      March 31 1999
      165658     tap.zip       April 21 1999
      111163     tcwdef.csm   April 14 1999
---
# entries = 5

```

The structs `SizePred` and `DatePred` are called **predicates** because they're used as boolean function objects. We use the method name `Satisfies` from a term from mathematical logic, but it makes sense that the predicate function object returns true for each `DirEntry` object satisfying the criteria specified by the class.

Program Tip 11.6: A function object specifies a parameterized policy.

Functions that implement algorithms like sorting, but allow function object parameters to specify *policy*, such as how to compare elements, are more general than functions that hardwire the policy in code.

Pause to Reflect



11.8 If the function `Print` from `sortwlen.cpp`, Program 11.5 is passed a vector of `DirEntry` objects as follows, the call to `Print` will fail.

```
tvector<DirEntry> dirvec;
// store values in dirvec
Print(dirvec, 0, dirvec.size()-1);
```

Why does this template instantiation fail? What can you do to make it succeed?

11.9 Show how to prompt the user for the name of a directory and call `CountIter` from `countiter.cpp`, Program 11.6 to count the number of files and subdirectories in the directory whose name the user enters.

11.10 Write a function object that can be used to sort strings without being sensitive to case, so that `"ZeBrA" == "zebra"` and so that `"Zebra" > "aardvark"`.

11.11 Write three function objects that could be used in sorting a vector of `DirEntry` objects ordered by three criteria: alphabetically, by name of file, in order of increasing size, or in order by the date the files were last modified (use `GetDate`).

11.12 Suppose you want to use `IterToVectorIf` to store every file and subdirectory accessed by a `DirStream` object into a vector. Write a predicate function object that always returns true so that every `DirEntry` object will be stored in the vector. (see Program 11.10, `dirvecfun.cpp`.)

11.13 Write a templated function that reverses the elements stored in a vector so that the first element is swapped with the last, the second element is swapped with the second to last, and so on (make sure you don't undo the swaps; stop when the vector is reversed). Do not use extra storage; swap the elements in place.

11.14 Write a function modeled after `IterToVecIf`, but with a different name: `IterToVecFilter`. The function stores every element accessed by an iterator in a vector, but the elements are filtered or changed first. The function could be used to read strings from a file, but convert the strings to lowercase. The code below could do this with the right class and function implementations.

```
string filename = PromptString("filename: ");
WordStreamIterator wstream;
tvector<string> words;
wstream.Open(filename);
LowerCaseConverter lcConverter;
IterToVecFilter(wstream, lcConverter, words);
// all entries in words are in lower case
```

11.4 Analyzing Sorts

Using function objects we can sort by different criteria, but what sorting algorithms should we use? In this section we'll discuss techniques for classifying algorithms in

general, and sorting algorithms in particular, as to how much time and memory the algorithms require.

We discussed several quadratic sorts in Section 11.1 and discussed selection sort and insertion sort in some detail. Which of these is the best sort? As with many questions about algorithms and programming decisions, the answer is, “It depends”⁵—on the size of the vector being sorted, on the type of each vector element, on how critical a fast sort is in a given program, and many other characteristics of the application in which sorting is used. You might, for example, compare different sorting algorithms by timing the sorts using a computer. The program *timequadsorts.cpp*, Program 11.4, uses the templated sorting functions from *sortall.h* Program G.14 (see How to G), to time three sorting algorithms. The graph in Figure 11.3 provides times for these sorts.

Although the timings are different, the curves have the same shape. The timings might also be different if selection sort were implemented differently; as by another programmer. However, the general shapes of the curves would not be different, since the shape is a fundamental property of the algorithm rather than of the computer being used, the compiler, or the coding details. The shape of the curve is called **quadratic**, because it is generated by curves of the family $y = ax^2$ (where a is a constant). To see (informally) why the shape is quadratic, we will count the number of comparisons between vector elements needed to sort an N -element vector. Vector elements are compared by the `if` statement in the inner `for` loop of function `SelectSort` (see Program 11.2.)

```
if (a[j] < a[minIndex])
{   minIndex = j;   // new smallest item, remember where
}
```

We’ll first consider a 10-element vector, then use these results to generalize to an N -element vector. The outer `for` loop (with k as the loop index) iterates nine times for a 10-element vector, because k has the values 0, 1, 2, . . . , 8. When $k = 0$, the inner loop iterates from $j = 1$ to $j < 10$, so the `if` statement is executed nine times. Since k is incremented by 1 each time, the `if` statement will be executed 45 times, since the inner loop iterates nine times, then eight times, and so on:

$$9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = \frac{9(10)}{2} = 45 \quad (11.1)$$

The sum is computed from a formula for the sum of the first N integers; the sum is $N(N + 1)/2$. To sort a 100-element vector, the number of comparisons needed is $99(100)/2 = 4,950$. Generalizing, to sort an N -element vector, the number of comparisons is calculated by summing the first $N - 1$ integers:

$$\frac{(N - 1)(N)}{2} = \frac{N^2 - N}{2} = \frac{N^2}{2} - \frac{N}{2} \quad (11.2)$$

This is a quadratic, which at least partially explains the shape of the curves in Figure 11.3. We can verify this analysis experimentally using a templated class `SortWrapper` (accessible in *sortbench.h*) that keeps track of how many times sorted elements are

⁵Note that the right answer is *never* bubble sort.

compared and assigned. We've discussed comparisons; assignments arise when vector elements are swapped. The class `SortWrapper` is used in Program 11.11.

Program 11.11 `checkselect.cpp`

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
#include "sortbench.h"
#include "sortall.h"
#include "prompt.h"

int main()
{
    typedef SortWrapper<string> Wstring;
    string word, filename = PromptString("filename:");
    ifstream input(filename.c_str());

    tvector<Wstring>list;
    while (input >> word)
    { list.push_back(Wstring(word));
    }
    cout << "# words read =\t " << list.size() << endl;

    Wstring::clear(); // clear push_back assigns
    SelectSort(list,list.size());
    cout << "# compares\t = " << Wstring::compareCount() << endl;
    cout << "# assigns\t = " << Wstring::assignCount() << endl;
    return 0;
}
```

`checkselect.cpp`

O U T P U T

```
prompt> checkselect
filename: poe.txt
# words read =    2324
# compares       = 2699326
# assigns        = 4646
```

The number of comparisons is $2323 \times 2324/2$ which matches the formula in Equation 11.2 exactly. The number of assignments is exactly $2(N - 1)$ for an N -element vector because a swap requires two assignments and one construction, for example, for strings:

```

void Swap(string& x, string& y)
{
    string temp = x;    // construction, not assignment
    x = y;              // assignment
    y = temp;          // assignment
}

```

In general, when there are N elements there will be $N - 1$ swaps and $3(N - 1)$ data movements (assignments and constructions). Before analyzing other sorts, we need to develop some terminology to make the discussion simpler.

11.4.1 O Notation

When the execution time of an algorithm can be described by a family of curves, computer scientists use **O notation** to describe the general shape of the curves. For a quadratic family, the expression used is $O(N^2)$. It is useful to think of the O as standing for **order**, since the general shape of a curve provides an approximation *on the order of* the expression rather than an exact analysis. For example, the number of comparisons used by selection sort is $O(N^2)$, but more precisely is $(N^2/2) - (N/2)$. Since we are interested in the general shape rather than the precise curve, coefficients like 13.5 and lower-order terms with smaller exponents like N , which don't affect the general shape of a quadratic curve, are not used in O notation.

In later courses you may learn a formal definition that involves calculating limits, but the idea of a family of curves defined by the general shape of a curve is enough for our purposes. To differentiate between other notations for analyzing algorithms, the term **big-Oh** is used for O notation (to differentiate from little-oh, for example).

Algorithms like sequential search (Table 8.1) that are linear are described as $O(N)$ algorithms using big-Oh notation. This indicates, for example, that to search a vector of N elements requires examining nearly all the elements. Again, this describes the shape of the curve, not the precise timing, which will differ depending on the compiler, the computer, and the coding. Binary search, which requires far fewer comparisons than sequential search, is an $O(\log N)$ algorithm, as discussed in Section 8.3.7.

Table 11.2 provides data for comparing the running times of algorithms whose running times or **complexities** are given by different big-Oh expressions. The data are for a (hypothetical) computer that executes one million operations per second.

11.4.2 Worst Case and Average Case

When we use O -notation, we're trying to classify an algorithm's running time, or sometimes the amount of memory it uses. Some algorithms behave differently depending on the input. For example, when searching sequentially for an element in a vector, we might find the element in the first location, in the middle location, or we might not find it after examining all locations. How can we classify sequential search when the behavior is different depending on the item searched for? Typically, computer scientists use two methods to analyze an algorithm: **worst case** and **average case**. The worst case analysis is based on inputs to an algorithm that take the most time or use the most memory. In

Table 11.2 Comparing big-Oh expressions on a computer that executes one million instructions per second

N	Running time (seconds)			
	$O(\log N)$	$O(N)$	$O(N \log N)$	$O(N^2)$
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hours
1,000,000	0.000020	1.0	19.9	11.6 days
1,000,000,000	0.000030	16.7 min	8.3 hours	318 centuries

sequential search, for example, the worst case occurs when the element searched for isn't found; every vector element is examined. It's more difficult to define average case, and if you continue your studies of computer science you'll encounter different ways of defining average. In this book I'll use average case very informally, to mean what happens with most kinds of input, not the worst and not the best. To get an idea of what average case means we'll consider sequential search again. In an N -element vector there are $N + 1$ different ways for a sequential search algorithm to terminate:

- The item searched for is found in one of N different locations.
- The item searched for is not found.

If we look at the total number of vector items examined for every possible case when the search is successful we'll be able to apply Equation 11.2 again to get the total number of comparisons as $N(N + 1)/2$. Since there are N different ways to terminate successfully, we can argue that the average number of elements examined is

$$\frac{N(N + 1)/2}{N} = \frac{(N + 1)}{2} \quad (11.3)$$

This is still $O(N)$, so sequential search is $O(N)$ in both the worst and average case.

11.4.3 Analyzing Insertion Sort

The code for insertion sort in `insertsort.cpp`, Program 11.3 shows that the outer for loop executes $N - 1$ times for an N -element vector since k varies from 1 to $N - 1$. The number of times the inner while loop executes depends on the order of the elements and the value of `loc` after the while loop as shown in Figure 11.2. In the worst case, the vector is in reverse order and the inner loop will execute k times. The total number of times the body of the inner while loop executes is $(N - 1)N/2$ using Equation 11.2 since we're summing the first $N - 1$ numbers. There is one assignment each time the inner loop executes, and one assignment after the loop. The total number of assignments is $(N - 1)N/2 + N$ which is $O(N^2)$ and exactly $N(N + 1)/2$. There is one vector

comparison each time the inner loop executes and one comparison of $0 < \text{loc}$. We'll count only the vector comparison since although the comparison to see that the index loc is valid affects the execution time, it is independent of the kind of element being sorted. There are a total then of $(N - 1)N/2$ comparisons in the worst case.

In the best case, when the vector is already sorted, the inner loop body is never executed. There will be $O(N)$ comparisons and $O(N)$ assignments, which is about as good as we can expect since we have to examine every vector element simply to determine if the vector is sorted.

We can argue informally that on average the inner loop executes $k/2$ times since the worst case is k and the best case is zero. The algorithm is still an $O(N^2)$ algorithm, but the number of comparisons will be fewer than selection sort. This is why the timings in Figure 11.3 show insertion sort as faster than selection sort—it won't be faster always, but on average it is. We can verify some of these results experimentally with Program 11.12, *checkinsert.cpp*.

Program 11.12 *checkinsert.cpp*

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

#include "sortbench.h"
#include "sortall.h"
#include "prompt.h"
#include "tvector.h"

typedef SortWrapper<string> Wstring;

struct ReverseComparer // for sorting in reverse alphabetical order
{
    int compare(const Wstring& lhs, const Wstring& rhs) const
    {
        if (lhs < rhs) return +1;
        if (rhs < lhs) return -1;
        return 0;
    }
};

int main()
{
    string word, filename = PromptString("filename:");
    ifstream input(filename.c_str());
    tvector<Wstring>list;

    while (input >> word)
    {
        list.push_back(Wstring(word));
    }
    cout << "# words read =\t " << list.size() << endl;

    Wstring::clear(); // clear push_back assigns
```

```

InsertSort(list,list.size());
cout << "# compares\t = " << Wstring::compareCount() << endl;
cout << "# assigns\t = " << Wstring::assignCount() << endl;

Wstring::clear();
cout << endl << "sorting a sorted vector" << endl;
InsertSort(list,list.size());
cout << "# compares\t = " << Wstring::compareCount() << endl;
cout << "# assigns\t = " << Wstring::assignCount() << endl;

InsertSort(list,list.size(),ReverseComparer());
Wstring::clear();
cout << endl << "sorting a reverse-sorted vector" << endl;
InsertSort(list,list.size());
cout << "# compares\t = " << Wstring::compareCount() << endl;
cout << "# assigns\t = " << Wstring::assignCount() << endl;

return 0;
}

```

checkinsert.cpp

O U T P U T

```

prompt> checkinsert
filename: poet.txt
# words read = 2324
# compares = 1339264
# assigns = 1339269

sorting a sorted vector
# compares = 2323
# assigns = 2323

sorting a reverse-sorted vector
# compares = 2673287
# assigns = 2674325

```

11.5 Quicksort

The graph in Figure 11.3 suggests that selection sort and bubble sort are both $O(N^2)$ sorts.⁶ In this section we'll study a more efficient sort called **quicksort**. Quicksort is a recursive, three-step process.

⁶To be precise, the graph does not prove that bubble sort is an $O(N^2)$ sort; it provides evidence of this. To prove it more formally would require analyzing the number of comparisons.

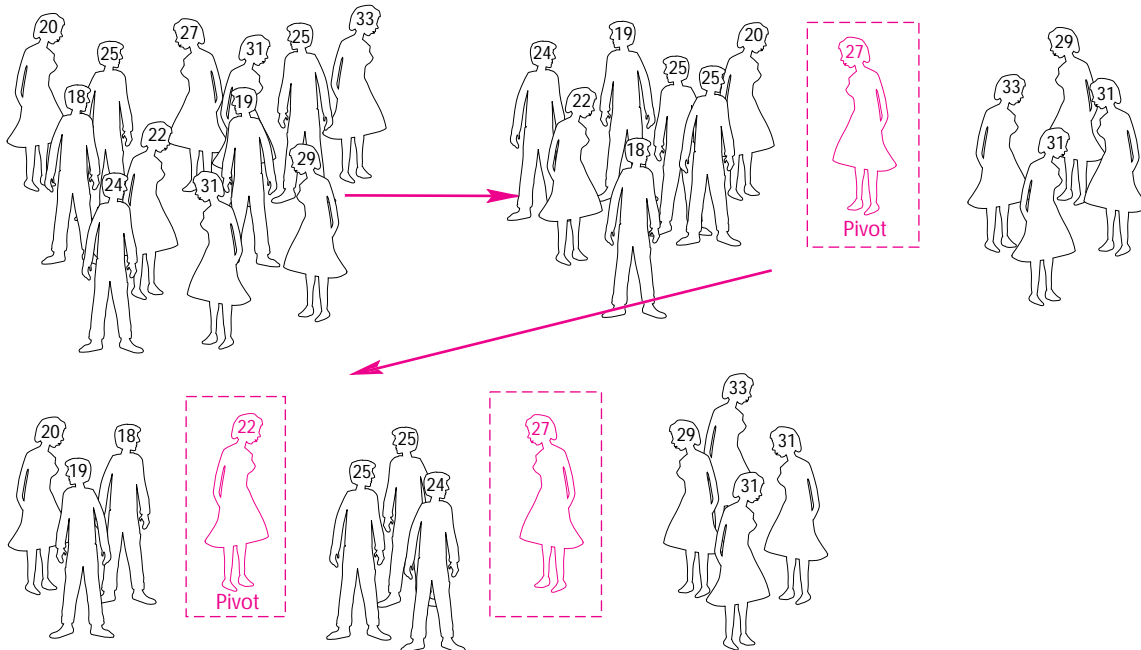


Figure 11.4 Quicksort.

1. A pivot element of the vector being sorted is chosen. Elements of the vector are rearranged so that elements less than or equal to the pivot are moved before the pivot. Elements greater than the pivot are moved after the pivot. This is called the **partition** step.
2. Quicksort (recursively) the elements before the pivot.
3. Quicksort (recursively) the elements after the pivot.

The partition step bears an explanation; we'll discuss the algorithm pictured in Figure 11.4.

Suppose a group of people must arrange themselves in order by age, so that the people are lined up, with the youngest person to the left and the oldest person to the right. One person is designated as the pivot person. All people younger than the pivot person stand to left of the pivot person and all people older than the pivot person stand to the right of the pivot. In the first step, the 27-year-old woman is designated as the pivot. All younger people move to the pivot's left (from our point of view); all older people move to the pivot's right. It is imperative to note at this point that the 27-year-old woman *will not move again!* In general, after the rearrangement takes place, the pivot person (or vector element) is in the correct order relative to the other people (elements). Also, people to the left of the pivot always stay to the left.

After this rearrangement, a recursive step takes place. The people to the left of the 27-year-old pivot must now sort themselves. Once again, the first step is to partition the group of seven people. A pivot is chosen—in this case, the 22-year-old woman. All people younger move to the pivot's left, and all people older move to the pivot's right. The group that moves to the right (two 25-year-olds and a 24-year-old) are now located between the two people who are in their final positions. To continue the process, the group of three (20, 18, and 19 years old) would sort themselves. When this group is done, the group of 25-, 25-, and 24-year-olds would sort themselves. At this point, the entire group to the left of the original 27-year-old pivot is sorted. Now the group to the right of this pivot must be recursively sorted.

The code for quicksort is very short and reflects the three steps outlined above: partition and recurse twice. Since the recursive calls specify a range in the original vector, we'll use a function with parameters for the left and right indexes of the part of the vector being sorted. For example, to sort an n -element `int` vector `a`, the call `Quick(a, 0, n-1)` works, where `Quick` is

```
void Quick(tvector<int>& a,int first,int last)
// postcondition: a[first] <= ... <= a[last]
{
    int piv;
    if (first < last)
    {
        piv = Pivot(a,first,last);
        Quick(a,first,piv-1);
        Quick(a,piv+1,last);
    }
}
```

The three statements in the `if` block correspond to the three parts of quicksort. The function `Pivot` rearranges the elements of `a` between positions `first` and `last` and returns the index of the pivot element. This index is then used recursively to sort the elements to the left of the pivot (in the range `[first ... piv-1]`) and the elements to the right of the pivot (in the range `[piv+1 ... last]`).

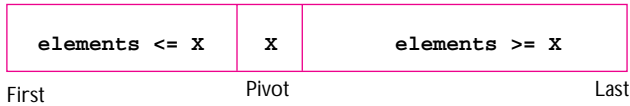
11.5.1 The Partition/Pivot Function

There are many different ways to implement the partition function. All these methods are linear, or $O(N)$, where N is the number of elements rearranged. We'll use a partition method described in [Ben86] that is simple to remember and that can be developed using invariants.

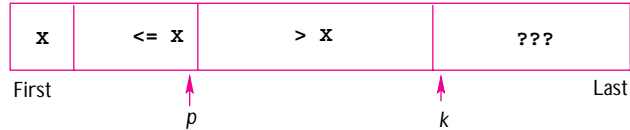
The diagrams in Figure 11.5 show the sequence of steps used in partitioning the elements of a vector between (and including) locations `first` and `last`. Understanding the second diagram in the sequence is the key to being able to reproduce the code. The second diagram describes an invariant of the `for` loop that partitions the `tvector`.

The `for` loop examines each vector element between locations `first` and `last` once; this ensures that the loop is linear, or $O(N)$, for partitioning N elements. The loop has the following form, where the element with index `first` is chosen as the pivot element:

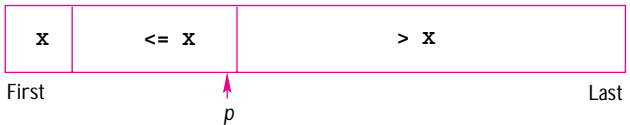
Desired properties of vector, partitioned around pivot



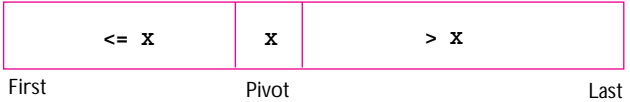
After several iterations, partially re-arranged, more elements to process



All elements processed



Final configuration after swapping first element into pivot location



```

}
if (a[k] <= piv)
{
    p++;
    Swap(a[k], a[p])
}

```

```
Swap(a[p], a[first]);
```

```
return p;
```

Figure 11.5 Partitioning for Quicksort.

```

for(k=first+1; k <= last; k++)
{
    if (a[k] <= a[first])
    {
        p++;
        swap(a[k], a[p])
    }
}

```

As indicated by the question marks “???” in Figure 11.5, the value of `a[k]` relative to the pivot is not known. If `a[k]` is less than or equal to the pivot, it belongs in the first part of the vector. If `a[k]` is greater than the pivot, it belongs in the second part of the vector (where it already is!). The `if` statement in Figure 11.5 compares `a[k]` to the pivot and then reestablishes the picture as true, so that the picture represents an invariant (true each time the loop test is evaluated).

This works because when `p` is incremented, it becomes the index of an element larger than the pivot, as shown in the diagram. The statement `Swap(a[k], a[p])` interchanges an element less than or equal to the pivot, `a[k]`, and an element greater than the pivot, `a[p]`. This informal reasoning should help convince you that the picture shown is an invariant and that it leads to a correct partition function. One more step is necessary, however; the invariant needs to be established as true the first time the loop test is evaluated. In this situation, the part of the vector labeled “???” represents the entire vector, because none of the elements have been examined.

The first element is arbitrarily chosen as the pivot element. Setting $k = \text{first} + 1$ makes k the index of the leftmost unknown element, the ??? section, as shown in the diagram. Setting $p = \text{first}$ makes p the index of the rightmost element that is known to be less than or equal to the pivot, because in this case only the element with index first is known to be less than or equal to the pivot—it is equal to the pivot, because it *is* the pivot.

The last step is to swap the pivot element, which is $a[\text{first}]$, into the location indexed by the variable p . This is shown in the final stage of the diagram in Figure 11.5.

The partition function, combined with the three-step recursive function for quicksort just outlined, yields a complete sorting routine that is included as part of *sortall.h*, Program G.14. We can change the call of `SelectSort` to `QuickSort` in *timequadsorts.cpp*, Program 11.4 and remove the call to `BubbleSort` to compare quicksort and insertion sort. We'll call the renamed program *timequicksort.cpp*, and won't show a listing since it doesn't change much from the original program.

```

                                O U T P U T
prompt> timequicksort
min and max size of vector: 6000 20000
increment in vector size between 1 and 10000: 2000

n          insert  quick
6000      1.792    0.03
8000      3.185    0.04
10000     5.177    0.05
12000     7.331    0.06
14000    10.075    0.07
16000    13.009    0.09
18000    16.604    0.101
20000    20.52     0.11

```

You can see from the sample runs that quicksort is *much* faster than insertion sort. If we extrapolate the data for insertion sort to a 300,000 element vector, we can approximate the time as 4660 seconds. The ratio $300,000/10,000 = 30$ shows that the execution time jumps by a factor of 900 from 10,000 to 300,000 since insertion sort is an $O(N^2)$ sort. Multiplying $5.177 \times 900 = 4659.3$, we determine that insertion sort takes a little more than 1 hour and 17 minutes to sort a 300,000 element vector. Removing the call to `InsertSort` so the program executes more quickly, we find that `QuickSort` takes 2.16 seconds to sort a 300,000 element vector. That's quick.

11.5.2 Analysis of Quicksort

With the limited analysis tools we have, a formal analysis of quicksort that provides a big-Oh expression of its running time is difficult. The choice of the pivot element in the partition step plays a crucial role in how well the method works. Suppose, for example, that in Figure 11.4 the first person chosen for the pivot is the 18-year-old person. All the people younger than this person move to the person's left; all the older people move to the person's right. In this case there are no younger people. This means that the two subgroups that would be sorted recursively are not the same size. If a "bad" partition continues to be chosen in the recursively sorted groups, quicksort degenerates into a slower, quadratic sort. On the other hand, if the pivot is chosen so that each subgroup is roughly the same size (i.e., half the size of the group being partitioned) then quicksort works very quickly.⁷

Since the partition algorithm is linear, or $O(N)$, for an N -element vector, the **computational complexity**, or running time, of quicksort depends on how many partitions are needed. In the best case the pivot element divides the vector being sorted into two equal parts. A more sophisticated analysis than we have the tools for shows that in the average case the vector is still divided approximately in half. If you examine the code for `Quick` below carefully, and assume that the value of `piv` is roughly in the middle, then you can reason about the sizes of the vector segments sorted with each pair of recursive calls.

```
void Quick(tvector<int>& a,int first,int last)
// postcondition: a[first] <= ... <= a[last]
{
    int piv;
    if (first < last)
    {   piv = Pivot(a,first,last);
        Quick(a,first,piv-1);
        Quick(a,piv+1,last);
    }
}
```

If `Quick` is first called with a 1000-element vector, a "good" pivot generates two recursive calls on 500-element vectors.⁸ The number of elements being sorted is $2 \times 500 = 1000$. Each of the 500-element vectors generates two recursive calls on 250-element vectors. Since there are two 500-element vectors, each generating two recursive calls, the number of elements being sorted is $4 \times 250 = 1000$. This continues with four 250-element vectors each generating two calls on 125-element vectors, but the total number of elements being sorted is still $8 \times 125 = 1000$. Every group of recursive calls yields a total of 1000 elements to sort, but the size of the vectors being sorted decreases. Eventually there will be 500 2-element vectors. Each of these will generate two recursive calls, but these recursive calls are the base case of a one- or zero-element vector. With

⁷It's not an accident that C.A.R. Hoare named the sort quicksort.

⁸A perfect partition will yield one 499-element vector and one 500-element vector since the pivot element doesn't move. We'll ignore this difference and treat each vector as a 500-element vector.

each group of recursive calls, there are 1000 elements to sort. For an N -element vector there will be N elements to partition and sort. Since we know that the partition code is $O(N)$, there is $O(N)$ work done at each recursive stage.

How many recursive stages are there? As we saw in Section 8.3.7 the number N can be divided in half approximately $\log_2 N$ times. Each group of recursive calls requires $O(N)$ work, and there are $\log_2 N$ groups of calls. This makes quicksort an $O(N \log N)$ algorithm. We ignore the base 2 on the log because $\log_b N / \log_2 N$ is a constant, for any value of b . Since we ignore constants in O -notation, we ignore the base of the log. However, in computer science nearly all uses of a logarithm function can be assumed to use a base 2 log. Quicksort is an $O(N \log N)$ algorithm in the average case, but not in the worst case where we've noted that it's an $O(N^2)$ algorithm. If we choose the first element as the pivot, then a sorted vector generates the worst case. It's possible to choose the partition in such a way that the worst case becomes extremely unlikely, but there are other sorts that are always $O(N \log N)$ even in the worst case. Nevertheless, quicksort is not hard to code, and its performance is extremely good in general. In the implementation of `QuickSort` in *sortall.cpp*, the median (or middle) of the first, middle, and last vector elements is chosen as the pivot. This makes `QuickSort` very fast except in degenerate cases that are unlikely in practice, though still possible. Implementations of two other $O(N \log N)$ sorts, `MergeSort` and `HeapSort`, are accessible from *sortall.h*. These sorts have good $O(N \log N)$ worst-case behavior, so if you must guarantee good performance, use one of them. Merge sort is particularly simple to implement for lists and we'll explore this in an exercise.

Pause to Reflect



- 11.15** Why are the average and worst cases of selection sort the same, whereas these cases are different for insertion sort?
- 11.16** In the output of *checkinsert.cpp*, Program 11.12, the worst case for insertion sort, sorting a vector that's in reverse order, yields 2,673,287 comparisons for a vector with 2,324 elements. However, $(2323 \times 2324)/2 = 2,699,326$. Explain this discrepancy (hint: are all the words in the vector unique?)
- 11.17** The timings for insertion sort are better than for selection sort in Figure 11.3. Selection sort will likely be better if strings are sorted rather than ints (int vectors were used in Figure 11.3.) If `DirEntry` objects are sorted the difference will be more pronounced, selection sort timings will not change much between ints, strings, and `DirEntry` objects, but insertion sort timings will get worse. What properties of the sorts and the objects being sorted could account for these observations?
- 11.18** If we sort a 100,000 element int vector using quicksort, where all the ints are in the range $[0 \dots 100]$, the sort will take a very long time. This is true because the partition diagrammed in Figure 11.5 cannot result in two equal parts of the vectors; the execution will be similar to what happens with quick sort when a bad pivot is chosen. If the range of numbers is changed to $[0 \dots 50,000]$ the performance gets better. Why?

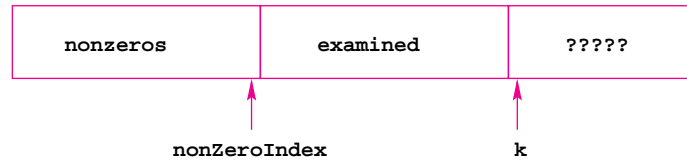


Figure 11.6 Removing zeroes from an array/vector.

11.19 The implementation of quicksort in *sortall.cpp* uses a different stopping criterion than the one described in Section 11.5. The code presented there made recursive calls until the size of the vector being sorted was one or zero; that is, the base case was determined by `if (first < last)`. Instead, the function below is used, where `CUTOFF` is 30.

```
void Quick(tvector<int>& a, int first, int last)
// postcondition: a[first] <= ... <= a[last]
{
    if (last - first > CUTOFF)
    {
        int piv = Pivot(a, first, last);
        Quick(a, first, piv-1);
        Quick(a, piv+1, last);
    }
    else
    {
        Insert(a, first, last); // call insertion sort
    }
}
```

This speeds up the execution of quicksort. Why?

11.20 Write a function that removes all the zeros from a vector of integers, leaving the relative order of the nonzero elements unchanged, without using an extra vector. The function should run in linear time, or have complexity $O(N)$ for an N -element vector.

```
void RemoveZeros(tvector<int> & list, int & numElts)
// postcondition: zeros removed from list,
//                numElts = # elts in list
```

If you're having trouble, use the picture in Figure 11.6 as an invariant. The idea is that the elements in the first part of the vector are nonzero elements. The elements in the section "???" have yet to be examined (the other elements have been examined and are either zeros or copies of elements moved into the first section.) If the k th element is zero, it is left alone. If it is nonzero, it must be moved into the first section.

11.21 After a vector of words read from a file is sorted, identical words are adjacent to each other. Write a function to remove copies of identical words, leaving only one occurrence of the words that occur more than once. The function should have complexity $O(N)$, where N is the number of words in the original vector (stored in the file). Don't use two loops. Use one loop and think carefully about the right invariant. Try to draw a picture similar to the one used in the previous exercise.

11.22 Binary search requires a sorted vector. The most efficient sorts are $O(N \log N)$, binary search is $O(\log N)$, and sequential search is $O(N)$. If you have to search an N element vector that's unsorted, when does it make sense to sort the vector and use binary search rather than to use sequential search?

11.6 Chapter Review

We discussed sorting, generic programming, templated functions, function objects, and algorithm analysis including O -notation. Two quadratic sorts: insertion sort and selection sort, are fast enough to use for moderately sized data. For larger data sets an $O(N \log N)$ sort like quicksort may be more appropriate. Functions that implement sorts are often implemented as templated functions so they can be used with vectors of any type, such as `int`, `string`, `double`, `Date`, and so on. A second template parameter can be used to specify a sorting policy, (e.g., to sort in reverse order or to ignore the case of words). This parameter is usually a function object: an object used like a function in code. Using big- O h expressions allows us to discuss algorithm efficiency without referring to specific computers. O -notation also lets us hide some details by ignoring low-order terms and constants so that $3N^2$, $4N^2 + 2N$ and N^2 are all $O(N^2)$ algorithms.

Topics covered include the following:

- Selection sort, an $O(n^2)$ sort that works fast on small-sized vectors (where small is relative).
- Insertion sort is another $O(n^2)$ sort that works well on nearly sorted data.
- Bubble sort is an $O(n^2)$ sort that should rarely be used. Its performance is much worse, in almost all situations, than that of selection sort or insertion sort.
- Overloaded functions permit the same name to be used for different functions if the parameter lists of the functions differ.
- Templated functions are used for functions that represent a pattern, or template, for constructing other functions. Templated functions are often used instead of overloading to minimize code duplication.
- Function objects encapsulate functions so that the functions can be passed as policy arguments; that is, so that clients can specify how to compare elements being sorted.
- O -notation, or big- O h, is used to analyze and compare different algorithms. O -notation provides a convenient way of comparing algorithms, as opposed to implementations of algorithms on particular computers.
- The sum of the first n numbers, $1 + 2 + \dots + n$, is $n(n + 1)/2$.

- Quicksort is a very fast sort, $O(n \log n)$ in the average case. In the worst case, quicksort is $O(n^2)$.

11.7 Exercises

11.1 Implement *bogosort* from Chapter 1 using a function that shuffles the elements of a vector until they're sorted. Test the function on n -element vectors (for small n) and graph the results showing average time to sort over several runs.

11.2 You may have seen the word game Jumble in your newspaper. In Jumble the letters in a word are mixed up, and the reader must try to guess what the word is (there are actually four words in a Jumble game, and a short phrase whose letters have to be obtained from the four words after they are solved). For example, *neicma* is *iceman*, and *cignah* is *aching*.

Jumbles are easy for computers to solve with access to a list of words. Two words are anagrams of each other if they contain the same letters. For example, *horse* and *shore* are anagrams.

Write a program that reads a file of words and finds all anagrams. You can modify this program to facilitate Jumble-solving. Use the declaration below to store a "Jumble word".

```
struct Jumble
{
    string word;           // regular word, "horse"
    string normal;       // sorted/normalized, "ehors"
    Jumble(const string& s); // constructor(s)
};
```

Each English word read from a file is stored along with a sorted version of the letters in the word in a `Jumble` struct. For example, store *horse* together with *ehors*. To find the English word corresponding to a jumbled word like *cignah*, sort the letters in the jumbled word yielding *acghin*, then look up the sorted word by comparing it to every `Jumble` word's `normal` field. It's easiest to overload operator `==` to compare `normal` fields, then you can write code like this:

```
string word;
cout << "enter word to de-jumble";
cin >> word;
Jumble jword(word);
// look up jword in a vector<Jumble>
```

A word with anagrams will have more than one `Jumble` solution. You should sort a vector of words by using the sorted word as the key, then use binary search when looking up the jumbled word. You can overload operator `<` for the struct `Jumble`, or pass a function object that compares the `normal` field of two `Jumble` objects when sorting.

You should write two programs, one to find all the anagrams in a file of words and one to allow a user to interactively search for Jumble solutions.

11.3 Write a program based on *dirvecfun.cpp*, Program 11.10. Replace `IterToVectorIf`

with a function `IterToListIf` that returns a `CList` object rather than a vector object.

- 11.4** Write a program based on *dirvecfun.cpp*, Program 11.10, specifically on the function `IterToVectorIf`, but specialized to the class `DirStream`. The program should allow the client to implement Predicate function objects and apply them to an entire directory hierarchy, not just to a top-level directory (see the run of the Program 11.10). The client should be able to specify a directory in a `DirStream` object and get back a vector of every file that matches some Predicate function object's `Satisfies` criteria that's contained in the specified directory or in any subdirectory reachable from the specified directory.

Users of the program should have the option of printing the returned files sorted by several criteria: date last modified, alphabetically, or size of file.

- 11.5** In Exercise 7 of Chapter 6 an algorithm was given for calculating the variance and standard deviation of a set of numbers. Other statistical measures include the **mean** or average, the **mode** or most frequently occurring value, and the **median** or middle value.

Write a class or function that finds these three statistical values for a `tvector` of `double` values. The median can be calculated by sorting the values and finding the middle value. If the number of values is even, the median value can be defined as either the average of the two values in the middle or the smaller of the two. Sorting the values can also help determine the mode, but you may decide to calculate the mode in some other manner.

- 11.6** The bubble sort algorithm sorts the elements in a vector by making N passes over a vector of N items. On each pass, adjacent elements are compared, and if the element on the left (smaller index) is greater it is swapped with its neighbor. In this manner the largest element “bubbles” to the end of the vector. On the next pass, adjacent elements are compared again, but the pass stops one short of the end. On each pass, bubbling stops one position earlier than the pass before until all the elements are sorted. The following code implements this idea.

```
template <class Type>
void BubbleSort(tvector<Type> & a, int n)
// precondition: n = # of elements in a
// postcondition: a is sorted
//           note: this is a dog of a sort
{
    int j,k;
    for(j=n-1; j > 0; j--)
    { // find largest element in 0..k, move to a[j]
        for(k=0; k < j; k++)
        { if (a[k+1] < a[k])
            { Swap(a[k],a[k+1]);
            }
        }
    }
}
```

Bubble sort can be “improved” by stopping if no values are swapped on some pass,⁹ meaning that the elements are in order. Add a `bool` flag variable to the preceding code so that the loops stop when no bubbling is necessary. Then time this function and compare it to the other $O(n^2)$ sorts: selection sort and insertion sort.

- 11.7** Write a function that implements insertion sort on `CList` objects. First test the function on lists of strings. When you’ve verified that it works, template the function and try it with lists of other types, e.g., `int`. Since a `CList` object cannot change, you’ll have to create a new sorted list from the original. The general idea is to insert one element at a time from the original list into a new list that’s kept sorted. The new list contains those elements moved from the original list processed so far. It’s easiest to implement the function recursively. You may also find it helpful to implement a helper function:

```
CList<string> addInOrder(const string& s,
                       CList<string>& list)
// pre: list is sorted
// post: return a new list, original with s added,
//       and the new list is sorted
```

Instrument the sort in a test program that prints the results from `CList::ConsCalls`. Graph the number of calls as a function of the size of the list being sorted.

- 11.8** Merge sort is another $O(N \log N)$ sort (like quicksort), although unlike quicksort, merge sort is $O(N \log N)$ in the worst case. The general algorithm for merge sort consists of two steps to sort a `CList` list of N items.

- Recursively sort the first half and the second half of the list. To do this you’ll need to create two half-lists: one that’s a copy of the first half of a `CList` and the other which is the second half of the `CList`. This means you’ll have to cons up a list of $N/2$ elements given an N element list. The other $N/2$ element list is just the second half of the original list.
- Merge the two sorted halves together. The key idea is that merging two sorted lists together, creating a sorted list, can be done efficiently in $O(N)$ time if both sorted lists have $O(N)$ elements. The two sorted lists are scanned from left to right, and the smaller element is copied into the list that’s the merge of the two.

Write two functions that together implement merge sort for `CList` lists.

```
CList<string>
merge(const CList<string>& a, const CList<string>& b);
// pre: a and b are sorted
// post: return a new list that’s sorted,
//       containing all elements from a and b

CList<string> mergesort(CList<string> list);
// post: return a sorted version of list using mergesort
```

⁹This improvement can make a difference for almost-sorted data, but it does not mitigate the generally atrocious performance of this sort.