

Arrays, Data, and Random Access

8

A teacher who can arouse a feeling for one single good action . . . , accomplishes more than he who fills our memory with rows on rows of natural objects, classified with name and form.

Goethe

Elective Affinities, Book II, Ch. 7

Computers are useless, they can only give you answers.

Pablo Picasso

21st Century Dictionary of Quotations

A compact disc (CD), a computer graphics monitor, and a group of campus mailboxes share a common characteristic, as shown in Figure 8.1: Each consists of a sequence of items, and each item is accessible independently of the other items. In the case of a CD, any track can be played without regard to whether the other tracks are played. This arrangement is different from the way songs are recorded on a cassette tape, where, for example, the fifth song is accessible only after playing or fast-forwarding past the first four. In the case of a graphics monitor, any individual picture element, or **pixel**, can be turned on or off, or changed to a different color, without concern as to what the values of the other pixels are. The independence of each pixel to display different colors permits images to be displayed very rapidly. The address of a student on many campuses, or a person living in an apartment building, is typically specified by a box number.

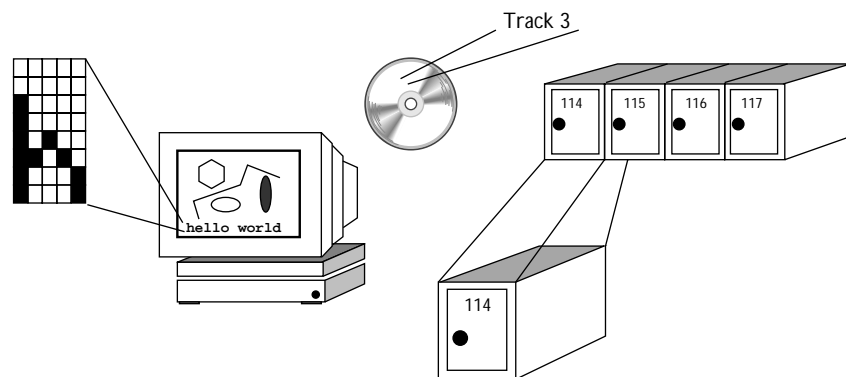


Figure 8.1 Random or constant-time access.

Postal workers can deliver letters to box 117 without worrying about the location of the first 100 boxes, the last 100 boxes, or any boxes other than 117.

This characteristic of instant access is useful in programming applications. The terminology used is **random access**, as opposed to the **sequential access** to a cassette tape. Most programming languages include a construct that allows data to be grouped together so that each data item is accessible independently of the other items. For example, a collection of numbers might represent test scores; a collection of strings could represent the different words in *Hamlet*; and a collection of strings and numbers combined in a struct might represent the words in *Hamlet* and how many times each word occurs.

We've studied three ways of structuring data in C++ programs: classes, structs, and files accessible using streams. In this chapter you will learn about a data structure called an **array**—one of the most useful data structures in programming. Examples of array use in this chapter include:

- Using an array as many counters, for example, to keep track of how many times all sums of rolling n -sided dice occur or to keep track of how many times each letter of the alphabet occurs in *Hamlet*.
- Using an array to store a list of words in a file, keeping track of each different word and then extending this array to track how many times each different word occurs.
- Using an array to maintain a database of on-line information for over 3,000 different CD titles, or alternatively, an on-line address book.

8.1 Arrays and Vectors as Counters

Consider Program 8.1, *dieroll.cpp*, which tracks the number of times each sum between 2 and 8 occurs when two four-sided dice are rolled. Modifying this program to track the number of times each possible dice roll occurs for six-sided dice would be very ugly.

Program 8.1 *dieroll.cpp*

```
#include <iostream>
using namespace std;
#include "dice.h"
#include "prompt.h"

// illustrates cumbersome programming
// roll two dice and track occurrences of all possible rolls

const int DICE_SIDES = 4;

int main()
{
    int twos=    0;           // counters for each possible roll
    int threes=  0;
    int fours=   0;
```

```
int fives= 0;
int sixes= 0;
int sevens= 0;
int eights= 0;

int rollCount = PromptRange("how many rolls",1,20000);
Dice d(DICE_SIDES);

int k;
for(k=0; k < rollCount; k++) // simulate all the rolls
{ int sum = d.Roll() + d.Roll();
  switch (sum)
  {
    case 2:
      twos++;
      break;
    case 3:
      threes++ ;
      break;
    case 4:
      fours++;
      break;
    case 5:
      fives++;
      break;
    case 6:
      sixes++;
      break;
    case 7:
      sevens++;
      break;
    case 8:
      eights++;
      break;
  }
}
// output for each possible roll # of times it occurred

cout << "roll\t# of occurrences" << endl;
cout << "2\t" << twos << endl;
cout << "3\t" << threes << endl;
cout << "4\t" << fours << endl;
cout << "5\t" << fives << endl;
cout << "6\t" << sixes << endl;
cout << "7\t" << sevens << endl;
cout << "8\t" << eights << endl;

return 0;
}
```

```

                                O U T P U T
prompt> dieroll
how many rolls between 1 and 20000  10000
roll          # of occurrences
2             623
3             1204
4             1935
5             2474
6             1894
7             1246
8             624

```

The code in *dieroll.cpp* would be much more compact if loops could be used to initialize the variables and generate the output. To do this we need a new kind of variable that maintains several different values at the same time; such a variable could be used in place of *twos*, *threes*, *fours*, and so on. Most programming languages support such variables; they are called **arrays**. An array structures data together, but has three important properties:

1. An array is a *homogeneous* collection. Each item stored in an array is the same type; for example, all integers, all `doubles`, or all strings. It is not possible to store both integers and strings in the same array.
2. Items in an array are numbered, or ordered; that is, there is a first item, a fifteenth item, and so on. The number that refers to an item is the item's **index**, sometimes called the **subscript**.
3. An array supports **random access**. The time to access the first item is the same as the time to access the fifteenth item or the hundredth item.

In C++ the built-in array type has many problems; it is difficult for beginning programmers to use and its use is too closely coupled with its low-level implementation.¹ We'll study built-in arrays, but we want to study the concept of homogeneous collections and random access without the hardships associated with using the built-in array type. Instead, we'll use a class that behaves like an array from a programming perspective but insulates us from the kind of programming problems that are common with built-in arrays. We'll use a class `tvector`, defined in the header file *tvector.h*.² The "t" in

¹The built-in array type in C++ is the same as its C-based counterpart. It is based on pointers, designed to be very efficient, and prone to hard-to-find errors, especially for beginning programmers.

²The class `vector` is defined as part of the STL library in standard C++. The class `tvector` declared in *tvector.h* is consistent with this standard class. The class `apvector`, defined for use in the Advanced Placement computer science course, is based on the class `tvector`. All member functions of the `apvector` class are also member functions of the `tvector` class. However, the `tvector` class supports `push_back` and `pop_back` functions not supported by `apvector`.

`tvector` stands for “Tapestry.” You can use the standard `vector` class in any of the programs in this book, but you’ll find the `tvector` class is much more forgiving of the kinds of mistakes that beginning and experienced programmers make. Because `tvector` catches some errors that `vector` doesn’t catch, `tvector` is slightly less efficient. If you really need the efficiency, develop using `tvector` and then switch to `vector` when you know your program works correctly.

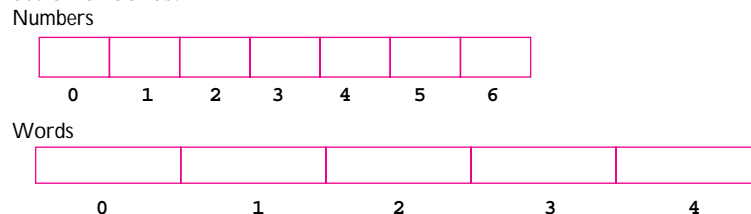
Before studying Program 8.2, a program that is similar to *dieroll.cpp* but uses a `tvector` to track dice rolls, we’ll discuss important properties of the `tvector` class and how to define `tvector` variables.

8.1.1 An Introduction to the Class `tvector`

The simplest definition of a `tvector` variable includes the variable’s name, the type of item being stored, and an integer value passed to the constructor that indicates how many items the vector can store. The definitions below define a variable `numbers` that can store seven integer values and a variable `words` that can store five string values.

```
tvector<int>    numbers(7);
tvector<string> words(5);
```

Because a `tvector` is a homogeneous collection, you can think of a `tvector` variable as a collection of boxes:



Each box or item in the `tvector` is referenced using a numerical index. In C++ the first item stored in a `tvector` has index zero. Thus, in the diagram here, the five items in `words` are indexed from zero to four. In general, the valid indices in a `tvector` with n elements are $0, 1, \dots, n - 1$.

An element of a `tvector` is selected, or referenced, using a numerical index and brackets: `[]`. The following statements store the number 13 as the first element of `numbers` and the string “fruitcake” as the first element of `words` (remember that the first element has index zero):

```
numbers[0] = 13;
words[0]   = "fruitcake";
```

`tvector` variables can be indexed using a loop as follows, where all the elements of `numbers` are assigned the value zero:

```
int k;
for(k=0; k < 5; k++)
{   numbers[k] = 0;
}
```

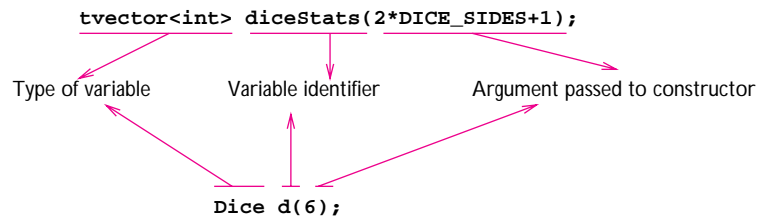


Figure 8.2 Comparing a `tvector` variable definition to a `Dice` variable definition. Both variables have names and a constructor parameter.

The number of elements in a vector variable is specified by a parameter to the `tvector` constructor, just as the number of sides of a `Dice` variable is specified when the `Dice` variable is constructed, as shown in Figure 8.2. This value can be a variable whose value is entered by the user; an expression; or, in general, any integer value. More details on defining `tvector` variables are given in Section 8.2.1.

8.1.2 Counting with `tvector`s

Program 8.2, *dieroll2.cpp* uses a `tvector` to keep track of different dice rolls but otherwise performs the same tasks as Program 8.1, *dieroll.cpp*.

From a black-box viewpoint there is no difference between the programs *dieroll.cpp* and *dieroll2.cpp*. The `tvector` variable `diceStats` can store nine different integer values. The capacity of `diceStats` is determined when the variable is defined by the statement `tvector<int> diceStats(2*DICE_SIDES+1)`.

Program 8.2 *dieroll2.cpp*

```
#include <iostream>
using namespace std;
#include "dice.h"
#include "prompt.h"
#include "tvector.h"

// use vector to simulate rolling of two dice
// Owen Astrachan, March 9, 1994, modified 5/2/99

const int DICE_SIDES = 4;

int main()
{
    int sum;
    int k;
    Dice d(DICE_SIDES);
    tvector<int> diceStats(2*DICE_SIDES+1); // room for largest dice sum
    int rollCount = PromptRange("how many rolls",1,20000);
```

```

for(k=2; k <= 2*DICE_SIDES; k++)          // initialize counters to zero
{   diceStats[k] = 0;
}

for(k=0; k < rollCount; k++)             // simulate all the rolls
{   sum = d.Roll() + d.Roll();
    diceStats[sum]++;
}

cout << "roll\t\t# of occurrences" << endl;
for(k=2; k <= 2*DICE_SIDES; k++)
{   cout << k << "\t\t" << diceStats[k] << endl;
}
return 0;
}

```

dieroll2.cpp

O U T P U T

```

prompt> dieroll2
how many rolls between 1 and 2000010000
roll          # of occurrences
2             623
3             1204
4             1935
5             2474
6             1894
7             1246
8             624

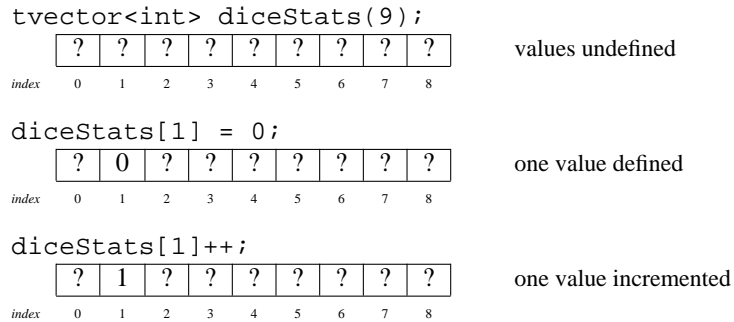
```

There is one major difference between the definition of `diceStats` as a `tvector` variable and that of `d` as a `Dice` variable: the `tvector` definition indicates that the `tvector` contains integers. We'll discuss this in depth after examining other parts of the program.

Because the indexing begins with 0, the last location in a nine-element array has index 8. This is why space for nine integer values is allocated in Program 8.2 even though only seven of the locations are accessed in the program—`diceStats[2]` through `diceStats[8]`—as shown in Figure 8.3. The conceptual simplicity of using `diceStats[sum]` to represent the number of times two dice are rolled more than compensates for the two memory locations that could be saved by defining an array of seven values and using `diceStats[sum-2]` to store the number of times `sum` is obtained.

In Figure 8.3 the `switch` statement used to increment the appropriate counter in Program 8.1 is contrasted with the single statement `diceStats[sum]++`, which increments the corresponding vector location serving as a counter in Program 8.2.

When a vector is defined, the values in each vector location, or **cell**, are initially undefined. The vector cells can be used as variables, but they must be indexed, as shown here for a vector named `diceStats` containing nine cells:



The indexing expression determines which of the many array locations is accessed. Indexing makes arrays extraordinarily useful. One array variable represents potentially thousands of different values, each value specified by the array variable name and the indexing value. The expression `diceStats[1]` is read as “diceStats sub one,” where the word “sub” comes from the mathematical concept of a *subscripted* variable such as n_1 .

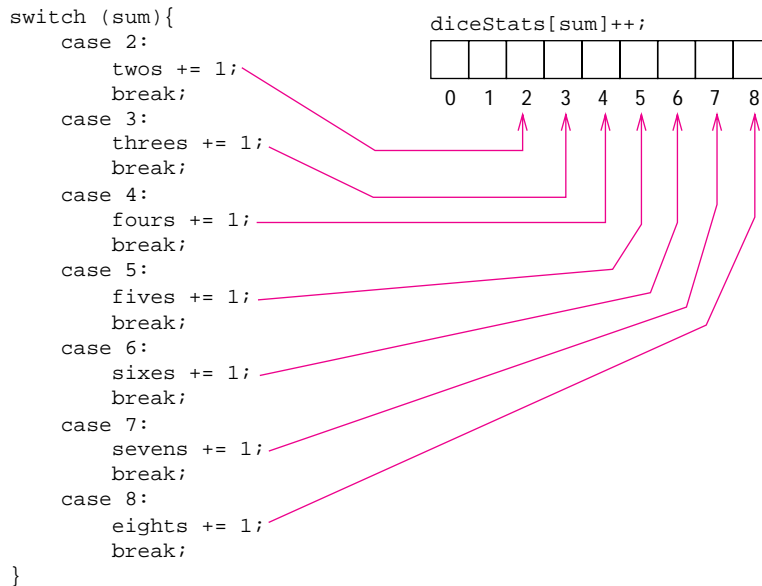


Figure 8.3 Using a `tvector` to store counts for tracking dice rolls.

8.2 Defining and Using tvector

8.2.1 tvector Definition

When you define a `tvector` variable you'll normally specify the number of entries, or cells, in the `tvector`. As we'll see later in this chapter, when you use a `vector` so that it grows to accommodate more cells, it's possible that you won't specify the number of cells when the vector is first constructed.

Since vectors are homogeneous collections, you must also specify the type stored in each entry, such as `int`, `string`, or `double`. The following statements define three `tvector` variables: `values` stores 200 doubles, `names` stores 50 strings, and `scores` which stores some number between 1 and 1000 of `int` values.

```
tvector<double> values(200);
tvector<string> names(50);
tvector<int> scores(PromptRange("# of scores",1,1000));
```

The type of value stored in each cell of a `tvector` variable is specified between angle brackets (the less-than and greater-than symbols) before the name of the variable is given. The size of the `tvector` is an argument to the constructor, as illustrated in Figure 8.2.

Syntax: tvector definition

```
tvector<type> varname;
tvector<type> varname(size expression);
tvector<type> varname(size expression,
                    value);
```

The type that defines what *kind* of element is stored in each array cell can be any built-in type (e.g., `int`, `double`, `bool`). It can also be a programmer-defined type such as `string`. The only qualification on programmer-defined types is that the type must have a default (or parameterless) constructor. For example, it is *not* possible to have a definition `tvector<Dice> dielist(10)` for an array of 10 dice elements, because a `Dice` object requires a parameter indicating the number of sides that the `Dice` object has. It is possible to define a vector of `Date` elements (see `date.h`, in *Howto G* or Program 5.10, `usedate.cpp`), because there is a default constructor for the `Date` class.

The *expression* in the `tvector` constructor determines the number of cells of the `tvector` variable. This integer expression can use variables, arithmetic operators, and function calls. For example, it is possible to use

```
tvector<int> primes(int(sqrt(X)));
```

to allocate a variable named `primes` whose number of cells is given by the (integer) truncated value of the square root of a variable `X`. If no integer expression is used, as in `tvector<int> list`, a vector with zero cells is created. We'll see later that sometimes this is necessary and that the number of cells in a vector can grow or shrink. The third form of constructor initializes all the cells to the value passed as the second argument to the constructor.



8.2.2 `tvector` Initialization

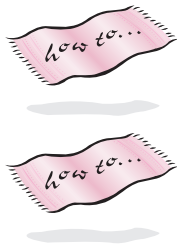
For vectors of user-defined types like `string`, all vector elements are initialized using the default constructor unless an argument is supplied the `tvector` constructor for initialization as shown below. For built-in types the `tvector` class does not initialize each vector cell, so the values will most likely be undefined.³ For example, when a variable is used to represent several counters, as it is in Program 8.2, *dieroll2.cpp*, each element of the vector must be initialized to zero. As shown in the syntax diagram for `tvector` construction, it's possible to initialize all elements of a vector when the vector is constructed. For example, the statements below create a vector of 20 strings in which each string has the value "Fred Brooks" and a vector of 100 ints in which each int has the value 25.

```
tvector<string> names(20, "Fred Brooks");
tvector<int>     nums(100, 25);
```

8.2.3 `tvector` Parameters

`tvector` variables can be passed as parameters just like any other variable in C++.⁴ To illustrate how vectors are used and passed as parameters, we'll study another example in which a vector is used to count several quantities. First we'll count how many times each different character occurs in an input file. For example, we can count the number of occurrences of the letter *e* in *Hamlet* using Program 8.3, *letters.cpp*. Just as a vector of counters was used to count dice rolls in *dieroll2.cpp*, Program 8.2, a `tvector` of counters is used to track how many times each character occurs in the file.

Counting characters is similar to counting dice rolls: each `tvector` element records the number of occurrences of one character. We mentioned the type `char` briefly in Section 3.2.4. You can find more information on the type `char` in Howto A and in Chapter 9. Two character processing functions are used in *letters.cpp* that we haven't seen before, but hopefully you'll understand their use from the context and the comments. The function `isalpha` from `<cctype>` (or `<cctype.h>`) returns `true` if its `char` parameter is a letter, either 'a'-'z' or 'A'-'Z' (see Table F.2 in Howto F for more information.) The stream function `get()` reads one character at a time from a stream; white space is not skipped. More information on `get` is found in Howto A and in Chapter 9.



Program 8.3 *letters.cpp*

```
#include <iostream>
#include <fstream>           // for ifstream
```

³The standard `vector` class initializes all vector elements, including built-in types. Built-in types are initialized to 0, where 0 means `false` for `bool` values and 0.0 for `double` values, for example. The `tvector` class uses a different method to allocate memory than the standard `vector` class, so cells will not, necessarily, have a defined value unless one is supplied when the `tvector` is constructed.

⁴This is not quite true of arrays, as we'll see later in this chapter. This is another reason to prefer using the `tvector` class to using built-in arrays.

8.2 Defining and Using tvector

349

```
#include <cstdlib>           // for exit()
#include <cctype>           // for tolower()
#include <climits>         // for CHAR_MAX
#include <string>
#include <iomanip>
using namespace std;

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

// count # occurrences of all characters in a file
// written: 8/5/94, Owen Astrachan, modified 5/1/99

void Print(const tvector<int> & counts, int total);
void Count(istream & input, tvector<int> & counts, int & total);

int main()
{
    int totalAlph = 0;
    string filename = PromptString("enter name of input file: ");
    ifstream input(filename.c_str());

    if (input.fail() )
    {
        cout << "could not open file " << filename << endl;
        exit(1);
    }
    tvector<int> charCounts(CHAR_MAX+1,0);    // all initialized to 0

    Count(input,charCounts,totalAlph);
    Print(charCounts,totalAlph);

    return 0;
}

void Count(istream & input, tvector<int> & counts, int & total)
// precondition: input open for reading
//                counts[k] == 0, 0 <= k < CHAR_MAX
// postcondition: counts[k] = # occurrences of character k
//                total = # alphabetic characters
{
    char ch;
    while (input.get(ch))           // read a character
    {
        if (isalpha(ch))           // is alphabetic (a-z)?
        {
            total++;
        }
        ch = tolower(ch);          // convert to lower case
        counts[ch]++;              // count all characters
    }
}

void Print(const tvector<int> & counts, int total)
// precondition: total = total of all entries in counts['a']..counts['z']
// postcondition: all values of counts from 'a' to 'z' printed
{
    const int MIDALPH = 13;
```

```

cout.setf(ios::fixed);    // print 1 decimal place
cout.precision(1);
char k;
for(k = 'a'; k <= 'm'; k++)
{
    cout << k << setw(7) << counts[k] << " ";
    cout << setw(4) << 100 * double(counts[k])/total << "% \t\t";
    cout << char(k+MIDALPH) << setw(7) << counts[k+MIDALPH] << " ";
    cout << setw(4) << 100 * double(counts[k+MIDALPH])/total << "%" << endl;
}
}

```

letters.cpp

For all practical purposes, a `char` variable is an integer constrained to have a value between 0 and `CHAR_MAX`. Since `char` variables can be used as integers, we can use a `char` variable to index an array. We'll use the vector element with index 'a' to count the occurrences of 'a', the element with index 'b' to count the b's, and so on. The constant `CHAR_MAX` is defined in `<climits>` (or `<limits.h>`.) We use it to initialize `charCounts`, a `tvector` of counters, so that all counters are initially zero.

```
tvector<int> charCounts(CHAR_MAX+1, 0);
```

Only the 26 vector elements corresponding to the alphabetic characters 'a' through 'z' are printed, but every character is counted.⁵ An alternative method of indexing `charCounts` that uses only 26 array elements rather than `CHAR_MAX` elements is explored in the Pause to Reflect exercises. To make the output look nice, we use stream member functions to limit the number of places after a decimal point when a `double` value is printed. These member functions are discussed in Howto B.



`tvector` parameters should always be passed by reference, unless you need to pass a copy of the `tvector` rather than the `tvector` itself, but it's very rare to need a copy. Avoid copying, because it takes time and uses memory. Some functions require value parameters, but these are rare when `tvector` parameters are used, so you should use reference or const-reference parameters all the time. Use a const reference parameter, as shown in `Print` in Program 8.3, when a `tvector` parameter isn't changed. A const reference parameter is efficient and also allows the compiler to catch inadvertent attempts to change the value of the parameter. The parameter `counts` in the function `Print` is *not* changed; its contents are used to print the values of how many times each letter occurs.

Program Tip 8.1: `tvector` parameters should be passed by reference (using `&`) or by const reference. Use a const reference parameter as part of a defensive programming strategy when a parameter is not changed, but is passed by reference because of efficiency considerations.

⁵I had a bug in the version of this program that appeared in the first edition: I used `CHAR_MAX` instead of `CHAR_MAX+1` as the size of the vector. If `CHAR_MAX` has the value 255, then the array will have 255 elements, but the largest index will be 254, and a character with value 255 will cause an illegal-index error. I never encountered this error in practice because I use `letters.cpp` to read text files, and the characters in text files typically don't have values of `CHAR_MAX`. This kind of off-by-one indexing error is common when using vectors. Some people call this an OBOB error (off-by-one bug).

Notice that the `for` loop in the function `Print` uses a `char` variable to index the values between `'a'` and `'z'`. The loop runs only from `'a'` to `'m'` because each line of output holds data for two letters, such as `'a'` and `'n'` or `'b'` and `'o'`. The result of adding 13 to `'a'` is `'n'`, but the explicit cast to `char` in `Print()` of Program 8.3 ensures that a character is printed. When ASCII values are used, these characters `'a'` to `'z'` correspond to array cells 97 to 122 (see Table F.3 in Howto F.)

O U T P U T

```
prompt> letters
enter name of input file: hamlet.txt
a  9950  7.6%      n  8297  6.4%
b  1830  1.4%      o 11218  8.6%
c  2606  2.0%      p  2016  1.5%
d  5025  3.9%      q   220  0.2%
e 14960 11.5%      r  7777  6.0%
f  2698  2.1%      s  8379  6.4%
g  2420  1.9%      t 11863  9.1%
h  8731  6.7%      u  4343  3.3%
i  8511  6.5%      v  1222  0.9%
j   110  0.1%      w  3132  2.4%
k  1272  1.0%      x   179  0.1%
l  5847  4.5%      y  3204  2.5%
m  4253  3.3%      z    72  0.1%
```

Pause to Reflect



- 8.1 In Program 8.1, how many lines must be changed or added to simulate two 12-sided dice? How many lines must be changed or added in Program 8.2 to simulate two 12-sided dice?
- 8.2 What changes must be made to Program 8.2 to simulate the rolling of three 6-sided dice?
- 8.3 Write definitions for a `tvector` `doubVec` of 512 doubles and `intVec` of 256 ints. Write code to initialize each vector location to twice its index so that `doubVec[13] = 26.0` and `intVec[200] = 400`.
- 8.4 Is it possible to create a vector of `Balloons` as declared in Program 3.7, `gballoon.h`? Why?
- 8.5 Write a definition for a `tvector` of strings that stores the names of the computer scientists for whom “Happy Birthday” was printed in Program 2.6, `bday2.cpp`. Write a loop that would print the song for all the names stored in the vector.

8.6 Suppose *letters.cpp* is modified so that the count of how many times 'a' occurs is kept in the vector element with index zero (and the count of 'z' occurrences is in the vector element with index 25). What changes are needed to do this (hint: if 'a' + 13 == 'n' as shown in `Print`, the value of 'b' - 'a' is 1 and the value of 'z' - 'a' is 25).

8.7 Write a short program, with all code in `main`, that determines how many two-letter, three-letter, ..., up to 15-letter words there are in a text file.

8.2.4 A `tvector` Case Study: Shuffling CD Tracks

Many CD players have an option for “random play.” Pressing the random-play or shuffle button causes the tracks on the CD to be “shuffled” and played in some arbitrary order, which may be different each time the CD is played. CD jukeboxes shuffle collections of CDs rather than just the collection of tracks on a single CD. In this section we’ll develop the program *shuffle.cpp*, Program 8.4, to simulate this random-play feature for a single CD.

We’ll need to store the tracks in a `tvector` and rearrange the elements in the `tvector` to simulate the shuffling. We’ll want to identify the original track number as well as the title of the track, so we’ll use a `struct` to encapsulate this information.

Developing the Program. We’ll start with the declaration below for a `struct Track` to store information about each track on a CD. All the tracks on a CD are stored in a `tvector<Track>` object.

```
struct Track
{
    string title;    // title of song/track
    int    number;  // the original track number
};
```

Rather than designing, coding, and testing the entire program at once, we’ll concentrate first on the two main features of the program: printing, and shuffling CD track information. Before shuffling, we’ll need to print, so we’ll implement `Print` first. Programming Tip 7.2 reminds us to grow a program — develop a program by adding to a working program rather than implementing the entire program at once.

A function to print the contents of a vector will need the vector and the number of elements in the vector. We’ll write a function to encapsulate the loop below that prints the first `count` elements of a vector `tracks`.

```
int k;
for(k=0; k < count; k++)
{    cout << tracks[k].number << "\t"
    << tracks[k].title << endl;
}
```

Sometimes it is hard to interpret (and even read) the expressions from the loop above that follow:

```
tracks[k].title;
tracks[k].number;
```

To decipher such expressions, you can read them inside out, one piece at a time.⁶ The `[]` are used to indicate an entry in a vector. The identifier to the left of them indicates that the name of the `tvector` is `tracks`. The identifier `k` is used to select a particular cell—note that the initial value of `k` is 0, indicating the first cell. I read the first expression as “tracks sub `k` dot title.”

Now you should think about what kind of element is represented by `tracks[k]`, what is stored in `tracks`? We’re dealing with a vector of `Track` structs. Now you should think about what `Track` is. It’s a `struct`, so, as with a class, a period or dot `.` is needed to access one of its fields. The `struct Track` has two fields: `title` and `number`. Examining the `struct` declaration may remind you what type each field is. In particular, `title` is a `string`.

Initializing a tvector. To test a print function we’ll need to store track information in a vector. Instead of reading track names from a file, we’ll test by hard-wiring several tracks in `main`, then pass the vector to the print function. Given the declaration for the `struct Track` above, we’re stuck writing code like the following:

```
tvector<Track> tracks(9);

tracks[0].title= "The Squirming Coil";
tracks[0].number= 1;
tracks[1].title= "Reba";
tracks[1].number= 2;
...
```

When you find yourself writing ugly code like this you should say to yourself, “There must be a better way.”

Program Tip 8.2: If you find yourself writing code that seems unnecessarily redundant, tedious, or that just offends your sense of aesthetics (it’s ugly), step back and think if there might be a way to improve the code.

Sometimes you’ll just have to write code you don’t consider ideal, either because you don’t know enough about the language, because you can’t think of the right approach, or because there just isn’t any way to improve the code. Ugly code is often a maintenance headache, and some time invested early in program development can reap benefits during the lifetime of developing and maintaining a program.

In this case, adding a constructor to the `struct Track` makes initialization simpler. We want to write code like the following:

⁶Sometimes the most inside piece isn’t obvious, but there are often several places to start.

```
tvector<Track> tracks(10);

tracks[0] = Track("The Squirming Coil",1);
tracks[1] = Track("Reba",2);
tracks[2] = Track("My Sweet One",3);
...
```

Adding a two-parameter constructor to the struct lets us write this code; see the new declaration for `Track` in *shuffle.cpp*, Program 8.4. Since we want to make a vector of `Track` structs we must supply a default/parameterless constructor as well (see the Syntax Diagram for `tvector` construction.) With initialization in `main` and the implementation of `Print`, we're ready to remove compilation errors, test the program, and then add the shuffling function. When we write `Print` we'll need to pass the number of elements in the vector. As we'll see in the next section, we can avoid using two parameters by having the vector keep track of how many elements it has, but for now we'll pass two parameters to `Print`: a vector and a count of how many elements are in the vector.

Program Tip 8.3: Functions that have `tvector` parameters sometimes require an `int` parameter that specifies the number of values actually stored in the `tvector`. The number of values stored is often different from the capacity of the `tvector`. We'll see that it's easy to avoid this second size parameter if the vector itself keeps track of the number of values it stores as well as its capacity.

We'll discuss the shuffling algorithm and code after the program listing.

Program 8.4 *shuffle.cpp*

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

#include "tvector.h"
#include "randgen.h"

struct Track
{
    string title; // title of song/track
    int    number; // the original track number

    Track::Track()
        : title("no title"),
          number(0)
    { }

    Track::Track(const string& t, int n)
        : title(t),
```

```

        number(n)
    { }
};

void Print(const tvector<Track>& tracks, int count)
// precondition: there are count locations in tracks
// postcondition: contents of tracks printed
{
    int k;
    for(k=0; k < count; k++)
    {
        cout << tracks[k].number << "\t" << tracks[k].title << endl;
    }
}

void Shuffle(tvector<Track> & tracks,int count)
// precondition: count = # of entries in tracks
// postcondition: entries in tracks have been randomly shuffled
{
    RandGen gen;    // for random # generator
    int randTrack;
    Track temp;
    int k;
    // choose a random song from [k..count-1] for song # k

    for(k=0; k < count - 1; k++)
    {
        randTrack = gen.RandInt(k,count-1);    // random track
        temp = tracks[randTrack];             // swap entries
        tracks[randTrack] = tracks[k];
        tracks[k] = temp;
    }
}

int main()
{
    tvector<Track> tracks(10);

    tracks[0] = Track("Box of Rain",1);
    tracks[1] = Track("Friend of the Devil",2);
    tracks[2] = Track("Sugar Magnolia",3);
    tracks[3] = Track("Operator",4);
    tracks[4] = Track("Candyman",5);
    tracks[5] = Track("Ripple",6);
    tracks[6] = Track("Brokedown Palace",7);
    tracks[7] = Track("Till the Morning Comes",8);
    tracks[8] = Track("Attics of my Life",9);
    tracks[9] = Track("Truckin",10);

    Print(tracks,10);
    Shuffle(tracks,10);
    cout << endl << "-- after shuffling --" << endl << endl;
    Print(tracks,10);
}

```

shuffle.cpp

Each time the program is run a different order of tracks is generated.

```

                                O U T P U T
prompt> shuffle
1      Box of Rain
2      Friend of the Devil
3      Sugar Magnolia
4      Operator
5      Candyman
6      Ripple
7      Brokedown Palace
8      Till the Morning Comes
9      Attics of my Life
10     Truckin

---- after shuffling ----

5      Candyman
2      Friend of the Devil
8      Till the Morning Comes
4      Operator
10     Truckin
7      Brokedown Palace
6      Ripple
3      Sugar Magnolia
9      Attics of my Life
1      Box of Rain

```

ShufflingTracks. The shuffling algorithm we'll employ is simple and is good theoretically—that is, it really does shuffle things in a random way. In this case each of the possible arrangements, or **permutations**, of the tracks is equally likely to occur.

The basic algorithm consists of picking a track at random to play first. This can be done by rolling an N -sided die, where there are N tracks on the CD, or by using the `RandGen` class used in Program 7.14, *brownian.cpp*. Once the first random track is picked, one of the remaining tracks is picked at random to play second. This process is continued until a song is picked for the first track, second track, and so on through the N th track. Without a `tvector` this would be difficult (though not impossible) to do. Program 8.4, *shuffle.cpp*, performs this task.

The expression `randTrack = gen.RandInt(k, count-1)` is used in the function `Shuffle` to choose a random track from those remaining. The first time the `for` loop is executed, the value of `k` is 0, all the tracks are eligible for selection, and the random number is a valid index between 0 and `count-1` (which is a number from 0 to 9 in *shuffle.cpp*.) The contents of the `tvector` cell at the randomly generated

index are swapped with the contents of the cell with index 0 so that the random-index track is now the first track. The next time through the loop, the random number chosen is between 1 and `count-1` so that the first track (at index 0) cannot be chosen as the random track.

Pause to Reflect



8.8 Suppose a new function `Initialize` is added to `shuffle.cpp` to initialize the elements of a vector of `Track` structs. Write the header/prototype, pre-, and post-conditions for the function. You'll need two parameters, just as the two functions `Print`, and `Shuffle` have.

8.9 In `Print`, why can't the output be generated by this statement?

```
cout << tracks[k] << endl;
```

8.10 In `Shuffle`, is it important that the test of the `for` loop be `k < count - 1` instead of `k < count`? What would happen if the test were changed?

8.11 The statement below from `Shuffle` assigns the contents of one vector element to another.

```
tracks[randTrack] = tracks[k];
```

What kind of object is assigned in this statement? How many assignments do you think are part of this assignment?

8.12 Suppose no items are specifically assigned in `main`, but instead this code is used.

```
tvector<Track> tracks(10);
Print(tracks,10);
Shuffle(tracks,10);
Print(tracks,10);
return 0;
```

Would you be able to tell if the shuffle function works? Why (what's printed)?

8.13 A different method of shuffling is suggested by the following idea. Pick two random track numbers and swap the corresponding vector entries. Repeat this process 1,000 times (or some other time as specified by a constant). Write code that uses this method for shuffling tracks. Do you have any intuition as to why this method is "worse" than the method used in `shuffle.cpp`?

8.3 Collections and Lists Using `tvector`s

Our first example programs used vectors as counters to determine how many times each of several possible simulated dice rolls occurs and how many times each character in an input file occurs. As we saw with the CD track-shuffling program, it's possible to use vectors to store objects other than counters. For example, we could store all the words

from a text file in a vector and write a program like Program 6.16, *maxword3.cpp*, to find the most frequently occurring word. Using a vector will make the program execute quickly since words will be in memory (in a vector) rather than on disk as they're scanned repeatedly to find the word that occurs most often.

In many programs, the number of items stored in a vector will not be known when the program is compiled, but will be determined at runtime. This would be the case, for example, if we store all the words in a text file in a vector. How big should we define vectors to be in order to accommodate the many situations that may arise? If we make a vector that can hold lots of data, to accommodate large text files, then we'll be wasting memory when the program is run on small text files. Conversely, if the vector is too small we won't be able to process large files. Fortunately, vectors can grow during a program's execution so that vector usage can be somewhat efficient. There will be some inefficiency because to grow a vector we'll actually have to make a new one and throw out the old one. As a metaphor, suppose you keep addresses and phone numbers of friends in an electronic personal organizer. You may become so popular, with so many friends, that you run out of memory for all the addresses you store. You may be able to buy more memory, but with most organizers you'll need to replace the old memory chip with a larger chip. This means you'll need to copy the addresses you've saved (to a computer, for example, but onto paper if you're really unlucky), install the new memory, then copy the addresses into the new memory.

8.3.1 Size and Capacity

In general, the number of elements stored in a vector will not be the same as the capacity of the vector. The capacity is how many elements could be stored, or how many cells the vector has. The size is the number of elements that are actually stored in the vector. These are different ideas conceptually, and programs will usually need to track both quantities separately. Using the electronic organizer as an example again, the capacity is how many names and addresses the organizer is capable of storing (how much memory it has) whereas the size is how many names are currently stored in it. Although client code can grow a vector explicitly, it's usually simpler and more efficient to have the vector grow itself.

8.3.2 Using `push_back`, `resize`, and `reserve`

When a vector is defined with an explicit size as an argument to the constructor, this argument determines both the capacity and the size of the vector. The size is determined since default objects will be constructed in each vector cell. Member function `tvector::size()` returns the size.

```
tvector<Date>    holidays(17);    // holidays.size() == 17
tvector<double> values(1000);    // values.size() == 1000
tvector<int>    scores;          // scores.size() == 0
tvector<string> names(10, "Joe"); // names.size() == 10
```

The member function `tvector::push_back` is used to add elements to the end of a vector; the vector resizes itself as needed by doubling its capacity. The lines below illustrate `push_back` and how size and capacity change each time an element is added.

```
tvector<string> names; // size() == 0, capacity() == 0
names.push_back("Fred"); // size() == 1, capacity() == 2
names.push_back("Wilma"); // size() == 2, capacity() == 2
names.push_back("Barney"); // size() == 3, capacity() == 4
names.push_back("Betty"); // size() == 4, capacity() == 4
names.push_back("Pebbles"); // size() == 5, capacity() == 8
```

The size of a vector is determined by three things.

- The number of times `push_back` is called, each call increases the size by one.
- The initial size of a vector when an argument is supplied at construction, this initial value is the size and the capacity.
- The argument in a call to `tvector::resize()` which changes the size and can change the capacity when the vector grows (resizing cannot shrink the capacity).

The code below prints the values stored in `names` in the example above.

```
int k;
for(k=0; k < names.size(); k++)
{ cout << names[k] << endl;
}
```

O U T P U T

```
Fred
Wilma
Barney
Betty
Pebbles
```

If a vector is given a size at construction, with subsequent elements added using `push_back`, the method `tvector::size` will not return the number of elements added by calling `push_back`.

```
tvector<string> names(7); // size() == capacity() == 7
names.push_back("Grace"); // size() == 8, capacity() == 14
names.push_back("Alan"); // size() == 9, capacity() == 14
```

The value of `names[0]` is `" "` because this value is constructed by the default string constructor. The value of `names[7]` is `"Grace"` since the initial size puts default string values in array elements 0–6.

A vector grows when its size and capacity are equal and `push_back` adds a new element to the vector. When a vector grows itself by client programs calling `push_back`, the capacity doubles.⁷

Since the capacity doubles, it might go from 8 to 16 to 32 and so on. If you're writing a program and you know you'll need to store at least 5,000 elements, this growing process can be inefficient.⁸ The member function `tvector::reserve()` is used to create an initial capacity, but the size remains at zero.

```
tvector<string> names; // size() == 0, capacity() == 0
names.reserve(2048); // size() == 0, capacity() = 2048
```

Program Tip 8.4: If you're going to use `push_back` do not define a vector by giving a size when the vector is constructed. If you construct with a size, the method `tvector::size` won't return the number of elements added by `push_back`, it will return that number plus the initial size. If you want to allocate space for efficiency reasons use `tvector::reserve`.

We'll use two functions in Program 8.5 that read words from a file and store them in a vector to illustrate the differences between using `push_back` and calling `resize` explicitly. The runs also show that using `tvector::reserve` can lead to increased efficiency when a vector would double frequently otherwise.

Program 8.5 `growdemo.cpp`

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

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

// show differences between push_back and calling resize explicitly

void ReadAll(WordStreamIterator& iter, tvector<string>& list)
// precondition: all words from iter stored in list
{
    for(iter.Init(); iter.HasMore(); iter.Next())
    {
        list.push_back(iter.Current());
    }
}
```

⁷The class `tvector` doubles its capacity each time except when the capacity is initially zero, that is, when the vector is first constructed. The capacity goes from 0 to 2, and then doubles each time. The standard `vector` class should double in capacity too, but implementations are not required to double the capacity. Most implementations use doubling, but there may be some that don't.

⁸Recall that doubling requires copying the elements into a new vector that's twice as large.

```
}

void ReadAll2(WordStreamIterator& iter,
             tvector<string>& list, int& count)
// postcondition: all words from iter stored in list,
//               count = number of words read
{
    count = 0;
    for(iter.Init(); iter.HasMore(); iter.Next())
    { if (count >= list.capacity())
      { list.resize(list.capacity()*2 + 1); // grow by doubling
      }
      list[count] = iter.Current();
      count++;
    }
}

int main()
{
    CTimer timer;
    string filename = PromptString("enter filename ");
    WordStreamIterator iter;
    iter.Open(filename);

    tvector<string> listA; // listA.reserve(100000);
    tvector<string> listB; // listB.reserve(100000);

    timer.Start();
    ReadAll(iter,listA);
    timer.Stop();
    cout << "# words: " << listA.size()
          << " capacity: " << listA.capacity()
          << " time: " << timer.ElapsedTime() << endl;

    int count; // # elements stored in listB
    timer.Start();
    ReadAll2(iter,listB,count);
    timer.Stop();
    cout << "# words: " << count
          << " capacity: " << listB.capacity()
          << " time: " << timer.ElapsedTime() << endl;
    return 0;
}
```

O U T P U T

```

enter filename hamlet.txt
# words: 31956 capacity: 32768 time: 0.751
# words: 31956 capacity: 32767 time: 0.941
enter filename hawthorne.txt
# words: 85753 capacity: 131072 time: 2.874
# words: 85753 capacity: 131071 time: 4.587

calls to reserve uncommented or both listA and listB

enter filename hawthorne.txt
# words: 85753 capacity: 100000 time: 1.302
# words: 85753 capacity: 100000 time: 1.302

```

The code in `ReadAll` is considerably simpler than the code in `ReadAll2`. As the runs show, `ReadAll` is also more efficient when there is considerable doubling.⁹

Pause to Reflect



- 8.14** If the `WordStreamIterator` is replaced by an `ifstream` variable in Program 8.5, the call to `ReadAll` returns the same values, but the call to `ReadAll2` returns a value of zero in reference parameter `count`, with nothing stored in the vector. Why?
- 8.15** Why is the expression `list.capacity()*2 + 1` used in `ReadAll2` of `growdemo.cpp` rather than `list.capacity()*2`?
- 8.16** What value would be returned by `listB.size()` during the middle run shown in the output box (when `listB.capacity()` returns 131071).
- 8.17** What changes are needed in `main` of Program 8.4, `shuffle.cpp` to use `push_back`? How could the functions `Print` and `Shuffle` change to take advantage of using `push_back` in `main`?
- 8.18** A `tvector` is constructed with size zero, then grows itself to a size of 2, 4, 8, 16, ...vector elements (assuming `reserve` is not used). Each time the vector grows, new memory is allocated, and old memory de-allocated. When the capacity of the vector is 512 how many vector elements has been allocated (including the final 512)? If the capacity is 16,384 how many vector elements have been allocated?
- 8.19** If a `tvector` grows by one vector element instead of doubling, (e.g., grows to 1, 2, 3, 4, ...elements) then how many elements have been allocated when the capacity is 32 (including the final 32)? When the capacity is 128? When the capacity is 16,384? (Hint: $1 + 2 + \dots + n = n(n + 1)/2$.)

⁹The efficiency improvements are a property of the `tvector` implementation. When the standard class `vector` is used instead of `tvector` in `growdemo.cpp` the efficiency gains are not nearly as pronounced.

8.20 Why do you think the time used in *growdemo.cpp*, Program 8.5 by the `push_back` function `ReadAll` is less than the time used by the function `ReadAll2` (when `reserve` isn't used)?

8.3.3 Vector Idioms: Insertion, Deletion, and Searching

To illustrate common vector operations, we'll use a small program that reads information representing a portfolio of stocks. We'll show examples of adding a new stock, deleting a stock, and finding stocks that match certain criteria, that is, trading below \$50.00, above \$100.00, on the NASDAQ exchange,¹⁰ or stocks whose symbols begin with the letter 'Q'.

We'll read a file of information similar to what's shown below, but without the company name on the end of each line.¹¹ The information below is out-of-date; it is from 1996 and not meant to reflect current stock prices. The data for each stock includes its symbol, such as, *KO*, the exchange (N = New York, T = NASDAQ), the price, the number of shares traded, and the name of the company which doesn't appear in the data file we'll use.

KO	N	50.5	735000	COCA COLA CO
DIS	N	64.125	282200	DISNEY CO WALT HLDG CO
ABPCA	T	5.688	49700	AU BON PAIN CO INC CL A
NSCP	T	42.813	385900	NETSCAPE COMM CORP
F	N	32.125	798900	FORD MOTOR CO

Program 8.6 uses a class `Portfolio` to read and print a collection of stocks.

Program 8.6 `stocks.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"

struct Stock
{
    string name;
    string exchange;
    double price;
};
```

¹⁰There are several stock exchanges in the world. Examples include the New York Exchange, the NASDAQ exchange, the Toronto Exchange, and others.

¹¹The other information on a line can be read using `>>`, but the company name requires the use of the function `getline` because the name consists of more than one word. We'll study `getline` in Chapter 9.

364

Chapter 8 Arrays, Data, and Random Access

```
int    shares;

Stock()
: name("dummy"),
  exchange("none"),
  price(0.0),
  shares(0)
{ }

Stock(const string& n, const string& xc,
      double p, int ns)
: name(n),
  exchange(xc),
  price(p),
  shares(ns)
{ }
};

class Portfolio
{
public:
    Portfolio();
    void Read(const string& filename);

    void Print(ostream& out) const;

private:
    tvector<Stock> myStocks;
};

Portfolio::Portfolio()
: myStocks(0)
{
    myStocks.reserve(20);    // start with room for 20 stocks
}

void Portfolio::Read(const string& filename)
{
    ifstream input(filename.c_str());
    string symbol, exchange, price, shares;

    while (input >> symbol >> exchange >> price >> shares)
    {
        myStocks.push_back(Stock(symbol, exchange, atof(price), atoi(shares)));
    }
}

void Portfolio::Print(ostream& out) const
{
    int k;
    int len = myStocks.size();

    out.precision(3);        // show 3 decimal places
    out.setf(ios::fixed);    // for every stock price

    for(k=0; k < len; k++)
```

```

    {   out << myStocks[k].name << "\t"
        << myStocks[k].exchange << "\t"
        << setw(8) << myStocks[k].price << "\t"
        << setw(12) << myStocks[k].shares << endl;
    }
    cout << endl << "---" << endl << "# stocks: " << len << endl;
}

int main()
{
    string filename = PromptString("stock file ");
    Portfolio port;

    port.Read(filename);
    port.Print(cout);

    return 0;
}

```

stocks.cpp



The conversion functions `atoi` and `atof` from `strutils.h` are discussed in Howto G. The formatting functions `precision` and `setw` for displaying a fixed number of decimal places are discussed in Howto B.

O U T P U T

```

prompt> stocks
stock file 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

```

The `Portfolio` constructor initializes the instance variable `myStocks` to have zero elements in the initializer list, then reserves space for 20 stocks in the body of the constructor. `tvector` instance variables *must be* constructed in an initializer list. It's not possible to include the size of a vector in the class declaration, (e.g., the following code does not work).

```

class Thing
{
    ...
private:
    tvector<int> myData(30); // ***illegal***
};

```

A class declaration does not allocate memory; memory is allocated in the class definition, specifically in a constructor. This means you must construct each private `tvector` data field in the initializer list of each constructor.¹²

Program Tip 8.5: When a `tvector` instance variable is used in a class, each constructor for the class should explicitly construct the `tvector` in the constructor's initializer list. A vector can be given a size, or sized to zero with space reserved by a call to `tvector::reserve` in the constructor body.

8.3.4 Insertion into a Sorted Vector

As shown in *stocks.cpp*, the function `push_back` is simple to use and effectively adds a new element to the end of a vector. What can you do if you want to add a new element to the middle of a vector, or to some other location? If the list of stocks is kept in alphabetical order by symbol, for example, new stocks should be added in the correct location to keep the list in sorted order. The only way to do this with a vector is to shift elements to create an empty vector cell for the new element.

Suppose, for example, that you keep books arranged alphabetically by author, or a collection of compact discs (CDs) arranged alphabetically by artist. When you get a new book (or a new CD), you'll probably have to move or shift several books to make a spot for the new one. If you're facing a bookshelf, you might start at the rightmost end and slide the books to the right until you find where the new book belongs. This mimics exactly how new elements are inserted into a vector when the vector is maintained in sorted order.

We'll write code to shift vector elements to the right. The key statement follows.

```
myStocks[loc] = myStocks[loc-1];
```

When `loc` has the value eight, for example, this copies the element with index seven into the vector cell with index eight, effectively shifting an element to the right. After this statement executes, the element in the vector cell with index seven is still there, but has been copied into the cell with index eight as well. We'll stop shifting when we've looked at every vector element or when we find where the new stock belongs in the sorted order. The code below inserts a stock `s` in sorted order by symbol.¹³

```
void Portfolio::Add(const Stock& s)
// postcondition: s added to portfolio in sorted order
{
    int count = myStocks.size(); // size before adding
    myStocks.push_back(s);       // vector size is updated
}
```

¹²If you don't include an explicit `tvector` constructor in a class' initializer list, the vector will have zero elements, which is actually the right thing to do if you're using `push_back`.

¹³This code is from *stocks2.cpp*, not shown in the book, but available with the programs that come with the book or from the book website.

```

int loc = count;

while (0 < loc && s.symbol <= myStocks[loc-1].symbol)
{
    myStocks[loc] = myStocks[loc-1];
    loc--;
}
myStocks[loc] = s;
}

```

The new stock is first inserted at the end of the vector using `push_back` simply to allow the vector to update its count of how many elements are in the vector. Elements are then shifted and the stock `s` is stored in the proper location when the loop finishes.

To understand and reason about the loop that shifts elements to the right, we'll concentrate on three properties of the variable `loc`. These properties are true each time the loop test is evaluated, so they constitute a **loop invariant** and should help us reason about the correctness of the loop.

- `loc-1` is the index of the item that will be shifted right if necessary; this is the rightmost element not yet processed.
- `loc` is the index of the cell in which the new stock will be inserted in sorted order.
- All items with index `loc + 1` through index `count` are greater than the new stock being inserted.

Figure 8.4 illustrates the process of inserting a stock with symbol 'D' into a sorted vector (for the purposes of illustration, all symbols are single characters.) Initially the vector has eight elements, so the value of `loc` is 8. The three properties that make up the loop invariant hold the first time the loop test is evaluated.

- `loc-1`, or 7, is the index of `V`, the rightmost unprocessed element. It will be shifted as necessary.
- `loc`, or 8, is the cell in which the new stock will be stored (if the new stock has symbol 'Z', it is stored in location 8.)
- All items with indexes 9 through 8 are greater than the stock being inserted. In this case the range 9...8 represents an **empty range**, since $9 > 8$. There are no elements in this empty range, so it's true that all the elements in the range are greater than the element being inserted.¹⁴

When `loc` is 4, as shown in Figure 8.4, the three properties still hold. At this point the letters `Q`, `S`, `T`, and `V` have been shifted to the right, since the loop body has been executed for values of `loc` of 7, 6, 5, 4.

Since the loop test is true, the body is executed, and `M` is shifted to the right. Finally, when `loc == 2`, the three properties still hold:

¹⁴Don't worry too much about this. The key here is that it's impossible to find a word in the range 9...8 that's smaller than the word being inserted. It's impossible because there are no words in the empty range.

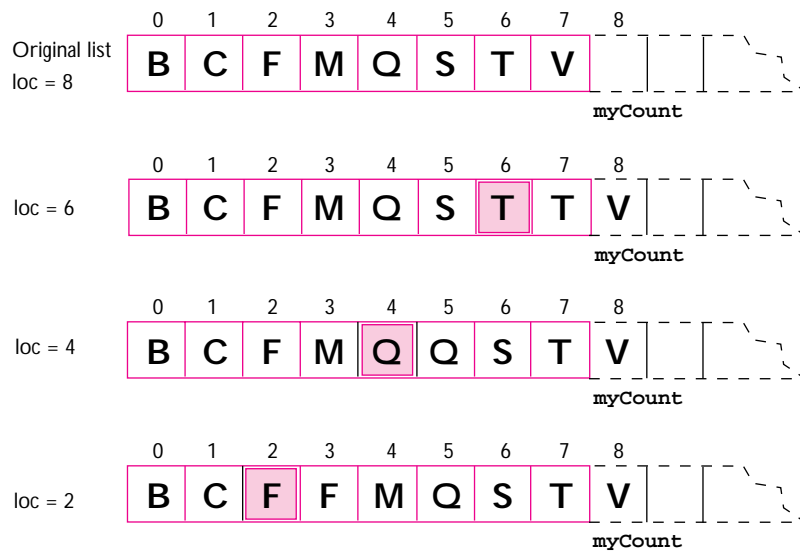
- `loc-1`, or 1, is the rightmost unprocessed element
- `loc`, or 2, is the index where the new stock will be inserted
- all items with indexes between 3 and 8 have values greater than 'D'.

However, the loop test is false, because `s.symbol > myStocks[loc].symbol` since `D > C`. The loop exits, and the new stock is inserted in the cell with index `loc`, as described by the invariant.

8.3.5 Deleting an Element Using `pop_back`

Deleting the last element of a vector is very simple; the method `tvector::pop_back` reduces a vector's size by one, effectively deleting the last element. The capacity of the vector is unchanged, but since client programs use `tvector::size()` to determine the number of elements stored in a vector, calling `pop_back` removes the element.

The code below shows a simple method for removing an element from the middle of a vector when the vector is *not* maintained in sorted order. The last element is copied into the vector cell that will be "deleted". Calling `pop_back` ensures that the vector updates its internal state properly.



Add D to vector maintained in sorted order

Figure 8.4 Maintaining a vector in sorted order. The new element will go in the vector cell with index `loc` when shifting is finished. The shaded location is being considered as the location of the new element.

```
// remove element with index loc from vector v
int lastIndex = v.size() - 1;
v[loc] = v[lastIndex];
v.pop_back();
```

If the vector is maintained in sorted order, vector elements must be shifted to delete an element while maintaining the sorted order. In contrast to the code that shifted elements to the right to make space for a new element, deletion requires shifts to the left.

```
// delete element with index loc
int k;
for(k=loc; k < myStocks.size()-1; k++)
{ myStocks[k] = myStocks[k+1];
}
myStocks.pop_back();
```

8.3.6 Searching a Vector

Searching and sorting are common applications in programming. In the stock portfolio example from *stocks.cpp*, Program 8.6, the program was modified to keep stocks in sorted order. In this section we'll see how to search for stocks that match some criterion. Sometimes searching will yield an exact, or single, match. If we search for the stock with symbol *HRL* we expect only one match. In general, searching for a stock by symbol should yield zero or one matches since stock symbols are unique. On the other hand, if we search for all stocks below \$10.00, or that traded more than 500,000 shares, there may be many matches.

Searching for a Unique Match. In a **sequential search** (sometimes called a **linear search**), elements in a vector are scanned in sequence, one after the other. Sequential search is necessary, for example, if you want to find the person whose phone number is 555-2622 in your local phone book. Phone books are arranged alphabetically by name rather than numerically by phone number, so you must scan all numbers, one after the other, hoping to find 555-2622.

A search function must return something. Typically the returned value is an index into the vector, or the matching element found during the search. Using an index as a return value makes it possible to encode a failed search by returning a bad index value like `-1`. If a vector element is returned, it's not possible, in general, to return a value indicating a failed search. Some people code search functions to return two values: a `bool` to indicate if the search is successful and the matching element. If the `bool` value is false, the matching element has no defined value. The code below shows a function that returns the index of a match in a vector of strings. This code can also be found in Program 8.7, *timesearch.cpp*.

```
int search(const tvector<string>& list, const string& key)
// pre: list.size() == # elements in list
// post: returns index of key in list, -1 if key not found
{
```

```

int k;
for(k=0; k < list.size(); k++)
{ if (list[k] == key)
  { return k;
  }
}
return -1; // reach here only when key not found
}

```

Counting Matches. You may want to know how many stocks sell for more than \$150.00 or traded more than 500,000 shares, but not care which stocks they are. This is an example of a **counting search** or **counting match**. Modifying the linear search code to count matches is straightforward. The sequential search code returned as soon as a match was found, but in counting all matches no early return is possible.

```

int countMatches(const tvector<Stock>& list, int minShares)
// pre: list.size() == # stocks in list
// post: returns # stocks that traded more than minShares shares
{
  int k, count = 0;
  for(k=0; k < list.size(); k++)
  { if (list[k].share > minShares)
    { count++;
    }
  }
  return count;
}

```

Collecting Matches. In the previous example, the function `countMatches` could determine the number of stocks that traded more than 500,000 shares, but could not determine which stocks these are. It would be simple to add an output statement to the function so that the stocks that matched were printed, but you may want to know the average price of the matching stocks rather than just a printed list of the stocks. The easiest way to collect matches in a search is to store the matches in a vector. The function below is a modification of `countMatches` that returns the matching stocks as elements of the parameter `matches`.

```

void collectMatches(const tvector<Stock>& list,
                   int minShares, tvector<Stock>& matches)
// pre: list.size() == # elements in list
// post: matches contains just the elements of list
//       that traded > minShares shares
{
  int k;
  matches.resize(0); // initially no matches

```

```

for(k=0; k < list.size(); k++)
{
    if (list[k].share > minShares)
    {
        matches.push_back(list[k]);
    }
}
}

```

The call to `matches.resize()` ensures that `matches` contains just the stocks that match the criterion of trading more than `minShares` shares. Recall that `resize` cannot reduce the capacity of a vector, but it does make the size zero.

Pause to Reflect



8.21 The loop below is designed to find the index at which a new item should be inserted in an array to keep the array in sorted order. The loop finds the index but doesn't insert. For example, if `list` is ("avocado", "banana", "lemon", "orange") and `s` is "cherry" the function should return 2; if `s` is "watermelon" the function should return 4.

```

int insertionIndex(const tvector<string>& list,
                  const string& s)
// pre: list is sorted, list[0] <= list[1] ... <= list[n]
//       where n = list.size()-1
// post: return index i of s in list, so that
//       i is largest value with list[0]..list[i-1] < s
{
    int len = list.size();
    int k=0;
    // invariant: list[0]..list[k-1] < s
    while (k < len && list[k] < s)
    {
        k++;
    }
    return k;
}

```

1. Why is `list` a const reference parameter?
2. What value should be returned if `s` is "apple"? Is this value returned?
3. Is 4 returned when `s` is "watermelon"?
4. Why is the text `k < len` needed?

8.22 Assuming the function `insertionIndex` from the previous problem satisfies its postcondition, write the function below which could be used as the basis for a new `Portfolio::Add` from Section 8.3.4.

```

void insertAt(tvector<string>& list,
              const string& s, int loc)
// post: s inserted into list at location with index loc
//       order of list elements unchanged

```

To insert a string into a sorted vector, leaving it sorted, the following call should work.

```
string s = "apple";
insertAt(list, s, insertionIndex(list,s));
```

8.23 In a vector of n elements, what is the fewest number of elements that are shifted to insert a new element in sorted order? What is the most number of elements that are shifted?

8.24 The method `tvector::clear` makes the size of a vector 0, the call `t.clear()` has the same effect as `t.resize(0)`. If there were no functions `clear` or `resize` you could write a function to remove all the elements of vector by calling `pop_back`. Write such a function.

8.25 Write a function `deleteAt` that works like `insertAt` from the second pause and reflect exercise in this section.

```
void deleteAt(tvector<string>& list, int loc)
// post: item at index loc removed,
//       order of other items unchanged
```

How could you call `deleteAt` to remove "banana" from the vector ("avocado", "banana", "lemon", "orange")?

8.26 Assume the function `isVowel` exists.

```
bool isVowel(const string& s)
// post: returns true if s is 'a', 'e', 'i', 'o', 'u'
//       (or upper case equivalent)
```

Write the function below.

```
int vowelCount(const tvector<string>& list)
// post: return # strings in list that begin with a vowel
```

Assuming `vowelCount` works, what expression returns the number of strings in a vector `list` that do *not* begin with a vowel?

8.27 Modify the function in the previous exercise to return a vector containing all the strings that begin with a vowel, instead of just the count of the number of strings.

8.28 Write a function to return the sum of all the elements in a vector of ints.

```
int sum(const tvector<int>& list)
// post: returns list[0] + ... + list[list.size()-1]
```

8.29 Write a function that removes duplicate elements from a sorted vector of strings.

```
void removeDups(tvector<string>& list)
// pre: list[0] <= ... <= list[list.size()-1] (sorted)
// post: duplicates removed from still sorted list
```

For example, the vector

```
("avocado", "avocado", "lemon", "lemon", "lemon", "orange")
```

should be changed to

```
("avocado", "lemon", "orange")
```

David Gries (b. 19??)

David Gries is a computer scientist and educator at Cornell University. He is well known for his advocacy of the use of formal methods in designing and implementing software and in the training of undergraduates in computer science. He has done perhaps more than any one person in making the study of loop invariants and formal methods accessible to students in introductory courses.



In his World Wide Web biography he writes of encounters with recursion when earning his master's degree in 1963: "it was fun, figuring out how to implement recursion efficiently before there were many papers on the topic." In an essay [Gri74] written in 1974 he provides timeless advice: *It must be made clear that one technique will never suffice (for example, top-down programming). A programmer needs a bag of tricks, a collection of methods for attacking a problem. Secondly, if we are to raise the level of programming, each programmer*

(no matter how good he feels he is) must become more conscious of the tools and techniques he uses. It is not enough to just program; we must discover how and why we do it.

Gries has twins, and in a coincidence of the highest order, the twins were born on the birthday of Gries and his twin sibling. In noting that he is (perhaps) better known for his educational work than his research work, Gries writes, "Do what you are good at; bloom where you are planted." For his work in education Gries was awarded the 1994 IEEE Taylor L. Booth Award, the 1991 ACM SIGCSE award, and the 1995 Karl V. Karlstrom Outstanding Educator Award.

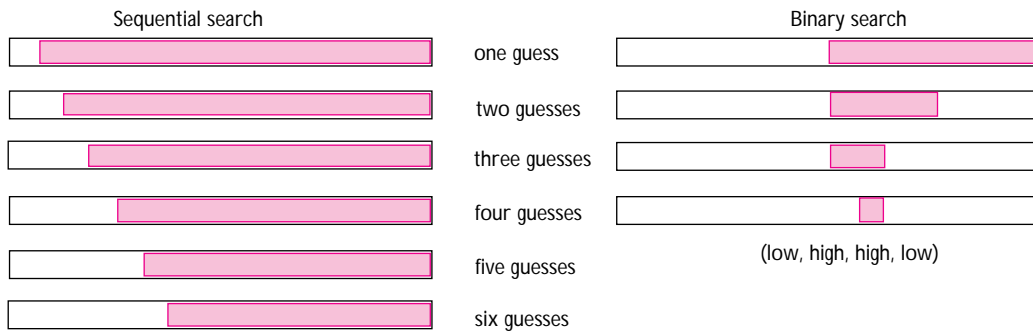


Figure 8.5 Comparing sequential/linear search, on the left, with binary search, on the right.

8.3.7 Binary Search

Phone books are arranged alphabetically by name rather than numerically by phone number, so you must scan all numbers, one after the other, hoping to find 555-2622. Of course if you were doing this, you could easily miss the number; people aren't good at this kind of repetitive task, but computers are. On the other hand, you can look up John Armstrong's, Nancy Drew's, or Mr. Mxyzptlk's number without scanning every entry. Since name/number pairs are stored alphabetically by name, it's possible to search for a name efficiently. In this section we'll investigate **binary search**: a method of searching that takes advantage of sorted data to speed up search. As we'll see, binary search is not always better than sequential search. Choosing the right searching algorithm depends on the context in which the search will be used.

Binary search is based on the method you may have used in playing a guess-a-number game. Suppose someone thinks of a number between 1 and 100 and will tell you whether your guess is low, high, or correct. You'll probably use 50 as the first guess. This will eliminate half of the numbers from consideration and is considerably more fruitful than guessing 1 (which, invariably, is low). The strategy of guessing the middle number works regardless of the the range of numbers. For example, if someone initially thinks of a number between 1 and 1024, you would guess 512. One guess shrinks the number of possibilities by half, from 1024 to 512. The number of possibilities continues to shrink from 512 to 256, 128, 64, 32, 16, 8, 4, 2, and finally 1. This is a total of 10 guesses to find one of 1024 possible different numbers. Consider what happens if you're told "yes" or "no" rather than high/low, and how this affects your guessing strategy. That example illustrates the difference between binary search and sequential search. Eliminating half of the numbers with one guess, rather than one number, is shown graphically in Figure 8.5. A `tvector` of 32 elements is shown; the shaded area represents the region of items still being considered after each guess is made. The size of the region being considered is reduced by half each time for binary search, but by only one for sequential search.

When binary search is used, each comparison cuts the range of potential matches in half. The total number of guesses will be how many times the initial number of items

can be cut in half. As we've seen, 1024 items require 10 guesses; it's not a coincidence that $2^{10} = 1024$. Doubling the number of items from 1024 to 2048 increases the number of guesses needed by only one, because one guess cuts the list of 2048 down to 1024 and we know that 10 guesses are needed for 1024 items. Again, it's not a coincidence that $2^{11} = 2048$.

Looking up a name in a phone book of 1024 names might require 11 guesses. When there is only one name left to check, it must be checked too, because the name being sought might not be in the phone book (this doesn't happen with the guess-a-number game). How many guesses are needed using binary search to search a list of one million names? As we've seen, this depends on how many times one million can be cut in half. We want to find the smallest number n such that $2^n \geq 1,000,000$; this will tell us how many items must be checked (we might need to add 1 if there's a possibility that the item isn't in the list; this cuts the final list of one item down to a list of zero items). Since $2^{19} = 524,288$ and $2^{20} = 1,048,576$, we can see that 20 (or 21) guesses are enough to find an item using binary search in a list of one million items. If you're familiar with logarithms, you may recall that log functions are the inverse of exponential functions, and therefore that the number of times a number x can be cut in half is $\log_2(x)$, or log base 2 of x . Again, we may need to add 1 if we need to cut a number in half to get down to zero instead of 1. This is the analog of reducing the items down to a zero-element list or a one-element list.

8.3.8 Comparing Sequential and Binary Search

We're more concerned with comparing sequential search and binary search than with the exact number of items examined with binary search. The difference between 20 and 21 items examined is far less important than the difference between 21 items (binary search) and one million items (sequential search). Although it's possible that only one item is examined when a sequential search is used (consider looking up a word like "aardvark" in the dictionary), the worst case is that one million items might need to be examined (consider looking up "zzzz" in a million-word dictionary). Table 8.1 provides a comparison of the number of items that must be examined using sequential and binary search.

Examining 18 items will be much faster than examining 100,000 items, but how

Table 8.1 Comparing sequential/linear search with binary search

list size	number of items examined	
	binary search	sequential search
1	1	1
10	4	10
1,000	11	1,000
5,000	14	5,000
100,000	18	100,000
1,000,000	21	1,000,000

much faster? If two strings can be compared in a microsecond (one millionth of a second)—which is very possible on moderately fast computers—both searches will take less than one second. Does it matter that binary search requires 0.000018 seconds and sequential search requires 0.1 seconds? The answer is, “It depends.” It probably won’t matter to you if you’re waiting for a response to appear on a computer monitor, but it may matter if the computer is “waiting” for the search and 100 million searches are necessary. On the computers I used to develop the code in this book, searching for one word in an on-line dictionary of 25,000 words appears to take no time to me using either sequential or binary search. To be precise, I can type a word to search for, press Enter, and the word found in the dictionary appears on the screen instantaneously.

However, in Program 8.7, *timesearch.cpp*, a file of words is read, and then every different word in the file is searched for in a vector of all the words in the file. To be precise, the following sequence takes place in *timesearch.cpp*.

1. All the words in a file are read and stored in a vector. Words are converted to lower case and leading/trailing punctuation is removed.
2. A `StringSet` is created from the words in the vector. The set is effectively a list of the different words in the file (the vector contains duplicates.)
3. A copy of the vector is made, and the copy is sorted. There are now two vectors: one sorted and one unsorted.¹⁵
4. Each word in the set is searched for in the vector. Sequential search is used with the unsorted vector; binary search is used with the sorted vector.

As you can see in the runs, the time to search using a sorted vector with binary search is very much faster than the time to search using sequential search. For Hawthorne’s *The Scarlet Letter*, searching for 9,164 different strings in a vector of 85,754 strings took 267 seconds using sequential search and only 0.17 seconds using binary search in a sorted vector. Of course it took more than one minute to sort the vector in order to use binary search, but the total time is still much less than the time for sequential search. On the other hand, consider the times for Poe’s *The Cask of Amontillado*. While still drastically different at 0.501 and 0.01 seconds, a user doesn’t see much impact in a process that finishes in half a second. That’s why the answer to whether binary search or sequential search is better is “It depends.”

Program 8.7 *timesearch.cpp*

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

#include "tvector.h"
#include "ctimer.h"
```

¹⁵The function `QuickSort` from *sortall.h* is used to sort. Sorting is discussed in Chapter 11, but you can call a sort function without knowing how it works.

8.3 Collections and Lists Using tvector

377

```
#include "strutils.h"          // for StripPunc and ToLower
#include "stringset.h"
#include "prompt.h"
#include "sortall.h"

// demonstrate differences between sequential and binary search
// Owen Astrachan, 5/4/99

void Read(const string& filename, tvector<string>& list)
// post: list is unsorted collection of all the strings
//       in text file filename, each string is converted to
//       lower case with leading/trailing punctuation removed
{
    ifstream input(filename.c_str());
    string word;
    while (input >> word)
    {   StripPunc(word);
        ToLower(word);
        list.push_back(word);
    }
}

void makeSet(const tvector<string>& list, StringSet& sset)
// post: sset is set of strings from list
{
    int k;
    int len = list.size();
    for(k=0; k < len; k++)
    {   sset.insert(list[k]);
    }
}

int search(const tvector<string>& list, const string& key)
// precondition: list.size() == # elements in list
// postcondition: returns index of key in list, -1 if key not found
{
    int k;
    for(k=0; k < list.size(); k++)
    {   if (list[k] == key)
        {   return k;
        }
    }
    return -1;    // reach here only when key not found
}

int bsearch(const tvector<string>& list, const string& key)
// precondition: list.size() == # elements in list
// postcondition: returns index of key in list, -1 if key not found
{
    int low = 0;                // leftmost possible entry
    int high = list.size()-1;   // rightmost possible entry
    int mid;                    // middle of current range
    while (low <= high)
    {   mid = (low + high)/2;
        if (list[mid] == key)    // found key, exit search
    }
```

378

Chapter 8 Arrays, Data, and Random Access

```
        {   return mid;
        }
        else if (list[mid] < key) // key in upper half
        {   low = mid + 1;
        }
        else // key in lower half
        {   high = mid - 1;
        }
    }
    return -1; // not in list
}

double timeLinear(const StringSet& sset, const tvector<string>& list)
{
    CTimer timer;
    StringSetIterator it(sset);

    timer.Start();
    for(it.Init(); it.HasMore(); it.Next())
    {   int index = search(list,it.Current());
        if (index == -1)
        {   cout << "missed a search for " << it.Current() << endl;
        }
    }
    timer.Stop();
    return timer.ElapsedTime();
}

double timeBinary(const StringSet& sset, const tvector<string>& list)
{
    CTimer timer;
    StringSetIterator it(sset);

    timer.Start();
    for(it.Init(); it.HasMore(); it.Next())
    {   int index = bsearch(list,it.Current());
        if (index == -1)
        {   cout << "missed a search for " << it.Current() << endl;
        }
    }
    timer.Stop();
    return timer.ElapsedTime();
}

int main()
{

    string filename = PromptString("enter file ");
    CTimer timer;
    tvector<string> list, sortedList;
    StringSet sset;

    timer.Start();
    Read(filename,list);
    timer.Stop();
}
```

```
cout << timer.ElapsedTime() << " secs to read "
    << list.size() << " total words" << endl;

timer.Start();
makeSet(list,sset);
timer.Stop();
cout << "make set time:\t" << timer.ElapsedTime() << " set size: "
    << sset.size() << endl;

timer.Start();
sortedList = list;
QuickSort(sortedList,sortedList.size());
timer.Stop();
cout << "make sorted time:\t" << timer.ElapsedTime() << endl;

cout << "unsorted search time:\t" << timeLinear(sset,list) << endl;
cout << "sorted search time:\t" << timeBinary(sset,sortedList) << endl;
return 0;
}
```

timesearch.cpp

OUTPUT

```
prompt> timesearch
enter file poe.txt
0.08 secs to read 2325 total words
make set time: 0.17 set size: 810
make sorted time: 0.09
unsorted search time: 0.501
sorted search time: 0.01

prompt> timesearch
enter file hamlet.txt
1.072 secs to read 31957 total words
make set time: 6.429 set size: 4832
make sorted time: 6.429
unsorted search time: 56.652
sorted search time: 0.08

prompt> timesearch
enter file hawthorne.txt
3.895 secs to read 85754 total words
make set time: 24.896 set size: 9164
make sorted time: 68.228
unsorted search time: 267.585
sorted search time: 0.17
```

The postconditions for functions `search` and `bsearch` in Program 8.7, *time-search.cpp*, are identical. You can use either function to search, but a vector must be sorted to use binary search.

8.4 Built-in Arrays

This section covers materials not used in this book other than in this section.

In this section we'll study the built-in C++ array type and compare it with the `tvector` class we've used to implement a homogeneous, random-access data structure. The `tvector` class is defined using the built-in C++ array type. Using built-in arrays results in code that will probably execute more quickly than when the `tvector` class is used, because of overhead associated with checking `tvector` indices.

However, it is much more difficult to develop correct programs with arrays than it is with vectors. Any integer value can be used to subscript an array, even if the value doesn't represent a valid array location. In some languages (e.g., Java), indexing values that do not represent valid array locations cause a program to generate an error message, which can be used to trace the program's behavior. In C and C++, on the other hand, an invalid subscript value can cause unexpected behavior that can lead to hard-to-find errors. Such invalid subscripts are not checked before being used to index a built-in array. Using the `tvector` class rather than the built-in array type provides some safety when using indexed variables, because indices are checked with the `tvector` class. Arrays in C++ have several properties that, at best, might be described as idiosyncratic, and at worst, are a programmer's nightmare.

There are three reasons to study arrays in addition to vectors.

- If you read programs written by other people you'll probably see lots of array code.
- Arrays are more low-level so can offer some performance gains, though the built-in vector class (which has no range checking) should be just as fast with any reasonable implementation.
- It's easier to initialize an array than it is to initialize a vector.

8.4.1 Defining an Array

In C++ the size of an array must be specified by an expression whose value can be determined at compile time. The three statements below on the left define two arrays: one named `numList`, capable of storing 200 values of type `double`, and one named `names` that can store 50 `string` values. Corresponding `tvector` definitions are given on the right.

```
const int SIZE = 100;           const int SIZE = 100;
double numList[SIZE*2];       tvector<double> numList(SIZE*2);
string names[SIZE/2];         tvector<string> names(SIZE/2);
```

In contrast, the following definition of `numList` is illegal according to the C++ standard, because the value of `size` must be determined at compile time but here is known only at

run time. Nevertheless, some compilers may permit such definitions, and in Chapter 12 we will see how to define in a legal manner an array whose size is not known at compile time. There is no compile-time limit on the size of `tvector` variables—only on built-in array variables.

```
int size;
cout << "enter size ";
cin >> size;
double numList[size];           // not legal in standard C++
```

8.4.2 Initializing an Array

Arrays can be initialized by assigning values to the individual array locations, using a loop. It is also possible to assign values to individual array locations when an array is defined. For example, the following definitions assign values representing the number of days in each month to `monthDays` and the names of each month to `monthNames`:

```
int monthDays[13] = {0, 31, 28, 31, 30, 31, 30,
                    31, 31, 30, 31, 30, 31};
string monthNames[13] = {"", "January", "February", "March",
                        "April", "May", "June", "July",
                        "August", "September", "October",
                        "November", "December"};
```

Given these definitions, it's possible to print the names of all the months, in order from January to December, and how many days are in each month, with the following loop.

```
for(k=1; k <= 12; k++)
{ cout << monthNames[k] << ", " << monthDays[k]
  << " days" << endl;
}
```

This kind of initialization is *not* possible with `tvector` variables—only with variables defined using built-in arrays. Note that the zeroth location of each array is unused, so that the k th location of each array stores information for the k th month rather than storing information for March in the location 2. Again, the conceptual simplicity of this scheme more than compensates for an extra array location.

Although the number of entries in each array (13) is specified in the definitions above, this is not necessary. It would be better stylistically to define a constant `const int NUM_MONTHS = 12`, and use the expression `NUM_MONTHS + 1` in defining the arrays, but no number at all needs to be used, as follows:

```
int monthDays [] = {0, 31, 28, 31, 30, 31, 30,
                   31, 31, 30, 31, 30, 31};
string dayNames [] = {"Sunday", "Monday", "Tuesday",
                     "Wednesday", "Thursday", "Friday",
                     "Saturday"};
```

The definition for `dayNames` causes an array of seven strings to be allocated and initialized. The definition of `monthDays` allocates and initializes an array of 13 integers. Since the compiler can determine the necessary number of array locations (essentially by counting commas in the list of values between curly braces), including the number of cells is allowed but is redundant and not necessary.

It is useful in some situations to assign all array locations the value zero as is done in Program 8.2. This can be done when the array is defined, by using initialization values as in the preceding examples, but an alternative method for initializing all entries in an array to zero follows:

```
int diceStats[9] = {0};
```

The `int` array `diceStats` has 9 locations, all equal to 0. When zero is used to initialize all array locations, the number of locations in the array is *not* redundant as it is in the earlier examples, because there is no comma-separated list of values that the compiler can use to determine the number of array values. This method *cannot* be used to initialize arrays to values other than zero. The definition

```
int units[100] = {1};
```

results in an array with `units[0] == 1`, but all other locations in `units` are zero. When a list of values used for array initialization doesn't have enough values, zeros are used to fill in the missing values. This is essentially what is happening with the shortcut method for initializing an array of zeros. I don't recommend this method of initialization; it leads to confusion, because zero is treated differently from other values.

In contrast, `tvector` variables can be initialized so that all entries contain any value, not just zero. This can be done using the two-parameter `tvector` constructor.

8.4.3 Arrays as Parameters

Arrays are fundamentally different from other types in C++ in two ways:

1. It is *not* possible to assign one array to another using an assignment operator `=`.
2. An array passed as a parameter is *not* copied; it is as though the array were passed by reference.

The reason for these exceptions to the normal rules of assignment and parameter passing in C++ (which permit assignment between variables of the same type and use call-by-value for passing parameters) is based on what an array variable name is: a *constant* whose value serves as a *reference* to the first (index 0) item in the array. Since constants cannot be changed, assignments to array variables are illegal:

```
int coins[] = {1,5,10,25};
int bills[] = {1,5,10,20};

coins = bills;           // illegal in C and C++
coins[3] = bills[3];    // legal, assigning to array location
```

Because the array name is a reference to the first array location, it can be used to access the entire contents of the array, with appropriate indexing. Only the array name is passed as the value of a parameter, but the name can be used to change the array's contents even though the array is not explicitly passed by reference. When an array is passed as a parameter, empty brackets [] are used to indicate that the parameter is an array. The number of elements allocated for the storage associated with the array parameter does not need to be part of the array parameter. This is illustrated in Program 8.8

Program 8.8 `fixlist.cpp`

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

// illustrates passing arrays as parameters

void Change(int list[], int numElts);
void Print(const int list[], int numElts);

int main()
{
    const int SIZE = 10;

    int numbers[SIZE];
    int k;
    for(k=0; k < SIZE; k++){
        numbers[k] = k+1;
    }

    cout << "before" << endl << "——" << endl;
    Print(numbers,SIZE);

    cout << endl << "after" << endl << "——" << endl;
    Change(numbers,SIZE);
    Print(numbers,SIZE);
    return 0;
}

void Change(int list[], int numElts)
// precondition: list contains at least numElts cells
// postcondition: list[k] = list[0] + list[1] + ... + list[k]
//                for all 0 <= k < numElts
{
    int k;
    for(k=1; k < numElts; k++)
    { list[k] += list[k-1];
    }
}

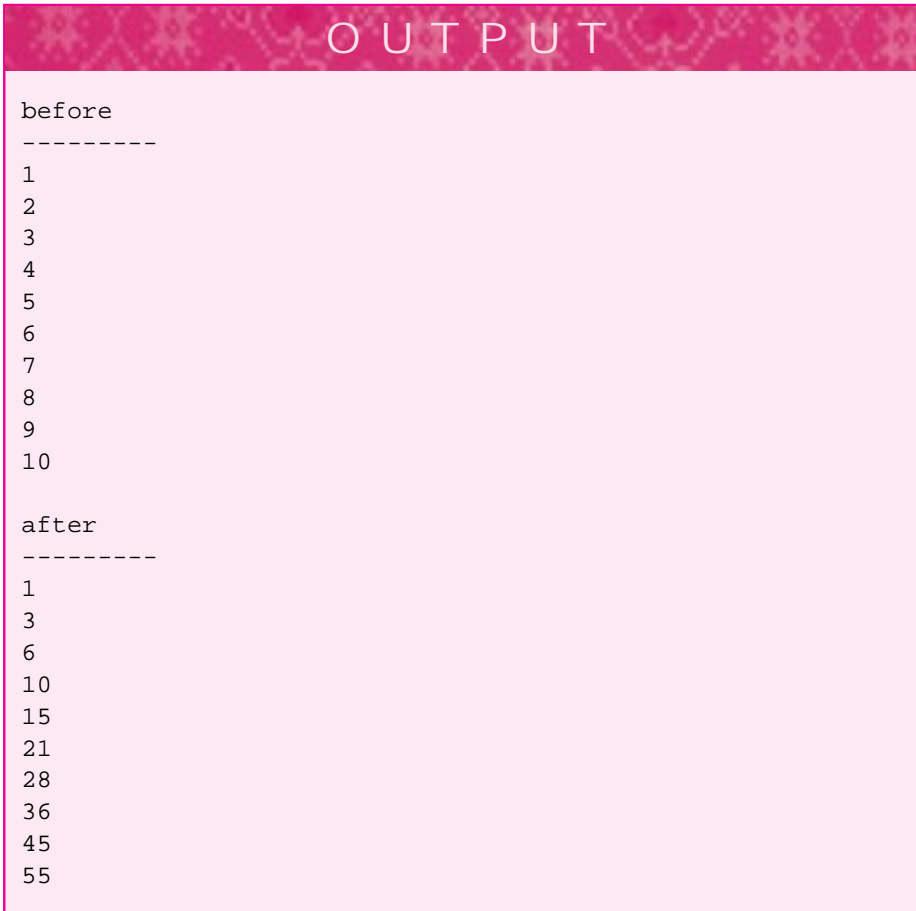
void Print(const int list[], int numElts)
// precondition: list contains at least numElts cells
// postcondition: all elements of list printed
{
```

384

Chapter 8 Arrays, Data, and Random Access

```
int k;  
for(k=0; k < numElts; k++)  
{   cout << list[k] << endl;  
  }  
}
```

fixlist.cpp



The screenshot shows the output of a program. At the top, the word "OUTPUT" is displayed in white capital letters on a dark red background. Below this, the output is divided into two sections: "before" and "after". Each section is preceded by a dashed line. The "before" section lists the numbers 1 through 10. The "after" section lists the numbers 1, 3, 6, 10, 15, 21, 28, 36, 45, and 55.

The identifier `numbers` is used as the name of an array; its value is the location of the first array cell (which has index zero). In particular, `numbers` does *not* change as a result of being passed to `Change()`, but the *contents* of the array `numbers` do change. This is a subtle distinction, but the array name is passed by value, as are all parameters by default in C and C++. The name is used to access the memory associated with the array, and the values stored in this memory can change. Since it is not legal to assign a new value to an array variable (e.g., `list = newList`), the parameter `list` cannot be changed in any case, although the values associated with the array cells can change.

Program Tip 8.6: An array name is like a handle that can be used to grab all the memory cells allocated when the array is defined. The array name cannot be changed, but it can be used to access the memory cells so that they can be changed.

const Parameters. The parameter for the function `Print` in Program 8.8 is defined as `const` or a constant array. The values stored in the cells of a constant array *cannot* be changed; the compiler will prevent attempts to do so. The values stored in a `const` array can, however, be accessed, as is shown in `Print`. If the statement `list[k] = 0` is added in the while loop of `Print`, the `g++` compiler generates the following error message:

```
fixlist.cpp: In function 'void Print(const int *, int)':
fixlist.cpp:46: assignment of read-only location
```

```
Compilation exited abnormally with code 1
at Sat Jun 4 14:02:18
```

Program Tip 8.7: Using a `const` modifier for parameters is good, defensive programming—it allows the compiler to catch inadvertent attempts to modify a parameter. A `const` array parameter protects the values of the array cells from being modified.

Array Size as a Parameter. The number of elements in an array parameter is *not* included in the formal parameter. As a result, there must be some mechanism for determining the number of elements stored in an array parameter. This is commonly done by passing this value as another parameter, by using a global constant, by using the array in a class that contains the number of entries, or by using a sentinel value in the array to indicate the last entry. As an example, the following function `Average` returns the average of the first `numScores` test scores stored in the array `scores`.

```
double Average(const int scores, int numScores)
// precondition: numScores = # of entries in scores
// postcondition: returns average of
//                scores[0] ... scores[numScores-1]
{
    int total = 0;
    double average = 0.0;           // stores returned average
    int k;
    for(k=0; k < numScores; k += 1)
    {    total += scores[k];
```

```

    }

    if (numScores != 0) // guard divide by zero
    {
        average = double(total)/numScores;
    }
    return average;
}

```

Some other section of code might read numbers, store them in an array, and call the function `Average` to compute the average of the numbers read. Alternatively, the numbers stored in the formal parameter `scores` might be data generated from a simulation or some other computer program.

In the following program segment, numbers representing grades (for students in a hypothetical course) are read until the input is exhausted or until the number of grades would exceed the capacity of the array. The average of these grades is then calculated using the function `Average`.

```

const int MAX_GRADES = 100; // maximum # of grades
int grades[MAX_GRADES];
int numGrades = 0; // # of grades entered

while (cin >> score && numGrades < MAX_GRADES)
{
    grades[numGrades] = score;
    numGrades++;
}
cout << "average grade = " << Average(grades, numGrades)
    << endl;

```

This example is meant to illustrate how an array might be used. The approach of storing the grades in an array to calculate the average is not a good one. Because the size of an array is determined at compile time, the code in this example is limited to manipulating at most `MAX_GRADES` grades. Since it is possible to calculate the average of a set of numbers without storing all the numbers, the approach used above is unnecessarily limiting.

The value of `numScores` in `Average` is exactly the number of values stored in the array `scores` but is one more than the largest index of an array cell with a valid value. This off-by-one difference is potentially confusing, so be careful in writing loops that access all the elements in an array.

8.5 Chapter Review

We studied the vector class `tvector` used in place of built-in arrays to store collections of values accessible by random access. Vectors can store thousands of values, and the fifth, five-hundredth, and five-thousandth values can be accessed in the same amount of time. Vectors and their built-in counterparts, arrays, are very useful in writing programs that store and manipulate large quantities of data.

The important topics covered include the following:

- tvecs can be used as counters, for example to count the number of occurrences of each ASCII character in a text file or the number of times a die rolls each number over several trials.
- tvecs are constructed by providing the size of the vector (the number of elements that can be stored) as an argument to the constructor. Vectors are indexed beginning at zero, so a six-element vector has valid indices 0, 1, 2, 3, 4, 5.
- tvecs can be grown by client programs using `resize` or can grow themselves when elements are added using `push_back`. Client programs should double the size when a vector is grown as opposed to growing the size by adding one element.
- When using `push_back`, vectors should be constructed without specifying a size, though space can be allocated using `tvector::reserve`.
- tvecs of all built-in types can be defined, and vectors of programmer-defined types (like `string`) can be defined if the type has a default constructor.
- tvecs can be initialized to hold the same value in every cell by providing a second argument to the constructor when the vector is defined.
- tvecs should always be passed by reference to save memory and the time that would be required to copy if pass by value were used. There are occasions when a copy is needed, but in general pass by reference is preferred. Use `const` reference parameters to protect the parameter from being altered even when passed by reference.
- Initializer lists should be used to construct vectors that are private data members of class objects.
- The function `pop_back` removes the last element of a vector and decreases by one the size of the vector.
- Sequential search is used to find a value in an unsorted vector. Binary search can be used to find values in sorted vectors. Binary search is much faster, needing roughly 20 comparisons to find an item in a list of one million different items. The drawback of binary search is that its use requires a sorted vector.
- Insertion and deletion in a sorted vector requires shifting elements to the right and left, respectively.
- Built-in arrays are cumbersome to use but may be more efficient than vectors. Nevertheless, you should use vectors and switch to arrays only when you've determined that speed is essential and that the use of vectors is making your program slow (which is probably not the case).
- Built-in arrays can be initialized with several values at once. Built-in arrays cannot be resized, cannot be assigned to each other, and do not support range-checked indexing. The size of a built-in array must be known at compile time (although we'll see in Chapter 12 that an alternative form of array definition does permit array size to be determined at run time).

8.6 Exercises

- 8.1** Modify Program 8.3, *letters.cpp*, so that a vector of 26 elements, indexed from 0 to 25, is used to track how many times each letter in the range 'a'–'z' occurs. To do this, map the character 'a' to 0, 'b' to 1,..., and 'z' to 25. Isolate this mapping in a function `CharToIndex` whose header is

```
int CharToIndex(char ch)
// pre: 'a' <= ch and ch <= 'z'
// post: returns 0 for 'a', 1 for 'b', ... 25 for 'z'
```

Note that 'a' - 'a' == 0, 'b' - 'a' == 1, and 'z' - 'a' == 25.)

- 8.2** Write a program that maintains an inventory of a CD collection, a book collection, or some other common collectible. Model the program on *stocks.cpp*, Program 8.6, but instead of implementing a class `Portfolio`, implement a class called `CDCollection`, for example.

The user of the program should have the choice of printing all items, deleting items given an identification number, or artist, searching for all work by a particular artist, reading data from a file and saving data to a file. The data file *cd.dat* that comes with the on-line materials for this book contains thousands of CD entries. For example, the lines below show information for five CDs: an id, the price, the group, and the name of the CD/album.

```
100121 : 15.98 : R.E.M. : Automatic for the People
100122 : 14.98 : Happy Mondays : Yes, Please
100126 : 14.98 : 10,000 Maniacs : Our Time In Eden
100127 : 11.98 : Skid Row : B-Side Ourselves
```

You won't be able to read a file in this format using the extraction operator `>>` because the artist and title contain whitespace. To read these you'll need to use the function `getline` discussed in Chapter 9. The loop below shows how to read a file in the format above, and store the information in a struct `CD`. The code is very similar to the function `Portfolio::Read` from *stocks.cpp*.

```
void CDCollection::Read(const string& filename)
{
    ifstream input(filename.c_str());
    string idnum, price, group, title;

    while (getline(input, idnum, ':') &&
           getline(input, price, ':') &&
           getline(input, group, ':') &&
           getline(input, title, '\n'))
    {
        myCDs.push_back(CD(idnum, atof(price),
                           group, title));
    }
}
```

- 8.3** Modify the class `RandomWalk` found in *walk.h*, Program 7.11 so that the one-dimensional walker keeps track of how many times it visits every position in the range -100 to 100.

You can either use one `tvector` with 201 elements or two `tvector` instance variables: one for nonnegative positions and one for negative positions. A `RandomWalk` object should also keep track of how many times it goes outside the `[-100..100]` range. You'll need to add one or more member functions to get or print the data kept about how many times each position is visited. The simplest approach is to add a method `PrintStats` to print the data. Alternatively you could return a vector of statistics to client programs. You'll need to think carefully about how to verify that the program is tracking visits properly.

For an extra challenge, keep track of every position visited, not just those in the range `[-100..100]`. You'll need to grow the vector(s) that keep track of visits to do this.

- 8.4** Write a program to implement the guess-a-number game described in Section 8.3.7 on binary search. The user should think of a number between 1 and 100 and respond to guesses made by the computer. Make the program robust so that it can tell whether the user cheats by providing inconsistent answers.

O U T P U T

```
prompt> guessnum
Think of a number between 1 and 100 and I'll guess it.

Is the number 50 [y/n]? no
Is the number less than 50 [y/n]? no
Is the number 75 [y/n]? no
Is the number less than 75 [y/n]? no
Is the number 87 [y/n]? no
Is the number less than 87 [y/n]? yes
Is the number 81 [y/n]? yes
I guessed your number using 7 questions.
```



You'll find it useful to call the function `PromptYesNo` in `prompt.h` (see Program G.1 in Howto G.)

- 8.5** Write a program that reads a text file and keeps track of how many time each of the different words occur. A `StringSet` object can keep track of the different words, but the program needs to keep track of how many times each word occurs. There are several ways you might solve this problem; one is outlined below.

Create a struct containing a word and a count of how many times the word occurs. Each time a word is read from the file, it is looked up in a vector of these structs. If the word has been seen before, the word's count is incremented, otherwise the word is added with one occurrence.

- 8.6** Design and implement a `Histogram` class for displaying quantities stored in a `tvector`. A **histogram** is like a bar graph that displays a line relative to the size of the data being visualized. You can construct a `Histogram` object from a vector, and use the vector as a source of data that generates the histogram. For example, the results of using `letters.cpp`, Program 8.3, to find occurrences of each

letter in *Hamlet* can be displayed as a histogram as follows:

```

                                O U T P U T
prompt> letters
enter name of input file: hamlet
a ( 9950 ) *****
b ( 1830 ) ****
c ( 2606 ) *****
d ( 5025 ) *****
e ( 14960 ) *****
f ( 2698 ) *****
g ( 2420 ) *****
h ( 8731 ) *****
i ( 8511 ) *****
j ( 110 )
k ( 1272 ) ***
l ( 5847 ) *****
m ( 4253 ) *****
n ( 8297 ) *****
o ( 11218 ) *****
p ( 2016 ) *****
q ( 220 )
r ( 7777 ) *****
s ( 8379 ) *****
t ( 11863 ) *****
u ( 4343 ) *****
v ( 1222 ) ***
w ( 3132 ) *****
x ( 179 )
y ( 3204 ) *****
z ( 72 )

```

The absolute counts for each letter are shown in parentheses. The bars are scaled so that the longest bar (for the letter *e*) has 40 asterisks and the other bars are scaled relative to this. For example, the letter *h* has 23 asterisks and $8731/14960 \times 40 = 23.32$ (where we divide using double precision, but truncate the final result to an integer).

Member functions for the Histogram class might include setting the length of the longest bar, identifying labels for each bar drawn, plotting a range of values rather than all values, and grouping ranges; for example, for plotting data in the range 0–99, you might group by tens and plot 0–9, 10–19, 20–29, ..., 90–99.

It's difficult to write a completely general histogram class, so you'll need to decide how much functionality you will implement. The following histogram tracks 10,000 rolls of two six-sided dice and scales the longest bar to 40 characters:

It's harder to get labels drawn well for the vertical histogram, so first try to determine how to draw the bars and don't worry initially about the labels.

8.8 Implement a *Sieve of Eratosthenes* to find prime numbers. A sieve is implemented using a `tvector` of `bool` values, initialized so that all elements are `true`. To find primes between 2 and N , use `tvector` indices 2 through N , so you'll need an $(N + 1)$ -element `tvector`.

1. Find the first entry that is `true` (initially this entry has index 2, because 0 and 1 do not count in the search for primes). We'll call the index of the `true` entry p , since this entry will be prime.
2. Set each entry whose index is a multiple of p to `false`.
3. Repeat until all `tvector` elements have been examined.

The process is illustrated in Figure 8.6 for the numbers 2 through 18. Circled numbers are true. In the topmost view of the array the first `true` cell has index 2, so all the even numbers (multiples of 2) are changed to `false`. These are shown as shaded entries in the diagram. The next `true` value is 3, so all multiples of 3 are changed to `false` (although 6, 12, and 18 have already been changed). In the third row no more new entries will be set to `false` that are not already false, and the primes have been determined (although the steps are repeated until all `tvector` elements have been examined).

8.9 Write a program that keeps track of important dates/events and reminds you of all the important dates that occur in the next two weeks each time you run the program. For example, you can store events in a data file as follows:

```
04 01 April Fools Day
02 08 Mom's birthday
01 01 New Year's Day
07 16 Laura's birthday
11 22 Margaret's birthday
```

To read data in this format you'll need to use the `getline` function from Chapter 9 to read all the words on a line after the month and day. The code below reads an `ifstream` named `input` in this format and prints all the events.



Figure 8.6 Using a Sieve of Eratosthenes to find primes

```

int month, day;
string event;
while (input >> month >> day &&
      getline(input, event)
{
    Date dday(month, day, 1999);
    cout << event << " occurs on " << dday << endl;
}

```

The program should prompt the user for a time frame, like one day, or one week, or 15 weeks, and print all the events that occur within that time frame of the current day (the day on which the program is run). See `date.h`, Program G.2 in *Howto G* for a review of the `Date` class.

You should allow the user the option of printing all the events in chronological order.



- 8.10** Write a program that determines the frequently used words in a text file. We'll define **frequently used** to mean that a word accounts for at least 1% of all the words in a file. For example, in the file *melville.txt* there are 14,353 total words, so any word that occurs more than 143 times is a frequently used word. For *melville.txt* the frequently used words are shown in Table 8.2 with their number of occurrences.

Table 8.2 Frequent words in *melville.txt*

334	a	196	my
376	and	151	not
218	he	359	of
219	his	194	that
519	i	603	the
265	in	432	to
164	it	195	was