



# Recursion, Lists, and Matrices 10

Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what detail one can do without and yet preserve the spirit of the whole—so that all that one has suppressed and cut away is there to the reader's consciousness as much as if it were type on the page.

Will a Cather  
*On the Art of Fiction*

---

In this chapter we focus on **recursion**, a technique for structuring functions and classes that helps solve self-referential problems. We'll also study two classes that structure data: a self-referential structure called a **list** and an extension of the `tvector` class, called `tmatrix`, that represents **two-dimensional data**. In studying these structures we'll also explore properties of objects in a program that relate to how and where the objects can be accessed. We'll see two important properties of objects: **scope**, where the object can be accessed, and **lifetime**, the duration of an object during program execution. We'll see that recursive functions seem to "call themselves," but they are better understood as functions that solve problems whose solution can be expressed by combining solutions to problems that are similar, but smaller. Some problems have terse and comprehensible solutions expressed as recursive functions but have convoluted nonrecursive solutions. Other problems seem to be suitable for recursive solution but are better solved nonrecursively.

## 10.1 Recursive Functions

As a first example of a problem whose solution is elegantly expressed using recursion, we turn to the problem of outputting an English version of an integer by printing each digit's spelled-out English equivalent. For example, 1053 should be output as "one zero five three." We solved this problem with *digits.cpp*, Program 5.5, using `string` concatenation. Now we limit ourselves to a solution using only `int` variables. To make the problem simpler, we'll initially limit the input to four-digit numbers. However, the recursive solution will work for all `int` values.

### 10.1.1 Similar and Simpler Functions

When we solved this problem using strings, we concatenated digits to the front of a string as it was built up from each digit in an `int`. To convert 123, we first concatenated "three" to an empty string called `s`. Then we concatenated "two" to the front of `s`, forming "two three". Concatenating "one" to the front of `s` now yields the desired string (see *digits.cpp*, Program 5.5). Basically, we peeled the number's digits

450

## Chapter 10 Recursion, Lists, and Matrices

from the right, concatenating them to the string from the left. Since we aren't using string functions, we must rewrite the program to print string literals for each digit of an int. This is done in *digits2.cpp*, Program 10.1.

---

**Program 10.1** *digits2.cpp*

---

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

// prelude to recursion: print English form of each digit
// in an integer: 123 -> "one two three"

void PrintDigit(int num)
// precondition: 0 <= num < 10
// postcondition: prints english equivalent, e.g., 1->one,...9->nine
{
    if (0 == num)      cout << "zero";
    else if (1 == num) cout << "one";
    else if (2 == num) cout << "two";
    else if (3 == num) cout << "three";
    else if (4 == num) cout << "four";
    else if (5 == num) cout << "five";
    else if (6 == num) cout << "six";
    else if (7 == num) cout << "seven";
    else if (8 == num) cout << "eight";
    else if (9 == num) cout << "nine";
    else cout << "?";
}

void PrintOne(long number)
// precondition: 0 <= number < 10
// postcondition: prints English equivalent of number
{
    if (0 <= number && number < 10)
    {
        PrintDigit(number);
    }
}

void PrintTwo(long int number)
// precondition: 10 <= number < 100
// postcondition: prints English equivalent of number
{
    if (10 <= number && number < 100)
    {
        PrintOne(number / 10);
        cout << " ";
        PrintDigit(number % 10);
    }
}

void PrintThree(long int number)
// precondition: 100 <= number < 1000
// postcondition: prints English equivalent of number
```

```
{
    if (100 <= number && number < 1000)
    {
        PrintTwo(number / 10);
        cout << " ";
        PrintDigit(number % 10);
    }
}

void PrintFour(long int number)
// precondition: 1000 <= number < 10,000
// postcondition: prints English equivalent of number
{
    if (1000 <= number && number < 10000)
    {
        PrintThree(number / 10);
        cout << " ";
        PrintDigit(number % 10);
    }
}

int main()
{
    int number = PromptRange("enter an integer",1000,9999);
    PrintFour(number);
    cout << endl;

    return 0;
}
```

---

`digits2.cpp`

## OUTPUT

```
prompt> digits2
enter an integer between 1000 and 9999: 8732
eight seven three two
prompt> digits2
enter an integer between 1000 and 9999: 7003
seven zero zero three
prompt> digits2
enter an integer between 1000 and 9999: 1000
one zero zero zero
```

The function `PrintFour` prints a four-digit number. We know how to peel the last digit from a number using the modulus and division operators, `%` and `/`. In *digits2.cpp*, a four-digit number is printed by printing the first three digits using the function `PrintThree`, then printing the final digit using the function `PrintDigit`. For example, to print 1357 we first print 135, which is  $1357/10$ , by calling `PrintThree`, and then print "seven", the last digit of 1357 obtained using  $1357\%10$ . Printing a three-digit number is a similar process: first print a two-digit number by calling `PrintTwo`,

and then print the last digit. For example, to print 135 we first print 13, which is  $135/10$ , and then print "five", which is  $135\%10$ . Continuing with this pattern we call `PrintOne` and `PrintDigit` to print a two-digit number. Finally, to print a one-digit number we simply print the only digit.

The code in *digits2.cpp* should offend your emerging sense of programming style. Each of the functions `PrintFour`, `PrintThree`, and `PrintTwo` are virtually identical except for the name of the function, `PrintXXXX`, that each one calls (e.g., `PrintThree` calls `PrintTwo`). We can combine the similar code in all the `PrintXXXX` functions. Rather than using four separate functions, each one processing a certain range of numbers, we can rewrite the nearly identical functions as a single function `Print`. This is shown in *digits3.cpp*, Program 10.2.

---

Program 10.2 *digits3.cpp*

---

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

// recursion: print English form of each digit
// in an integer: 123 -> "one two three"

void PrintDigit(int num)
// precondition: 0 <= num < 10
// postcondition: prints english equivalent, e.g., 1->one,...9->nine
{
    if (0 == num)        cout << "zero";
    else if (1 == num)   cout << "one";
    else if (2 == num)   cout << "two";
    else if (3 == num)   cout << "three";
    else if (4 == num)   cout << "four";
    else if (5 == num)   cout << "five";
    else if (6 == num)   cout << "six";
    else if (7 == num)   cout << "seven";
    else if (8 == num)   cout << "eight";
    else if (9 == num)   cout << "nine";
    else cout << "?";
}

void Print(long int number)
// precondition: 0 <= number
// postcondition: prints English equivalent of number
{
    if (0 <= number && number < 10)
    {   PrintDigit(int(number));
    }
    else
    {   Print(number / 10);
        cout << " ";
        PrintDigit(int(number % 10));
    }
}
```

```
int main()
{
    long number = PromptRange("enter an integer",0L,1000000L);
    Print(number);
    cout << endl;

    return 0;
}
```

---

`digits3.cpp`

## OUTPUT

```
prompt> digits3
enter an integer between 1 and 1000000: 13
one three
prompt> digits3
enter an integer between 1 and 1000000: 7
seven
prompt> digits3
enter an integer between 1 and 1000000: 170604
one seven zero six zero four
```

The `if` statement in `Print` from `digits3.cpp` corresponds to the equivalent `if` in the function `PrintOne` from the previous program, `digits2.cpp`. A number in the range 0–9 is simply printed by calling `PrintDigit`. In all other cases, the code in the body of the `else` statement in `Print` from `digits3.cpp`, Program 10.2, is the same code in the functions from `digits2.cpp`, Program 10.1.

Although you may think that the function `Print` is calling itself in `digits3.cpp`, it is not. As shown in Figure 10.1, four separate functions named `Print` are called when the user enters 1478. These functions are identical except for the value of the parameter `number` stored in each function. The first `Print`, shown in the upper-left corner of Figure 10.1, receives the argument 1478 and stores this value in `number`. Since the value of `number` is greater than 10, the `else` statements are executed. A function `Print` is called with the argument  $1478 / 10$ , which is 147. This is not the same function as in the upper left, but another version of the function `Print`, in essence a **clone function** of `Print`, except that the value of `number` is different. Altogether there are four clones of the `Print` function, each with its own parameter `number`. The last clone called (lower-right corner of Figure 10.1) does not generate another `Print` call, since the value of `number` is between 0 and 9. The `if` statement is executed, and the function `PrintDigit` is called with the argument 1. It's important to realize that this is the first call of `PrintDigit`, so the first digit printed is “one.” Although each clone executes the statement `PrintDigit(number % 10)`, this statement is executed only after the recursive clone function call to `Print`. Each clone waits for control to return from the recursive call, except for the last function, which doesn't make

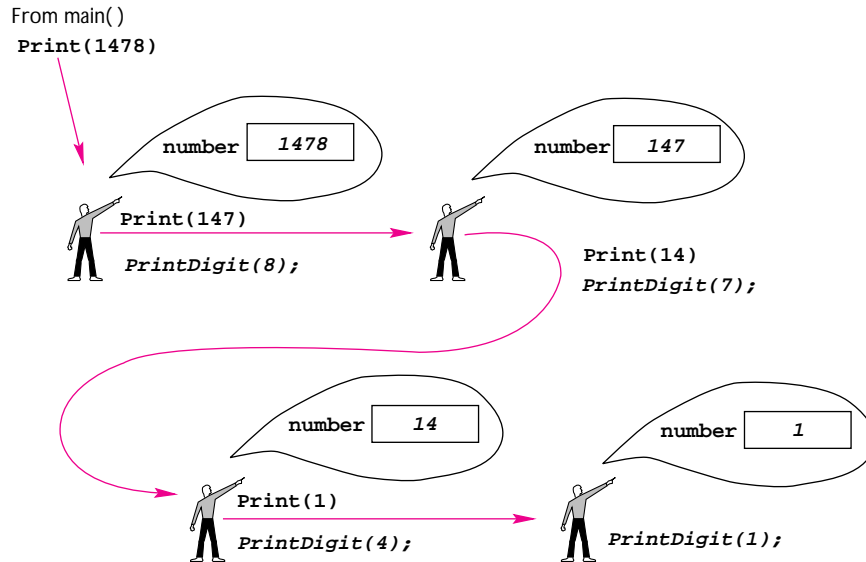


Figure 10.1 Recursively printing digits.

a recursive call. The first clone called is the last clone to print a digit, so the last digit printed is  $1478 \% 10$ , which is 8. This means the last word printed is “eight.”

You’ll need to develop two skills to understand recursive functions.

1. The ability to reason about a recursive function so that you can determine what the function does.
2. The ability to think recursively so that you can write recursive functions to solve problems.

Developing the second skill is more difficult than the first, but practice with reasoning about recursive functions will help with both skills.

### 10.1.2 General Rules for Recursion

When you write a loop, you reason about when the loop will stop executing so that you don’t write an infinite loop. You must take the same care when writing recursive functions to avoid an infinite succession of recursively called clones. Each clone uses space, so you won’t be able to actually generate an infinite number of clones, but you can easily use up all the memory in your computing environment if you’re not careful. To avoid an infinite chain of recursive calls, each recursive function must include a **base case** that does not make a recursive call. The base case in `Print` of `digits3.cpp`, Program 10.2, is a single-digit number identified by this test:

```
if (0 <= number && number < 10)
```

A function's base case is usually determined by finding a value, or a set of values, that does not require much work to compute. We'll look at a recursive version of the function to raise a number to a power that we studied in Section 5.1.7.

If you're asked to calculate  $3^8$ , you could multiply  $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$ . You could also calculate  $3^4 = 81$  and then calculate  $81 \times 81 = 6561$ , since  $3^8 = 3^4 \times 3^4$ . The second method uses far fewer multiplications to calculate  $a^n$  than the first. The method is summarized in the following (repeated from Section 5.1.7.)

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a^{n/2} \times a^{n/2} & \text{if } n \text{ is even} \\ a \times a^{n/2} \times a^{n/2} & \text{if } n \text{ is odd (note that } n/2 \text{ truncates to an integer)} \end{cases} \quad (10.1)$$

For example, to calculate  $4^{11}$  using this method, we first calculate  $4^{11/2} = 4^5 = 1024$  and then multiply  $4 \times 1024 \times 1024 = 4,194,304$ . The base case requires no power calculation and no recursion. The base case in the formula corresponds to an exponent of zero. For nonzero exponents, the recursion comes from the calculation of  $a^{n/2}$  in the formula. We'll write a function `Power` with two parameters: one for the base  $a$  and one for the exponent  $n$  in calculating  $a^n$ . Note that there is one recursive call and the value returned by the call is stored in a local variable `semi`:

```
double Power(double base, int expo)
// precondition: expo >= 0
// postcondition: returns base^expo
{
    if (0 == expo)
    { return 1.0; // correct for zeroth power
    }
    else
    { double semi = Power(base, expo/2);
      if (expo % 2 == 0) // even exponent
      { return semi*semi;
      }
      else // odd exponent
      { return base*semi*semi;
      }
    }
}
```

The calculation of  $2^{35}$  using `Power(2, 35)` generates seven clone `Power` functions with `expo` values 35, 17, 8, 4, 2, 1, 0. Since the recursive call uses `expo/2` as the value of the second argument, the total number of recursive calls is limited by how many times the original argument can be divided in half.

The seven clones are shown in Figure 10.2, where the value of `expo` can be used to determine the sequence of recursive calls. The result of each clone's one recursive call is stored in the calling function's local variable `semi`. The value of `semi` is used to calculate the returned result. Just as each iteration of a loop body changes values so

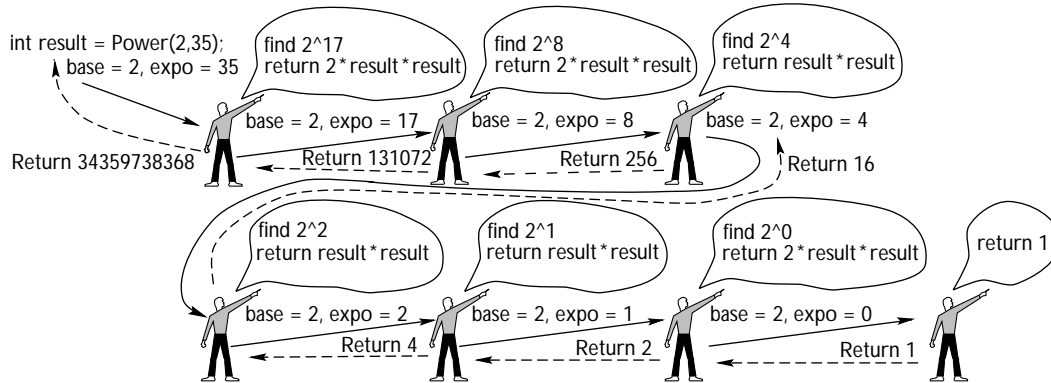


Figure 10.2 Recursively calculating  $2^{35}$ .

that the loop test eventually becomes false and the loop terminates, each recursive call should get closer to the base case. This ensures that the chain of recursively called clones will eventually stop. In general, recursive functions are built from calling similar, but simpler functions. The similarity yields recursion; the simplicity moves toward the base case.

**Program Tip 10.1: Recursive functions must make recursive calls that are similar to the original call, but simpler than the original call.**

1. Identify a base case that does not make any recursive calls. Each call should make progress towards reaching the base case; this ensures termination since the function will end.
2. Solve the problem by making recursive calls that are similar, but simpler, (i.e., that move towards the base case). The similarity ensures that the recursion works, you'll be solving a similar problem.

### 10.1.3 Infinite Recursion

You must guard against writing functions that result in infinite recursion, that is, functions that generate a potentially endless number of recursive calls. When you forget a base case, infinite recursion results, as shown in *recdepth.cpp*, Program 10.3. The output for Program 10.3 came from a Pentium PC with 256 megabytes of memory using Metrowerks Codewarrior; it shows that 36,977 recursive calls are made before memory is exhausted.

---

Program 10.3 recdepth.cpp

---

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

// Owen Astrachan
// illustrates problems with "infinite" recursion

void
Recur(int depth)
{
    cout << depth << endl;
    Recur(depth+1);
}

int main()
{
    Recur(0);
    return 0;
}
```

---

recdepth.cpp

---

**OUTPUT**

```
prompt> recdepth
0
1

some output removed

36977
36977
Unhandled exception: c00000fd
```

The maximum number of clones, or recursive calls, is limited by the memory of the computer used and depends on certain settings of the programming environment. For example, when I used g++ on a Linux machine with 32 megabytes of memory the program crashed with a segmentation fault after 698,911 calls.<sup>1</sup> Using Visual C++ 6.0 yields an exception, with the program halting, after 11,740 calls. As a programmer you must be careful when writing recursive functions. You should always identify a base case that does not make any recursive calls.

---

<sup>1</sup>The recursive call in `Recur` is an example of **tail recursion**. In a tail recursive function the last statement executed is a recursive call. Smart compilers can turn tail recursive functions into looping functions automatically, thus saving memory.

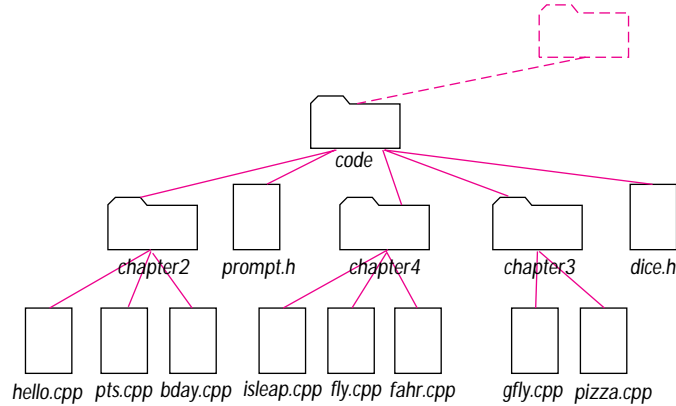


Figure 10.3 Hierarchy of directories and files.

You may study methods in more advanced courses that involve changing a recursive function to a nonrecursive function. This is often a difficult task. Sometimes, however, it is possible to write a simple nonrecursive version of a recursive function. Nevertheless, some functions are much more easily written using recursion; we'll study examples of these functions in the next section.

## 10.2 Recursion and Directories

In this section we'll use recursion to solve problems that cannot be solved without recursion unless auxiliary data structures are used. The recursive functions find information about files and directories stored on disk. Almost all computers have an operating system in which directories help organize the many files you create and use. For example, you may have a directory for each of the computer courses you have taken, a directory for the electronic mail you receive, and a directory for your home page on the World Wide Web. Using directories to organize files makes it easier for you to find a specific file. Directories contain files as well as **subdirectories**, which can also contain files and subdirectories. For example, the hierarchical arrangement of directories enables you to have a `courses` directory in which you have subdirectories for English, computer science, biology, and political science courses. A diagram of some of my directories and files for this book is shown in Figure 10.3. Directories are shown as file folders, and files are shown as rectangles.

In this section we investigate classes that use recursion to process directory hierarchies. For example, we will develop a program that mimics what some operating systems do (and some don't) in determining how much space files use on disk. We will also develop a program that scans a hierarchy of directories to find a file whose name you remember but whose location you have forgotten.

### 10.2.1 Classes for Traversing Directories

Program 10.4, *files.cpp*, prompts the user for the name of a directory and then prints all the files in that directory. This kind of listing is often needed when opening files from within a word processor; you must be able to type or click on the name of the file you want to edit. The variable `dir` is a class `DirStream` object. The `DirStream` class supports iteration using `Init`, `HasMore`, `Next`, and `Current` similarly to other classes, (e.g., the class `WordStreamIterator` from *maxword.cpp*, Program 6.11 and the class `StringSetIterator` from *setdemo.cpp*, Program 6.14).

---

#### Program 10.4 *files.cpp*

---

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

// illustrates use of the DirStream and DirEntry classes

int main()
{
    DirStream dir;           // directory information
    DirEntry entry;        // one entry from a directory
    int num = 0;           // each file is numbered in output

    string name = PromptString("enter name of directory: ");
    dir.open(name);

    if (dir.fail())
    {
        cerr << "could not open directory " << name << endl;
        exit(1);
    }
    for(dir.Init(); dir.HasMore(); dir.Next())
    {
        entry = dir.Current();
        num++;
        cout << "(" << setw(3) << num << " ) " << setw(12) << entry.Name() << "\t";
        if (! entry.IsDir() )
        {
            cout << entry.Size();
        }
        cout << endl;
    }
    return 0;
}
```

---

*files.cpp*

---

```


O U T P U T



```

prompt> files
enter name of directory: c:\book\mcgraw
( 1)      .
( 2)     ..
( 3)  design.pdf      246489
( 4) designspecs.pdf  60876
( 5)   fixreview      24481
( 6)  hromcik.doc     41472
( 7)   hsreviews      59797
( 8)      notes       305
( 9)     photo        1488
(10) schedule.xls     17408
(11)   tapestry       420692
(12)  tapsurv.SIT     15836
prompt> files
enter name of ..\chap22
could not open directory ..\chap22

```


```

The member function `DirStream::Current()` returns a `DirEntry` object. Repeated calls of `Current`, in conjunction with the iterating functions `HasMore` and `Next`, return each entry in the directory. These directory entries are either files or subdirectories. In `files.cpp`, the member function `DirEntry::IsDir()` differentiates files from directories, returning true when the `DirEntry` object is a directory and false otherwise. `DirEntry::Size()` returns the size, in bytes, of a file. On Windows machines this is zero for directories, on Linux/Unix machines directories have nonzero sizes. The filename `.` (a single period) represents the current directory. The filename `..` (a double period) represents the parent directory. This convention is followed by many operating systems. The member functions for the class `DirEntry` and the class `DirStream` are given in `directory.h`, Program G.11 in How to G.

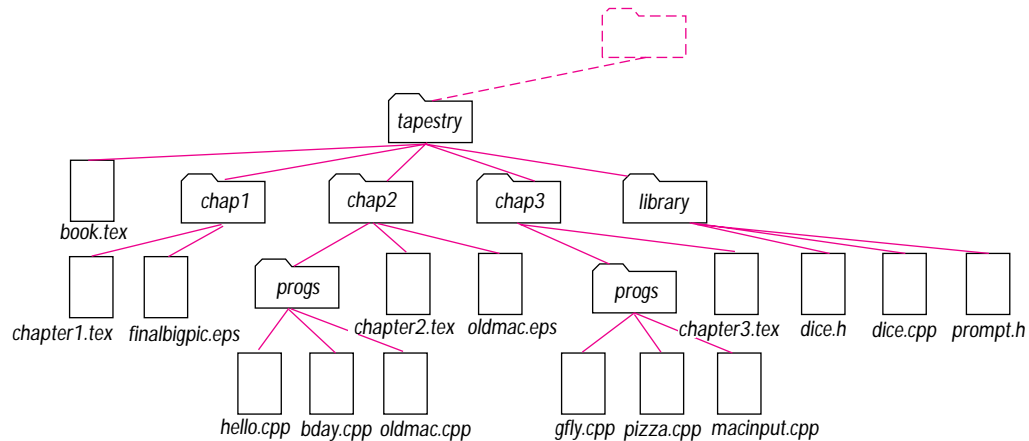


The header file, `directory.h`,<sup>2</sup> that contains declarations for the classes `DirEntry` and `DirStream`, is provided on-line with the code provided with this book.

## 10.2.2 Recursion and Directory Traversal

Program 10.4, `files.cpp`, prints all the files in a given directory. Some applications require lists of subdirectories, and the files within the subdirectories, as well. For example, to calculate the total amount of disk space used by all files and directories, a program must accumulate the sum of file sizes in all subdirectories. We'll modify `files.cpp`, Program 10.4, so that it prints both files and subdirectories (and files and subdirectories of the subdirectories, and so on). As a first step, we'll move the `for` loop that iterates

<sup>2</sup>On some 16-bit systems the file may be named `directry.h`.



**Figure 10.4** Files and subdirectories used in run of *subdir.cpp*, Program 10.5.

over all the files in a directory into a function `ProcessDir`. The final program is *subdir.cpp*, Program 10.5. Figure 10.4 contains a diagram of the files and subdirectories that generate the sample run.<sup>3</sup>

#### Program 10.5 *subdir.cpp*

```

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

#include "directory.h"
#include "prompt.h"

// print all entries in a directory (uses recursion)

void Tab(int count)
// postcondition: count tabs printed to cout
{
    int k;
    for(k=0; k < count; k++)
    {cout << "\t";
    }
}

```

<sup>3</sup>The suffixes in Figure 10.4 represent different kinds of files: `.cpp` for C++ source code, `.tex` for  $\text{\LaTeX}$  files (a document-processing system), `.eps` for PostScript files, and so on.

```

void ProcessDir(const string & path, int tabCount)
// precondition: path specifies pathname to a directory
//               tabCount specifies how many tabs for printing
// postcondition: all files and subdirectories in directory 'path'
//               printed, subdirectories tabbed over 1 more than parent
{
    DirStream indir(path);
    DirEntry entry;
    int num = 0;           // number of files in this directory

    for(indir.Init(); indir.HasMore(); indir.Next())
    {   entry = indir.Current();   // either file or subdirectory

        // don't process self: ".", or parent directory: ".."
        if (entry.Name() != "." && entry.Name() != "..")
        {   num++;
            Tab(tabCount);
            cout << "(" << setw(3)<< num << ")" << "\t" << entry.Name() << endl;
            if (entry.IsDir() )           // process subdir
            {   ProcessDir(entry.Path(),tabCount+1);
                }
            }
        }

int main()
{
    string dirname = PromptString("enter directory name ");
    ProcessDir(dirname,0);
    return 0;
}

```

subdir.cpp

The files in a subdirectory are indented and numbered after the name of the subdirectory is printed. For example, the subdirectory named `chap2` contains one subdirectory, `progs`, and two files, `chapter2.tex` and `oldmac.eps`. The subdirectory `progs` of `chap2` contains three files: `hello.cpp`, `bday.cpp`, and `oldmac.cpp`. The directory `tapestry`, whose name is entered when the program is run, contains four subdirectories: `chap1`, `chap2`, `chap3`, and `library`, and one file: `book.tex`. Notice that the files in a subdirectory are numbered starting from one. We cannot control the order in which files and subdirectories are processed using the `DirStream` iterating functions `Init`, `Next`, and `Current`. For example, the operating system may scan the files alphabetically, ordered by date of creation, or in some random order. However, you can print the files in any order by storing them in a vector and sorting by different criteria.

## O U T P U T

```
prompt> subdir
enter directory name tapestry
( 1)   book.tex
( 2)   chap1
      ( 1)   chapter1.tex
      ( 2)   finalbigpic.eps
( 3)   chap2
      ( 1)   chapter2.tex
      ( 2)   oldmac.eps
      ( 3)   progs
            ( 1)   bday.cpp
            ( 2)   hello.cpp
            ( 3)   oldmac.cpp
( 4)   chap3
      ( 1)   chapter3.tex
      ( 2)   progs
            ( 1)   gfly.cpp
            ( 2)   macinput.cpp
            ( 3)   pizza.cpp
( 5)   library
      ( 1)   prompt.h
      ( 2)   dice.cpp
      ( 3)   dice.h
```

We'll investigate the function `ProcessDir` from *subdir.cpp* in detail. One key to the recursion is an understanding of how a complete filename is specified in hierarchical file systems. Most systems specify a complete filename by including the directories and subdirectories that lead to the file. This sequence of subdirectories is called the file's **pathname**. The subdirectories that are pathname components are separated by different delimiters in different operating systems. For example, in UNIX the separator is a forward slash, so the pathname to `gfly.cpp` shown in the output run of *subdir.cpp* is `tapestry/chap3/progs/gfly.cpp`. On Windows computers the separator is a backslash, so the pathname is `tapestry\chap3\progs\gfly.cpp`. The string used as a separator is specified by the constant `DIR_SEPARATOR` in *directory.h*. The last component in a path is a file's name; it's returned by `DirEntry::Name`. The entire path, including the name, is returned by `DirEntry::Path`. Both of these member functions are used in *subdir.cpp*: one to print the name, and one to recurse on a subdirectory since the entire path is needed to specify a directory.

The `for` loop that iterates over directory entries in the function `ProcessDir` is similar to the loop used in *files.cpp*, Program 10.4. However, when the information stored in the `DirEntry` object `entry` represents a directory, the function `ProcessDir`

makes a recursive using the pathname for the subdirectory. For example, the call `ProcessDir("tapestry", 0)` directly generates four recursive calls for the subdirectories `chap2`, `chap1`, `chap3`, and `library`, as diagrammed in Figure 10.4. The pathname for the subdirectory `chap3` is obtained directly from the `DirEntry` object `entry`, but it can also be formed from the expression

```
path + DIR_SEPARATOR + entry.Name()
```

where, for example, `path` is `"tapestry"` and `entry.Name()` is `"chap3"`. The value of `tabCount` is calculated from the expression `tabCount+1` so that the arguments for the recursive call are

```
ProcessDir("tapestry/chap3", 1);
```

This recursive call will, in turn, generate a recursive call for the subdirectory `progs`.

Examine the output run of *subdir.cpp* on the directory `tapestry`, diagrammed in Figure 10.5. Each clone of the function `ProcessDir` is shown as a figure. The call `ProcessDir(dirname, 0)` from `main` is shown in the upper-left corner of Figure 10.5 as `ProcessDir("tapestry", 0)`; `dirname` has the value `"tapestry"`. Each recursive clone of `ProcessDir` has its own formal parameters `path` and `tabCount` and its own local variables `indir`, `entry`, and `num`. Each recursive clone will print all the files in the subdirectory specified by the clone's `path` parameter. For example, the four clones generated by calls from the upper-left clone of `ProcessDir` are shown with `num` values 1, 2, 3, and 5. When `num` is 4, the file `book.tex` is printed as shown in the output from *subdir.cpp*.

As shown in the output of the program, the files and subdirectories in `tapestry` are processed by `Next` and `Current` in the following order.

1. chap2
2. chap1
3. chap3
4. book.tex
5. library

The first file/subdirectory printed and processed is (1) `chap2`. The number 1 is the value of local variable `num` shown in the stick figure in the upper left corner. The files/subdirectories of `chap2` are shown indented one level. The indentation level is determined by the value of parameter `tabCount`, which is 1 because of the recursive call of `ProcessDir`:

```
ProcessDir(entry.Path(), tabCount+1);
```

The value passed as the second parameter is `tabCount+1`, which in this case is `0+1=1`. Because the value passed is always one more than the current value, each recursive call results in one more level of indentation. The output of *subdir.cpp* shows that the `progs` subdirectory is the second entry printed in the `chap2` directory. The first entry printed is `chapter2.tex`. The recursive call generated by `progs`, shown in Figure 10.5 as

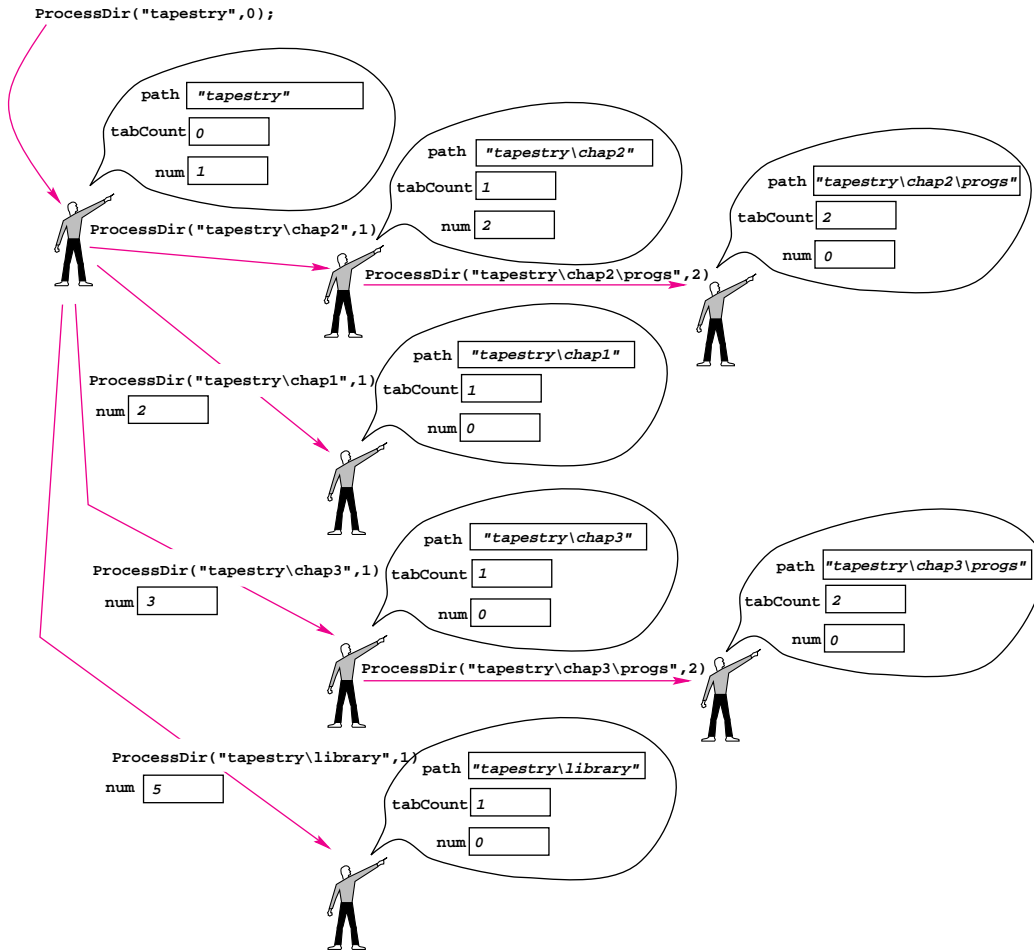


Figure 10.5 Recursive calls/clones for run of `subdir.cpp`, Program 10.5.

the call `ProcessDir("tapestry\chap2\progs", 2)`, shows that `num` has the value two when the call is made reflecting that `progs` is printed as the second entry under `chap2`: (2) `progs`.

Like all functions, the recursively called functions communicate only via passed parameters. There is nothing magic or different in the case of recursively called functions; each function just happens to have the same name as the function that calls it.

### 10.2.3 Properties of Recursive Functions

At most, three clones of function `ProcessDir` exist at one time, as shown in Figure 10.5. The three clones at the top of the figure exist at the same time (with path values of `"tapestry"`, `"tapestry\chap2"`, and `"tapestry\chap2\progs"`). When the recursive call that processes the `chap2\progs` subdirectory finishes executing, the clone with path parameter `"tapestry\chap2"` still has one more entry to process: `oldmac.eps` (see the output). Then this clone finishes executing, and only the first version of `ProcessDir`, invoked by the call `ProcessDir("tapestry", 0)`, exists.

A recursive call for the `chap1` subdirectory is then made. When the clone invoked by the call `ProcessDir("tapestry\chap1", 1)` finishes executing, a recursive call is made for the `chap3` subdirectory. This, in turn, makes a recursive call for the `chap3\progs` subdirectory. Note that at this point the value of `num` for the original `ProcessDir` is 3, as shown in Figure 10.5. Finally, after printing `(4) book.tex`, the subdirectory `library` generates the final recursive call; the value of `num` is 5 as shown.

#### Pause to Reflect



**10.1** Write a function based on `Print` in `digits3.cpp`, Program 10.2, that prints the base two representation of a number. The number 17 in base 2 is 10001 since  $17 = 2^4 + 2^0$ . Just as 5467 in base 10 means  $5 \times 10^4 + 4 \times 10^3 + 6 \times 10^1 + 7 \times 10^0$ , so does 10110 in base 2 mean  $1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$ .

**10.2** The recursive `Power` function makes the recursive call as follows, and squares the return value.

```
double semi = Power(base, expo/2);
...
return semi*semi;
```

It's possible to square `base` in the argument to the recursive call and just return the result as follows.

```
double semi = Power(base*base, expo/2);
...
return semi;
```

Explain why these are equivalent. Which do you think is better? Does your answer change if `BigInt` values are used instead of `double` values? How can you test your answers?

**10.3** Based on the output generated by *subdir.cpp*, Program 10.5 for the directory *tapestry*, what would be the output of the program *files.cpp*, Program 10.4 if run on *tapestry*? (Make up numbers for file size; it's the names of the files that are important in this question.)

**10.4** Why is the `if` statement

```
if (entry.Name() != "." && entry.Name() != "..")
```

used in *subdir.cpp* necessary? Describe what would happen if the comparison with `"."` were removed, but the other comparison remained. What would happen if the comparison with `".."` were removed (but the other remained)?

**10.5** How can you modify *subdir.cpp* to print a list of every file (starting from a directory whose name the user enters) whose size is larger than a number the user enters?

**10.6** How can you modify *subdir.cpp* to print the name of every file containing a word, in either upper or lower case, that the user enters.

**10.7** Describe how the output of *subdir.cpp* will change if the expression `tabCount+1` in the recursive call is replaced with `tabCount+2`.

**10.8** If the call of `Tab` and the `cout << ...` statement in function `ProcessDir` of *subdir.cpp* are moved *after* the `if (entry.IsDir())` statement, how will the output change (e.g., if the directory *tapestry* is used for input)?

## 10.3 Comparing Recursion and Iteration

As an apprentice software engineer and computer scientist you must learn to judge when recursion is the right tool for a programming task. We've already seen that recursion is indispensable when traversing directories. As an apprentice, you should learn part of the programming folklore of recursion. We'll use two common examples to investigate tradeoffs in implementing functions recursively and iteratively.

### 10.3.1 The Factorial Function

In *fact.cpp*, Program 5.2, the function `Factorial` computes the **factorial** of a number where  $n! = 1 \times 2 \times \cdots \times n$ . A loop accumulated the product of the first  $n$  numbers. An alternative version of the factorial function is defined mathematically using this definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases} \quad (10.2)$$

According to the definition,  $6! = 6 \times 5!$ . What, then, is to be done about  $5!$ ? According to the definition, it is  $5 \times 4!$ . This process continues until  $1! = 1 \times 0!$  and  $0! = 1$  by definition. The method of defining a function in terms of itself is called an **inductive**

definition in mathematics and leads naturally to a recursive implementation. The base case of  $0! = 1$  is essential since it stops a potentially infinite chain of recursive calls. As we noted in the first section of this chapter, the base case is often a case that requires little or no computation, such as the calculation of zero factorial, which, by definition, is one. Recursive and iterative versions of the factorial function are included and tested in *facttest.cpp*, Program 10.6. Statements are included to check if the values returned by the recursive and iterative functions are different, but the values returned are always the same when I run the program.

---

Program 10.6 *facttest.cpp*

---

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

BigInt RecFactorial(int num)
// precondition: 0 <= num
// postcondition: returns num! (num factorial)
{
    if (0 == num)
    { return 1;
    }
    else
    { return num * RecFactorial(num-1);
    }
}

BigInt Factorial(int num)
// precondition: 0 <= num
// postcondition: returns num! (num factorial)
{
    BigInt product = 1;
    int count;
    for(count=1; count <= num; count++)
    { product *= count;
    }
    return product;
}

int main()
{
    CTimer rtimer, itimer;
    long j,k;
    BigInt rval,ival;
    long iters = PromptRange("enter # of iterations",1L,1000000L);
    int limit = PromptRange("upper limit on factorial",10,100);

    for(k=0; k < iters; k++) // compute factorials specified # of times
    { for(j=0; j <= limit; j++)
      { rtimer.Start(); // time recursive version
```

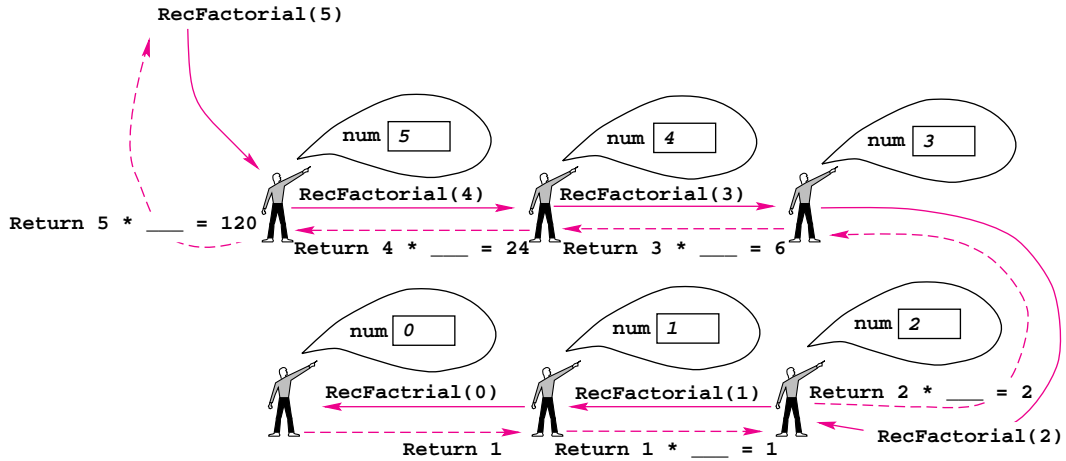


Figure 10.6 Recursive calls of RecFactorial(6).

```

    rval = RecFactorial(j);
    rtimer.Stop();
    itimer.Start(); // time iterative version
    ival = Factorial(j);
    itimer.Stop();
    if (rval != ival) // note any differences
    { cout << "calls differ for " << j << endl;
      cout << "recursive = " << rval << " iterative = " << ival << endl;
    }
  }
}
cout << iters << " recursive trials " << rtimer.CumulativeTime() << endl;
cout << iters << " iterative trials " << itimer.CumulativeTime() << endl;
return 0;
}
facttest.cpp

```



The recursive function `RecFactorial` is similar to the inductive definition of factorial given earlier. You will get better at understanding recursive functions as you gain more experience, but two ideas are helpful. (See *ctimer.h*, Program G.5 in How to G for information on the class `CTimer` used to time execution of program segments.)

To compute  $5!$ , six clones of the factorial function are needed as shown in Figure 10.6. The first call, from main, is shown in the upper left as `RecFactorial(5)`. The recursive calls are shown as solid arrows. The value passed to parameter `num` is shown in each clone. The return value is calculated by the expression `num * RecFactorial(num-1)`; this is shown by the dashed lines. For example, the last clone called generates no recursive calls and returns 1. This value is used to calculate  $1 \times 1$  so that 1 is returned from the clone with parameter `num == 1`. Each returned

470

**Chapter 10** Recursion, Lists, and Matrices

value is plugged into the expression `num * RecFactorial(num-1)` as the value of the recursive call, finally yielding  $5 \times 24 == 120$ , which is returned to `main`.

```


O U T P U T



Runs on a Pentium II 300Mhz running Windows NT



```
prompt> facttest
enter # of iterations between 1 and 1000000: 1000
upper limit on factorial between 10 and 100: 20
1000 recursive trials 6.2
1000 iterative trials 4.816
prompt> facttest
enter # of iterations between 1 and 1000000: 1000
upper limit on factorial between 10 and 100: 30
1000 recursive trials 24.807
1000 iterative trials 22.581
```



using int rather than BigInt



```
prompt> facttest
enter # of iterations between 1 and 1000000: 10000
upper limit on factorial between 10 and 100: 20
10000 recursive trials 0.791
10000 iterative trials 0.691
```



Runs on a Sparc Ultra 30 with 384 megabytes of memory



```
prompt> facttest
enter # of iterations between 1 and 1000000: 10000
upper limit on factorial between 10 and 100: 20
10000 recursive trials 4.28
10000 iterative trials 1.9
```


```

Two things will help you understand recursion, but practice in thinking recursively is the best way to gain understanding.

- Trace each recursive call by drawing clones or other diagrams that show each recursive function call, the function's variables and parameters, and the value returned.
- Believe the recursion works and verify that that returned value is used correctly.

**Program Tip 10.2: Believe the recursion works.** This means that you *assume* that the recursive call works correctly, and you examine the code to see that the result of the recursive call is *used* correctly. For example, in calculating  $4!$ , you assume that the call to calculate  $3!$  yields the correct result: 6. The statement that uses this result

```
return num * RecFactorial(num-1);
```

will then return  $4 \times 6$ , the value of `num` times the result of the recursive call. This is the correct answer for  $4!$ .

**Program Tip 10.3: Trace the recursive calls to see that the clones produce the correct results.** This can be a tedious task, but some people like the assurance of understanding precisely how the recursively called functions work together. (A trace is shown in Figure 10.6 for the computation of  $5!$ ). In many examples of recursion that you'll see, tracing all the calls will be difficult to impossible because there will be so many of them. It's often helpful to trace the last call *before* the base case is reached, and to verify that the base case return value works with the last call.

Based on the sample runs, which of the recursive and iterative functions is best? The answer is—as it is so often—“*it depends.*” It depends on (at least) how many times the factorial function will be called, it depends on what kind of computer is used, and it depends on what compiler is used. When run on a Pentium computer, the difference between the two versions is 0.1 seconds for 200,000 calls with `int` values as shown in the output. The difference is greater for `BigInt` values. The differences on a Sun UltraSparc computer are much more pronounced since that computer doesn't process recursion very well.

### 10.3.2 Fibonacci Numbers

Fibonacci numbers are integral in many areas of mathematics and computer science. These numbers occur in nature as well [PL90]. For example, the scales on pineapples are grouped in Fibonacci numbers. In [Emm93], Fibonacci numbers are cited as the conscious basis of works by the composers Bartok and Stockhausen. Knuth [Knu97] describes the mathematical constant  $\phi = \frac{1}{2}(1 + \sqrt{5})$  as “intimately connected with the Fibonacci numbers,” and the ratio of  $\phi$  to 1 is “said to be the most pleasing proportion aesthetically, and this opinion is confirmed from the standpoint of computer programming aesthetics as well.” The first 16 Fibonacci numbers are given below; this sequence originated in 1202 with Leonardo Fibonacci, whom Knuth calls “by far the greatest European mathematician before the Renaissance.”

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

In general, each number in this sequence is the sum of the two numbers before it; the first two Fibonacci numbers are the exception to this rule. In keeping with tradition in C++

numbering schemes, the first Fibonacci number is  $F(0)$ ; that is, we start numbering from zero rather than one. This leads to the inductive or recursive definition of the Fibonacci numbers:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases} \quad (10.3)$$

As is the case with the recursive definition of factorial, the recursive definition of the Fibonacci numbers can be translated almost verbatim into a C++ function. The function `RecFib` is shown in *fibtest.cpp*, Program 10.7. The function `Fib` computes Fibonacci numbers iteratively. The difference in this case between the recursive and iterative functions is much more pronounced than it was for the factorial function. Note that  $F(30) = 1,346,269$ .

---

#### Program 10.7 fibtest.cpp

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

// Illustrates "bad" recursion for computing Fibonacci numbers

const int FIB_LIMIT = 20;           // largest fib # calculated

long RecFib(int n)
// precondition: 0 <= n
// postcondition: returns the n-th Fibonacci number
{
    if (0 == n || 1 == n)
    {
        return 1;
    }
    else
    {
        return RecFib(n-1) + RecFib(n-2);
    }
}

long Fib(int n)
// precondition: 0 <= n
// postcondition: returns the n-th Fibonacci number
{
    long first=1, second=1, temp;
    int k;
    for(k=0; k < n; k++)
    {
        temp = first;
        first = second;
        second = temp + second;
    }
    return first;
}

int main()
```

```

{
    CTimer rtimer, itimer;
    int j;
    long k;
    long ival, rval;
    long iters = PromptRange("enter # of iterations", 1L, 100000L);
    int limit = PromptRange("n, for n-th Fibonacci ", 1, 30);

    for(k = 0; k < iters; k++)
    {   for(j=0; j <= limit; j++)
        {   rtimer.Start();
            rval = RecFib(j);
            rtimer.Stop();
            itimer.Start();
            ival = Fib(j);
            itimer.Stop();
            if (ival != rval)
            {   cout << "calls differ for " << j << endl;
                cout << "recursive = " << ival << " iterative = " << rval << endl;
            }
        }
    }
    cout << iters << " recursive trials " << rtimer.CumulativeTime() << endl;
    cout << iters << " iterative trials " << itimer.CumulativeTime() << endl;
    return 0;
}

```

fibtest.cpp

## O U T P U T

*Run on a Pentium II 300Mhz running Windows NT*

```

prompt> fibtest
enter # of iterations between 1 and 100000: 100
n, for n-th Fibonacci between 1 and 30: 30
100 recursive trials 49.932
100 iterative trials 0.02

```

*Run on a Pentium 100Mhz running Linux*

```

prompt> fibtest
enter # of iterations between 1 and 100000: 100
n, for n-th Fibonacci between 1 and 30: 30
100 recursive trials 205.5
100 iterative trials 0.0

```

The granularity of the timing doesn't accurately reflect the iterative function; 10,000 calls of the iterative function take about 1.1 seconds to compute  $F(30)$ . Extrapolating the result of 49.932 seconds for 100 trials of the recursive function shows that 100,000 iterations would take 49,932 seconds, or nearly 13 hours, for what is done in about 10

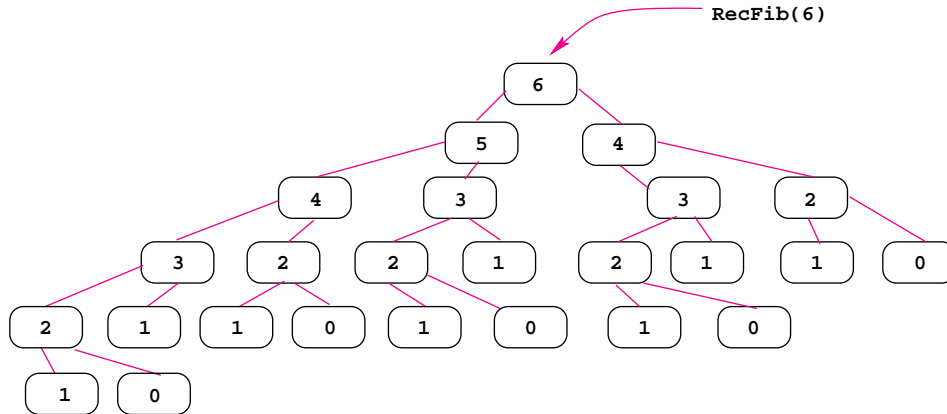


Figure 10.7 Recursive calls of `RecFib(6)`, the number in each box is the value of the parameter `num`.

seconds using the iterative function. What are the differences between calculating  $n!$  and  $F(n)$  that cause such a disparity in the timings of the recursive and iterative versions? For example, is the time due to the recursive depth (number of clones)? As we will see, the depth of recursive calls is not what causes problems here. Only 30 clones exist at one time to calculate  $F(30)$ . However, the total number of clones (or recursive calls) is 2,692,637. This huge number of calls is illustrated in Figure 10.7 for the calculation of  $F(6)$ , which requires a total of 25 recursive calls.

If you examine Figure 10.7 carefully, you'll see that the same recursive call is made many times. For example,  $F(1)$  is calculated eight times. Since the computer is not programmed to remember a number previously calculated, when the call  $F(6)$  generates calls  $F(5)$  and  $F(4)$ , the result of  $F(4)$  is not stored anywhere. When the calculation of  $F(5)$  generates  $F(4)$  and  $F(3)$ , the entire sequence of calls for  $F(4)$  is made again. The iterative function `Fib` in `fibtest.cpp` is fast because it makes roughly  $n$  additions to calculate  $F(n)$ ; the number of additions is **linear**. In contrast, the recursive function makes an **exponential** number of additions. In this case the speed of the machine is not so important, and the recursive function is *much* slower than the iterative function.

In later courses you may study methods that will permit you to determine when a recursive function should be used. For now, you should know that recursion is often very useful, as with the directory searching functions, and sometimes is very bad, as with the recursive Fibonacci function.

### 10.3.3 Permutation Generation

A **permutation** is a re-arrangement. In mathematics, a permutation of a list of  $n$  numbers like  $(1, 2, 3)$  is any one of the  $n!$  different orderings of the numbers. For example, all orderings of the numbers 1–3 follow.

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Permutations are used in a branch of computer science called **combinatorics**, and also in statistics, social sciences, and mathematics. As we'll see, generating permutations recursively uses a technique called **backtracking** that can be applied to solve many different problems.

We'll develop a recursive function for generating all the permutations of the elements in a vector. The function will print all permutations, but we'll discuss how to process the permutations in other ways. We'll follow the two guidelines from Program Tip 10.1 in developing our recursive function. First we'll identify a base case, a case that is easy to compute and that won't make any recursive calls. We also have to identify why it's the base case and use that to focus on the second guideline: what part of the problem will get smaller with each recursive call, thus eventually getting to the base case? Most recursive problems are parameterized by some notion of size. In each recursive call the size decreases, eventually reaching the base case. In *digits3.cpp*, Program 10.2, the size of the problem is the number of digits in the number being converted to English. In traversing directories the size is the the number of subdirectories in a directory; eventually a directory with no subdirectories must be found. In computing factorial, the number  $n$  for which  $n!$  is computed is the size of the problem.

The permutation problem is parameterized by the size of the vector being permuted. A vector with no elements, or with only one element, is very easy to permute in all ways. If this is the base case, we'll need to work on transforming the problem of permuting an  $n$ -element vector into a problem that permutes a smaller-sized vector. If you look at the list of the six different permutations of  $(1, 2, 3)$  you may see that the permutations can be divided into three groups of two permutations. In each group the first number stays the same and the other elements are permuted in all ways. This will work for a 4-element list too. The first element can take one of four values. For each of the four values, permute the remaining three in all possible ways. The first six of twenty-four permutations of  $(1, 2, 3, 4)$  are shown below. The four is fixed and the rest of the vector is permuted in all ways as a 3-element vector.

```
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

It's actually tricky to develop a recursive solution thinking about the problem this way because the simpler problem, one of permuting the rest of the vector, isn't the same kind of problem as what we start with. We start with a vector of  $n$ -elements, and the subproblem is to permute everything except the first element. But this subproblem

doesn't involve a vector, it involves a part of the vector. We'll adopt an approach that is often useful in recursive problems, we'll think of the problem in a different way that is more easily reducible to a simpler case. Note that in permuting  $(1, 2, 3, 4)$  when the first two elements are fixed, say  $(4, 1)$ , the rest of the elements are permuted in all possible ways. We'll use the idea of fixing the first  $k$  elements in a vector, those with indexes  $0 \dots k-1$  in a vector. We'll permute the other elements, with indexes  $k \dots n-1$ , in all possible ways. The base case that's easily solved is when all  $n$  elements are fixed, there are no more elements to permute. Initially no elements are fixed. This leads to the two functions whose headers follow.

```
void PermuteHelper(tvector<int>& list, int n);
// pre: first n elements of list are fixed and won't change
// post: elements n..list.size()-1 are permuted in
//       all possible ways, list is in original order

void Permute(tvector<int>& list)
// post: elements of list permuted in all possible ways
{
    PermuteHelper(list, 0);
}
```

Users will call `Permute`, the function `PermuteHelper` exists only to make the recursion simple to code. In a class, `PermuteHelper` would be a private helper function, not accessible to the user.

*Developing `PermuteHelper`.* We've already decided that the base case, in which all elements are fixed so that `n == list.size()`, results in printing the vector. What about the recursive calls? The vector element with index  $n$  is the left-most element that changes since elements with indexes  $0 \dots n-1$  are fixed. Element `list[n]` must take on all values from the remaining, unfixed elements, and then all permutations should be generated. For example, to permute  $(5, 3, 1, 4, 2)$ , with one element fixed (index zero), we'll let the index one element take on each of the unfixed values. This is shown below, where the `x` indicates where the 3, originally with index one, is swapped to bring each unfixed element into the index one slot. The 3 originally in the index one slot is swapped into slots with indexes two, three, and four to generate each recursive call. It's swapped back after the recursive call to restore the vector as it was, satisfying the postcondition.

```
5  ___ ___ ___ ___
5  _3_ ___ ___ ___
5  _1_ _x_ ___ ___
5  _4_ ___ _x_ ___
5  _2_ ___ ___ _x_
```

This leads to the function below.

```
void PermuteHelper(tvector<int>& list, int n);
```

## 10.3 Comparing Recursion and Iteration

477

```

// pre: first n elements of list are fixed and won't change
// post: elements n..list.size()-1 are permuted in
//       all possible ways, list is in original order
{
    int len = list.size();
    int k;
    if (n == len) // all elements fixed, print
    { Print(list);
    }
    else
    { for(k=n; k < len; k++)
      { Swap(list[n],list[k]);
        PermuteHelper(list,n+1);
        Swap(list[n],list[k]);
      }
    }
}

```

This prints all the permutations. If instead of printing, you wanted to pass the permuted vector to a function for processing, you'd have to change the call of `Print` in `PermuteHelper`. Alternatively, you could develop a method for iterating over the permutations, one at a time. The class `Permuter` does this (see *How to G* for details.) A `Permuter` object is constructed from a vector, and then iterates over the vector returning permutations in alphabetic or lexicographic order. If a `Permuter` is initialized with the vector  $(4, 3, 2, 1)$ , then the first two vectors returned by `Current` will be  $(4, 3, 2, 1)$  and  $(1, 2, 3, 4)$  since a `Permuter` wraps to the first vector alphabetically after the list one. A `Permuter` uses only `int` vectors, but as Program 10.8, shows, an `int` vector can be used to index any other vector effectively permuting any kind of vector.




---

 Program 10.8 usepermuter.cpp

```

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

#include "tvector.h"
#include "permuter.h"

int main()
{
    tvector<int> list;
    tvector<string> slist;
    string names[] = {"first", "second", "third"};
    int k;
    for(k=0; k < 3; k++)
    { list.push_back(k);
      slist.push_back(names[k]);
    }
}

```

478

## Chapter 10 Recursion, Lists, and Matrices

```

}
Permuter p(list);
for(p.Init(); p.HasMore(); p.Next())
{
    p.Current(list);
    for(k=0; k < list.size(); k++)
    {
        cout << list[k] << " ";
    }
    cout << endl;
}
for(p.Init(); p.HasMore(); p.Next())
{
    p.Current(list);
    for(k=0; k < list.size(); k++)
    {
        cout << slist[list[k]] << " ";
    }
    cout << endl;
}
return 0;
}

```

usepermuter.cpp

## O U T P U T

```

prompt> usepermuter
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
first second third
first third second
second first third
second third first
third first second
third second first

```

## 10.4 Scope and Lifetime

In this section we'll discuss methods that are used to alter the lifetime of a variable in a class or program and the scope of declaration. We'll use two simple examples that extend the computation of Fibonacci numbers from *fibtest.cpp*, Program 10.7, and then use these examples as a springboard to explore general principles of lifetime and scope. We touched on scope in Section 5.3; the scope of a declaration determines where in a function, class, or program the declaration can be used. *Lifetime* refers to the duration of storage associated with a variable. To be precise, scope is a property of a name or

identifier (e.g., of a variable, function, or class) that determines where in a program the identifier can be used. Lifetime is a property of the storage or memory associated with an object.

### 10.4.1 Global Variables

Suppose we want to calculate exactly how many times the function `RecFib` is called to compute `RecFib(30)` in `fibtest.cpp`, Program 10.7. We can increment a counter in the body of `RecFib`, but we need to print the value of the counter in `main` when the initial call of `RecFib` returns. The **global variable** `gFibCalls` in `recfib.cpp`, Program 10.9, keeps this count. The scope of a global variable is an entire program as opposed to a local variable that can be accessed only within the function in which the variable is defined. In C++ a global variable has **file scope** since it is accessible in all functions defined in the file in which the global variable appears.

---

#### Program 10.9 `recfib.cpp`

---

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

// Illustrates "bad" recursion for computing Fibonacci numbers
// and a global variable to count # function calls

int gFibCalls = 0;

long RecFib(int n)
// precondition: 0 <= n
// postcondition: returns the n-th Fibonacci number
{
    gFibCalls++;
    if (0 == n || 1 == n)
    {    return 1;
    }
    else
    {    return RecFib(n-1) + RecFib(n-2);
    }
}

int main()
{
    int num = PromptRange("compute Fibonacci #",1,40);
    cout << "Fibonacci # " << num << " = " << RecFib(num) << endl;
    cout << "total # function calls = " << gFibCalls << endl;
    return 0;
}
```

---

`recfib.cpp`

---

## O U T P U T

```

prompt> recfib
compute Fibonacci # between 1 and 40: 10
Fibonacci # 10 = 89
total # function calls = 177
prompt> recfib
compute Fibonacci # between 1 and 40: 20
Fibonacci # 20 = 10946
total # function calls = 21891
prompt> recfib
compute Fibonacci # between 1 and 40: 30
Fibonacci # 30 = 1346269
total # function calls = 2692537

```

I use the prefix `g` to differentiate global variables from other variables. Global variables are declared outside of any function, usually at the beginning of a file. Unlike local variables, global variables are automatically initialized to zero unless a different initialization is specified when the variable is defined. There are rare occasions when global variables must be used, as with `gFibCalls` in *recfib.cpp*. However, using many global variables in a large program quickly leads to maintenance headaches because it is difficult to keep track of what identifiers have been used. In particular, it's possible for a global declaration to be hidden or **shadowed** by a local declaration. For example, suppose you want to implement the member function `Point::toString`. The class `Point` has two private instance variables `x` and `y`, both are doubles.

The functions `toString` in *strutils.h*, Program G.8 (see How to G) convert ints and doubles to strings, so you might write:

```

string Point::toString() const
{
    return "(" + toString(x) + ", " + toString(y) + ")";
}

```

Unfortunately, this will not compile. The compiler treats the calls of `toString`, that are intended as calls of the free, or global functions in *strutils.h*, as recursive calls with arguments that do not match the formal parameter list. The member function `Point::toString` shadows the global, free functions.

We can fix this problem using the scope resolution operator `::`. Applied to an identifier, `::` references a global object (or function) so we can write the function as follows:

```

string Point::toString() const
{
    return "(" + ::toString(x) + ", " + ::toString(y) + ")";
}

```

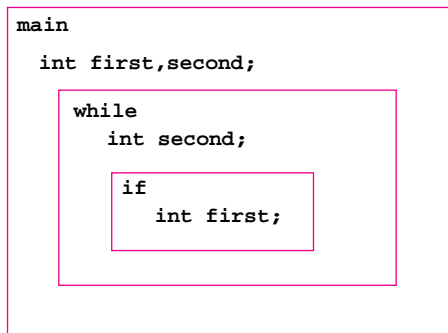


## 10.4.2 Hidden Identifiers

Even nonglobal identifiers can be shadowed, as illustrated in *scope.cpp*, Program 10.10. Because the braces, {}, that delimit function bodies and compound statements cannot overlap, there is always a scope “closest” to an identifier’s declaration. It is possible for an identifier to be reused within a **nested scope**. A scope is nested in another when the braces that define the scope occur within another set of braces. When a variable is used, it may seem unclear in which scope the variable is declared, but the “nearest” definition is the one used.

The variable `first` defined within the scope of the `if` statement is accessible only within the `if` statement. Assignments to `first` within the statement do not affect the variable `first` defined at the beginning of `main`, as shown in the output where `first` is printed as 4, 8, 16, and 32 except for the indented values, which show `first` within the `if` statement. It is also apparent from the output that the value of `second` defined in `main` is not affected by assignments to `second` in the `while` loop since these assignments are made to a variable defined within the loop. Schematically the scopes are illustrated in Figure 10.8, in which a variable is known within the innermost box in which it appears. The variable `second` defined within the `while` loop shadows the variable defined at the top of the function `main`. In general, shadowing leads to unexpected, although well-defined, behavior.

**Program Tip 10.4: Avoid using identifiers with the same name in nested scopes.** Hidden and shadowed identifiers lead to programs that are difficult to understand and ultimately lead to errors.



**Figure 10.8** Boxing to illustrate scope in Program 10.10

## Program 10.10 scope.cpp

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

// illustrates scope

int main()
{
    int first = 2;
    int second = 0;

    while (first < 20)
    {
        int second = first * 2;           // shadows previous second
        cout << "\tsecond = " << second << endl;
        first *= 2;
        if (first > 10)
        {
            int first = second;         // shadows previous first
            first = first/10;
            cout << "\tfirst = " << first << endl;
        }
        cout << "first = " << first << endl;
    }
    cout << "second = " << second << endl;
    return 0;
}
```

scope.cpp

## O U T P U T

```
prompt> scope
        second = 4
first = 4
        second = 8
first = 8
        second = 16
        first = 1
first = 16
        second = 32
        first = 3
first = 32
second = 0
```

### 10.4.3 Static Definitions

Global variables maintain their value throughout the execution of a program; they exist for the duration of the program. In contrast, a local variable defined in a function is constructed anew each time the function is called. We can change the lifetime of a local variable so that the variable maintains its value throughout a program's execution by using the word **static** as a modifier when the variable is defined. This is illustrated in *recfib2.cpp*, Program 10.11. A static `tvector` is defined to keep track of recursive calls so that the same recursive call is never made more than once. For example, the first call of `RecFib(4)` results in recursive calls of `RecFib(3)` and `RecFib(2)`. When these values are calculated, the values are stored in the `tvector` storage so that the values can be retrieved, for example, when `RecFib(2)` is called again. The key idea is that a recursive call is made once. All subsequent recursive calls with the same argument are evaluated by retrieving the stored value from `storage` rather than by making a recursive call. Notice how many fewer calls are made compared to the calculations of *recfib.cpp*, Program 10.9.

---

#### Program 10.11 *recfib2.cpp*

---

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

// Illustrates "bad" recursion for computing Fibonacci numbers
// but made better using a static vector for storing values

int gFibCalls = 0;
const int FIB_LIMIT = 40;

long RecFib(int n)
// precondition: 0 <= n
// postcondition: returns the n-th Fibonacci number
{
    static tvector<int> storage(FIB_LIMIT+1,0);

    gFibCalls++;
    if (0 == n || 1 == n)
    { return 1;
    }
    else if (storage[n] == 0)
    { storage[n] = RecFib(n-1) + RecFib(n-2);
      return storage[n];
    }
    else
    { return storage[n];
    }
}

int main()
{
```

484

## Chapter 10 Recursion, Lists, and Matrices

```

int num = PromptRange("compute Fibonacci #",1,FIB_LIMIT);
cout << "Fibonacci # " << num << " = " << RecFib(num) << endl;
cout << "total # function calls = " << gFibCalls << endl;
return 0;
}

```

recfib2.cpp

Like global variables, static local variables are automatically initialized to zero. However, it is a good idea to make initializations explicit. Static variables are constructed and initialized when a program first executes, *not* when a function is first called. The variable storage must be static in *recfib2.cpp*, or the values stored will not be maintained over all recursive calls. For recursive functions like `RecFib`, only one static variable is defined for all the recursive clones. The variable `storage` is local to `RecFib` but maintains its values for the duration of the program *recfib2.cpp*.

## O U T P U T

```

prompt> recfib2
compute Fibonacci # between 1 and 40: 10
Fibonacci # 10 = 89
total # function calls = 19
prompt> recfib2
compute Fibonacci # between 1 and 40: 20
Fibonacci # 20 = 10946
total # function calls = 39
prompt> recfib2
compute Fibonacci # between 1 and 40: 30
Fibonacci # 30 = 1346269
total # function calls = 59

```

## 10.4.4 Static or Class Variables and Functions

Just as it's possible for a static variable to have a lifetime for the duration of a program, maintaining its value over many function calls, a **static class variable** maintains its value over many object definitions. A static class variable actually exists outside of any object, it's part of a class rather than an object. In *staticdemo.cpp*, the static variable `ourCount` is incremented each time a `Pair` object is constructed. It's value is the number of `Pair` objects constructed in an entire program execution.

---

Program 10.12 staticdemo.cpp

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

struct Pair
{
    int x, y;
    Pair(int a, int b)
        : x(a), y(b)
    { ourCount++; }

    static int ourCount;
};

int Pair::ourCount = 0;

int main()
{
    Pair p(0,0);
    int k, limit = PromptRange("number of pairs? ", 1, 20000);
    for(k=0; k < limit; k++)
    { Pair p(k, 2*k); }
    cout << "# pairs created = " << Pair::ourCount << endl;
    return 0;
}
```

---

staticdemo.cpp

## OUTPUT

```
prompt> static
number of pairs? between 1 and 20000: 1000
# pairs created = 1001

prompt> static
number of pairs? between 1 and 20000: 5000
# pairs created = 5001
```

As shown, static class variables must be initialized outside the class declaration. Static variables are defined before `main` begins to execute. A static variable or function can be accessed using dot notation as though it were an instance variable or member function. In `staticdemo.cpp` the last output line could print `p.ourCount`. However, since static variables belong to a class rather than an object, it's possible to access them

using the class name and the scope resolution operator as shown. The prefix `our` signifies that the variable belongs to all objects, not to any particular object.

## Pause to Reflect



**10.9** The code segment shown below illustrates shadowing. Describe an input sequence that causes the words `Banana yellow Banana red Apple` to be printed (one per line).

```
string last = "Apple";
string word;
while (cin >> word && word != last)
{   string last = "Banana";
    cout << last << " " << word << endl;
}
cout << last << endl;
```

Describe an input sequence that causes the single word `Apple` to be printed. If the definition of `last` within the `while` loop is removed, what input sequence generates `Banana yellow Banana red Apple`?

**10.10** In the code fragment in the previous problem, if the definition of `last` before the `while` loop is removed, will the segment compile? Why?

**10.11** Describe how to use a static vector in a function to compute factorial to avoid computing  $n!$  if it has been computed before.

**10.12** In *staticdemo.cpp*, if `p.ourCount` is used instead of `Pair::ourCount` what variable `p` is accessed? If two `Pair` variables `p` and `q` are defined before the loop, and the only statement in the loop is

```
p = q;
```

What will the output of the program be?

## 10.5 Case Study: Lists and the Class `CList`

The programming language Lisp, and related languages like Scheme, have a long history of providing elegant and useful solutions to a wide variety of problems. Lisp was one of the first languages; its development began in 1958 and it was running on computers by 1960. Today Lisp is still used extensively, has an object-oriented extension, and is used in programming the text editor Emacs which I used to write this book. The basic structure in Lisp is a **list**. In this section we'll use a Lisp-like list class<sup>4</sup> called **CList** to explore recursion and an elegant solution to representing polynomials.

<sup>4</sup>It's Lisp-like in that programmers don't worry about memory management and cannot change a list once the list is created. It's not Lisp-like in that in this chapter list elements must be the same type.

### 10.5.1 What Is a `CList` Object?

The class `CList` is similar to the class `tvector` in that it's a homogeneous aggregate: each element of a list has the same type. It differs in two ways:

- `CList` collections do not support random access; accessing the first element takes less time than accessing the second, and accessing the  $n^{\text{th}}$  element takes  $n$ -times longer than accessing the first element.
- A `CList` collection is **immutable**. Once a list is created, it cannot be changed. You can't change an element of a list and you can't add an element to an existing list. Instead, you can create new lists. The `C` in `CList` stands for constant since lists cannot change once created.

There are two ways to create a `CList` object. Defining a `CList` object creates an empty list, one with no elements. The function `cons` is used to create a new list from a first element and an existing list. The program `listdemo.cpp`, Program 10.13 shows how `cons` is used to create lists from old lists.<sup>5</sup>

---

#### Program 10.13 `listdemo.cpp`

---

```
#include <iostream>
#include <string>
using namespace std;
#include "clist.h"

void Display(const CList<string>& list)
// post: list displayed on one line, comma separated
{
    cout << "size = " << list.Size() << ": " << list.Printer(",") << endl;
}
int main()
{
    CList<string> s1, s2, s3, s4, s5; // create empty lists
    s2 = cons(string("tomato"),s1);
    s3 = cons(string("carrot"),s2);
    s4 = cons(string("celery"),s3);
    s5 = cons(string("peapod"),s3);

    Display(s1); Display(s1.Tail()); cout << "-" << endl;
    Display(s2); Display(s2.Tail()); cout << "-" << endl;
    Display(s3); Display(s3.Tail()); cout << "-" << endl;
    Display(s4); Display(s4.Tail()); cout << "-" << endl;
    Display(s5); Display(s5.Tail()); cout << "-" << endl;
    return 0;
}
```

---

`listdemo.cpp`

---

<sup>5</sup>The explicit use of `string` as a constructor for the literal `"carrot"`, for example, is required in some compilers because of how templates are instantiated.

A `CList` is divided into two parts: the **Head** which is a string in a list of strings, an int in a list of ints, and so on; and the **Tail**, which is another `CList`, but without the first element (the **Head**). The function `cons` makes a new list by creating a new head and using an existing tail.

```


O U T P U T



```

prompt> listdemo
size = 0:
size = 0:
---
size = 1: tomato
size = 0:
---
size = 2: carrot,tomato
size = 1: tomato
---
size = 3: celery,carrot,tomato
size = 2: carrot,tomato
---
size = 3: peapod,carrot,tomato
size = 2: carrot,tomato
---

```


```

Figure 10.9 is a diagram of the five lists from `listdemo.cpp`. The lists `s4` and `s5` share the same tail. All the lists except for the empty `s1` share the list value "tomato", which is at the head of `s2`, is the tail of `s3`, and is part of `s4` and `s5` as well.

The method `CList::Printer` acts like an I/O manipulator. The separator/delimiter argument to `Printer` separates each item in the list being inserted onto the stream, so commas appear between each list element as shown. If no parameter is used, that is, `list.Printer()`, then each list item appears on a separate line, the separator is the newline character `'\n'`. It's possible to insert a list directly on a stream. For the list `s4` in `listdemo.cpp`, the call below generates the output shown with parentheses at the beginning and end of the output and commas separating each list element.

```
cout << "list = " << s4 << " size = " << s4.Size() << endl;
```

```


O U T P U T

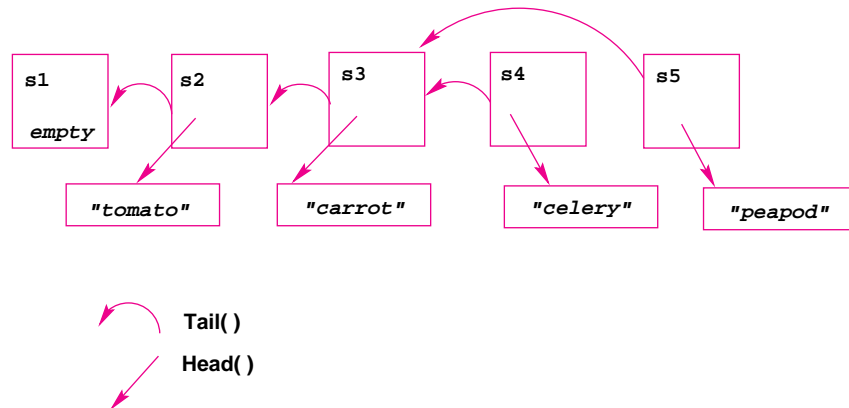


```

list = (celery, carrot, tomato) size = 3

```


```



**Figure 10.9** Diagram of five `CList` objects showing structure sharing.

### 10.5.2 Tail-ing Down a list

We can easily write a free function version of `CList::Size` as shown in Program 10.14. The first, recursive version is emblematic of many `CList` functions. The empty list is the base case, and the simpler recursive call results from using a list's tail which has one fewer element than the list. The iterative version uses the associated `CListIterator` class; it's slightly more cumbersome than the recursive version.

#### Program 10.14 `listsize.cpp`

```
int Size(const CList<string> & list)
// post: returns # elements in list
{
    if (list.IsEmpty()) return 0;
    return 1 + Size(list.Tail());
}

int Size(const CList<string> & list)
// post: returns # elements in list
{
    CListIterator<string> iter(list);
    int count = 0;
    for(iter.Init(); iter.HasMore(); iter.Next())
    { count++;
    }
    return count;
}
```

`listsize.cpp`

## Maurice Wilkes (b. 1913)

Maurice Wilkes is one of the elder statesmen of computer science. He was a peer of Alan Turing and worked in England on the EDSAC computer. Wilkes was awarded the second Turing award in 1967.



In work written in 1955 and published in 1956 [Wil56], Wilkes offers advice for team programming projects. It is interesting that the advice still seems to hold 40 years later. “It is very desirable that all the programmers in the group should make use of the same, or substantially the same, methods. Not only does this facilitate communication and cooperation between the members of the group, but it also enables their individual

experience more readily to be absorbed into the accumulated experience of the group as a whole ...the group should be organized to produce, on a common plan, the input routines, basic library subroutines, and error-diagnosis subroutines ...it will be much easier, once they are prepared, for an individual programmer to make use of them rather than to set about designing a system of his own.” Wilkes wonders where computer science fits—whether it is more closely tied to mathematics or to engineering [Wil95]:

*Many students who are attracted to a practical career find mathematics uncongenial and difficult; certainly it is not the most popular part of an engineering course for the majority of students. Admittedly, mathematics trains people to reason, but reasoning in real life is not of a mathematical kind. Physics is a far better training in this respect. The truth may be that computer science does not by itself constitute a sufficiently broad education, and that it is better studied in combination with one of the physical sciences or with one of the older branches of engineering.*

Wilkes pioneered many of the ideas in current computer architectures including microprogramming and cache memories. In 1951 he published the first book on computer programming. About object-oriented programming he says:

*[Object-oriented programming is] in my view, the most important development in programming languages that has taken place for a long time. Object-oriented programming languages may still be described as being in a state of evolution. No completely satisfactory language in this category is yet available.*

For more information see [Wil87, Wil95, Wil56].

### 10.5.3 Cons-ing Up a List

To see the benefits of recursion, we'll look at *readlist.cpp*, Program 10.15. The program reads words from a file and uses a standard list technique of **cons-ing up** a list by assigning the `cons` return-value to the list that's the argument to `cons`. Since the last word read is the last word cons-ed to the front, the list is in reverse order.

We'd like to have a version of `Read` that returns a list of words in the same order in which they're read. To do this efficiently (using `cons`) we'll need to think recursively. We'll recursively read from a stream using the following ideas.

1. If there are no words in the stream we'll return an empty list; this is the base case of the recursion.
2. Otherwise, we'll make a recursive call. We must decide what the arguments in the call are and how to process the returned result. To get closer to the base case of no words in the stream, we'll read a word. The resulting stream will be "shorter," and closer to the base case because it contains fewer words. What do we do with the result returned from the recursion?

---

#### Program 10.15 *readlist.cpp*

---

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

#include "clist.h"
#include "prompt.h"

CList<string> Read(istream& input)
// post: returns list, order of words reversed from input
{
    CList<string> result;
    string word;
    while (input >> word)
    {   result = cons(word,result);
    }
    return result;
}

int main()
{
    string filename= PromptString("filename ");
    ifstream input(filename.c_str());
    StringList list = Read(input);
    cout << "# words = " << list.Size() << endl;
    cout << "words: first = " << list.Head()
         << ", last = " << list.Last() << endl;
    return 0;
}
```

---

*readlist.cpp*

---

## O U T P U T

```

prompt> readlist
filename melville.txt
# words = 14353
words: first = death, last = Bartleby,

prompt> readlist
filename poe.txt
# words = 2324
words: first = requiescat!, last = The

```

In developing the recursive function, think about what the postcondition must be. We want the words to be in the same order in which they're read. Combined with the base case this is what we have so far.

```

CList<string> Read(istream& input)
// post: returns list, order of words same as in input
{
    string word;
    if (input >> word)
    {
        // ??? must develop this code
    }
    else
    { return CList<string>(); // an empty list
    }
}

```

Note that a temporary, or anonymous variable (see Section 7.4.1) is returned by constructing a `CList<string>` object. The following code is equivalent, but uses a named variable.

```

...
else
{ CList<string> temp;
  return temp;
}

```

In the recursive case, the recursive call will satisfy the postcondition. Remember that you must believe the recursion will work (see Program Tip 10.2.) What can you do with the returned result? What does it represent? The returned result represents a list of words except for the word just read, and *the order in list is the same as the order in input* according to the postcondition. This means you simply cons the word read to

what's returned by the recursive call. If we use this function, the output of `readlist.cpp` changes.

```
CList<string> Read(istream& input)
// post: returns list, order of words same as in input
{
    string word;
    if (input >> word)
    {   return cons(word,Read(input));
    }
    return CList<string>();
}
```

## OUTPUT

```
prompt> readlist
filename melville.txt
# words = 14353
first word = Bartleby, last word = death.

prompt> readlist
filename poe.txt
# words = 2324
first word = The last word = requiescat!
```

### 10.5.4 Append, Reverse, and Auxiliary Functions

In this section we'll look at one more list function, `append`, which adds a new element to the end of a list. Since we cannot change a list, a new element isn't really added. Instead, appending an element to a list creates a new list that's a copy of the old list, but with a new element added to the end. This is fundamentally different than `cons`. When `cons` creates a new list by adding an element to the front of a list, all the list storage, except that used by the new element at the front, is shared between the lists as diagrammed in Figure 10.9. When `append` is used, no storage can be shared. Program 10.16 illustrates the differences. The class, or static, function `ConsCalls` reports how much memory has been allocated by the `CList` class.

#### Program 10.16 `listappend.cpp`

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

494

## Chapter 10 Recursion, Lists, and Matrices

```

int main()
{
    CList<int> list,list2;
    int k;
    for(k=7; k >=0; k--)
    {    list = cons(k,list);
    }
    cout << list.Printer(",") << endl;
    cout << "memory = " << CList<int>::ConsCalls() << endl;

    for(k=0; k < 8; k++)
    {    list2 = append(k,list2);
    }
    cout << list2.Printer(",") << endl;
    cout << "memory = " << CList<int>::ConsCalls() << endl;

    return 0;
}

```

listappend.cpp

## O U T P U T

```

prompt> listappend
0,1,2,3,4,5,6,7
memory = 8
0,1,2,3,4,5,6,7
memory = 44

```

To create a list of eight elements using `cons` requires only eight list elements. However, using `append` requires 36 elements ( $44 - 8 = 36$ , note that  $1 + \dots + 8 = 36$ .) Essentially, creating an eight-element list using `append` requires creating a one-element list, a two-element list, a three-element list, and so on until the eight element list is created. So although `append` is a useful function, it's an expensive function to use.

**Program Tip 10.5: Don't worry about efficiency until you know that you need to.** If you can easily solve a list problem using `append`, then it's the right tool to use until you determine that its inefficiencies make a difference.

Suppose, for example, that you need to reverse a list. A natural recursive solution using `append` can be derived as follows.

1. The base case, as it is with many list functions, is an empty list. The reverse of an empty list is an empty list, so no recursion is needed.

Table 10.1 Reversing a list using an auxiliary reversed-so-far list.

list being reversed	list reversed-so-far
(1,2,3,4)	()
(2,3,4)	(1)
(3,4)	(2,1)
(4)	(3,2,1)
()	(4,3,2,1)

- Most recursive list functions recurse on a list's tail. If we've successfully reversed the tail (remember, believe in the recursion) how can we reverse the entire list? Appending the head of the list to the reversed tail yields the reverse of the entire list.

This reasoning leads to the following function.

```
CList<string> Reverse(const CList<string>& list)
{
    if (list.IsEmpty()) return list;
    return append(list.Head(), Reverse(list.Tail()));
}
```

This solution is terse and elegant, but it's expensive in time and memory. Reversing an  $n$ -element list requires calling `append`  $n$  times and a total of  $1 + 2 + \dots + n = n(n + 1)/2$  allocated list elements. We'd like to develop a reversing algorithm using `cons`, but if you think about the recursion for a while, you'll see that it's not straightforward to develop.

We'll use a common technique of accumulating the reversed result in another list variable. We'll use two parameters in the reversing function:

- The list being reversed. The function recurses on this list, using the standard technique of using the list's tail as the recursive argument.
- The list that's the reversed list so far. Initially this list is empty since nothing has been reversed. When there's one element left in the list being reversed, all the other elements from the original list will be in this reversed-so-far list, and will be in reverse order.

Table 10.1 shows what we want the relationship between these two lists to be if we start with a list  $(1, 2, 3, 4)$ .

The insight of using the auxiliary reversed-so-far list enables us to use `cons` to build the reversed list. We can add the head/first element from the list being reversed to the front of the reversed-so-far list making progress towards the base case. We'll call the auxiliary, two-parameter reversing function from a single parameter function so that client code can create a reversed list without knowing about the second parameter. Two reversing functions, `Reverse` and `Reverse2`, are shown in Program 10.17. `Reverse2` uses the auxiliary function. The output shows the number of list elements

allocated when both functions are called. In this program we use an alias `StringList` for `CList<string>`. We'll discuss the syntax for the alias after the program.

---

Program 10.17 listreverse.cpp

---

```

#include <iostream>
#include <string>
using namespace std;
#include "clist.h"

StringList RevAux(StringList list, StringList sofar)
// pre: list = (a_0, a_1, ..., a_(n-1))
// post: returns (a_(n-1), ..., a_1, a_0, sofar)
{
    if (list.IsEmpty()) return sofar;
    return RevAux(list.Tail(), cons(list.Head(),sofar));
}

StringList Reverse2(StringList list)
// pre: list = (a_0, a-1, ..., a_(n-1))
// post: return (a_(n-1), ... a_1, a_0)
{
    return RevAux(list,StringList());
}

StringList Reverse(StringList list)
// pre: list = (a_0, a-1, ..., a_(n-1))
// post: return (a_(n-1), ... a_1, a_0)
{
    if (list.IsEmpty()) return list;
    return append(list.Head(),Reverse(list.Tail()));
}

void Print(StringList list)
{
    cout << list.Printer(",") << endl;
    cout << "# cons calls = " << StringList::ConsCalls() << endl << endl;
}

int main()
{
    StringList spices,spices2;

    spices = cons(string("paprika"), cons(string("cayenne"),
        cons(string("chili"), cons(string("turmeric"),
            cons(string("pepper"), StringList()))));
    spices2 = cons(string("curry"), cons(string("coriander"),
        cons(string("cumin"), spices)));

    Print(spices);
    Print(spices2);
    Print(Reverse(spices));
    Print(Reverse2(spices));
    return 0;
}

```

---

listreverse.cpp

---

## O U T P U T

```

prompt> listreverse
paprika,cayenne,chili,turmeric,pepper
# cons calls = 5

curry,coriander,cumin,paprika,cayenne,chili,turmeric,pepper
# cons calls = 8

pepper,turmeric,chili,cayenne,paprika
# cons calls = 23

pepper,turmeric,chili,cayenne,paprika
# cons calls = 28

```

Using the name `CList<string>` each time we want to define a variable or declare a parameter leads to lots of typing and code that's hard to read. Using an alias, implemented in C++ using a **typedef**, makes code simpler to read and can make some modifications easier. The header file `clist.h`, Program G.12 in *How to G* introduces the typedefs `StringList` and `StringListIterator`. A typedef is a



convenience, but the alias introduced often helps in reading and understanding code. For example, you might use the new name `Integer`, as shown in the syntax diagram. You can use the name `Integer` like a

**Syntax: typedef**

```

typedef CList<string> StringList;
typedef BigInt Integer;

```

type, but change it later to `int` and recompile your program to update all uses of `Integer`. Complicated declarations are often easier to understand with a typedef. For example, the definition `CList<CList<string>> list` will not compile, because the compiler misinterprets the `>>` as the insertion operator; you must include a space, as in `CList<CList<string> > list`. Using a typedef can make this simpler: `CList<StringList> list`.

**Pause to Reflect**

**10.13** In `listdemo.cpp`, Program 10.13, how will the output change if the call below (where `X` is 1,2,3,4)

```
Display(sX.Tail());
```

is changed in all five places it occurs to the following

```
Display(sX.Tail().Tail());
```

**10.14** In the initialization of `spices` in Program 10.17, `listreverse.cpp`, the final argument in the constructor is `StringList()`. Why is this argument used? Can it be replaced by `spices`? Can it be replaced by `StringList::EMPTY`?

**10.15** If the following statement is added as the last statement in `main` in `readlist.cpp`, Program 10.15, what values are printed for each of the runs shown in the output box?

```
cout << "# cons calls = "
      << CList<string>::ConsCalls() << endl;
```

Suppose the call to `cons` in the `while` loop of the function `Read` is replaced by a call to `append`. What values are printed by the `ConsCalls()` statement?

**10.16** Write a nonrecursive function that reverses a list using a `CListIterator` and `cons`. Use the same idea that's used in the recursive function, define a variable `sofar` and maintain the invariant: *sofar is the reverse of all the elements already processed*. The loop test should be:

```
while (! list.IsEmpty())
```

**10.17** Write a function `append` that appends one list to another. Conceptually, the call below yields the list `(1, 2, 3, 4, 5, 6)`.

```
append( (1,2,3), (4,5,6) )
```

The function should `cons` as many elements as there are in parameter `lhs`.

```
CList<int> append(const CList<int>& lhs,
                const CList<int>& rhs)
// pre: lhs = (a1,a2,...,an), rhs = (b1,b2,...,bm)
// post: returns list (a1,a2,...,an,b1,b2,...,bm)
```

**10.18** Write a function `Flatten` that creates one list from a list of lists. For example, the first list below is flattened into the second.

```
( ("apple", "cherry"),
  ("big", "little", "tiny"),
  ("november") )
```

```
("apple", "cherry", "big", "little", "tiny", "november")
```

```
StringList Flatten(CList<StringList> list)
// post: return a flattened form of list
```

**10.19** Consider the function `Create` that follows. What's printed by the statement calling `Create`?

```
cout << Create(5).Printer(",") << endl;
```

You'll need to think carefully about what's going on here and review what happens when a list is inserted onto an output stream (see *listdemo.cpp*, Program 10.13).

```
typedef CList<int> IntList;
CList<IntList> Create(int n)
{
    CList<IntList> result;
    int j,k;
    for(j=0; j < n; j++)
    {   IntList nlist;
        for(k=j; k >= 0; k--)
        {   nlist = cons(k,nlist);
        }
        result = cons(nlist,result);
    }
    return result;
}
```

### 10.5.5 Polynomials Implemented with Lists

The class `CList` is simple to use and motivates recursion since many list functions are more easily implemented recursively than iteratively. But what good is the class other than as something to study? In general, when should we think about using a `CList` object rather than a `tvector` object? A `tvector` can grow to accommodate more elements, supports random access, and allows its elements to change. A `CList` cannot be changed, is grown by creating new lists, and access is sequential. Lists provide efficient representations of **sparse structures**. For example, consider the polynomial  $2x^7 + 4x^3 + 6x^2 + 8$ . For the moment we'll consider just the exponents and ignore the coefficients. Conceptually the polynomial's exponents are (7, 3, 2, 0). Should these be stored as a list or a vector? As with many questions, the answer is "it depends."

It depends on what operations we'll perform on polynomials. Suppose that we want to add  $5x^4 + 3x^3 + x$  to the polynomial. Again using exponents we have (7, 3, 2, 0) + (4, 3, 1). The result (again without coefficients) is (7, 4, 3, 2, 1, 0) since the resulting polynomial is  $2x^7 + 5x^4 + 7x^3 + 6x^2 + x + 8$ . To add a term like  $5x^4$  to a polynomial requires shifting the terms of the polynomial to make room for the new term if we keep the terms in order, sorted by exponent. Keeping terms in order is a good idea because it will make arithmetic on polynomials much simpler. However, shifting vector elements is expensive. If we use vectors, we might choose to represent  $2x^7 + 4x^3 + 6x^2 + 8$  with coefficients as (2, 0, 0, 0, 4, 6, 0, 8) giving the coefficient for every term, where the position in the vector determines the exponent. This structure makes addition very simple. For example,  $(2x^7 + 4x^3 + 6x^2 + 8) + (5x^4 + 3x^3 + x)$  is realized with vectors as follows:

500

**Chapter 10** Recursion, Lists, and Matrices

$$\begin{array}{r}
 (2, 0, 0, 0, 4, 6, 0, 8) \\
 + \quad \quad \quad (5, 3, 0, 1, 0) \\
 \hline
 (2, 0, 0, 5, 7, 6, 1, 8)
 \end{array}$$

which is the result  $2x^7 + 5x^4 + 7x^3 + 6x^2 + x + 8$ . This representation is very inefficient in its use of storage for the polynomial  $7x^{100} + 2x + 1$ . In general, polynomials are sparse because not every exponent between 0 and the degree of the polynomial is typically represented by a nonzero coefficient.<sup>6</sup>

**10.5.6** `CList` and Sparse, Sequential Structures

Using `CList` objects to represent polynomials lets us represent sparse polynomials simply. Since most polynomials are accessed sequentially, processing each term of the polynomial in order, vectors don't supply an advantage since their principal strength is random access. As we'll see, `CList` representations of polynomials are efficient and easy to use in programs.

**Program 10.18** `polydemo.cpp`

```

#include <iostream>
using namespace std;

#include "poly.h"

// simple demo of polynomials

int main()
{
    Poly p1, p2, p3;

    p1 = Poly(5,7) + Poly(4,2) + Poly(3,1) + Poly(2,0);
    p2 = Poly(3,5) + Poly(2,4) + Poly(3,2);

    cout << "p1 = " << p1 << endl;
    cout << "p2 = " << p2 << endl;
    cout << "sum = " << p1+p2 << endl;
    cout << "p3 = " << p3 << endl;
    return 0;
}

```

`polydemo.cpp`

<sup>6</sup>The degree of a polynomial is the largest exponent.

## O U T P U T

```

prompt> polydemo
p1 = 5x^7 + 4x^2 + 3x + 2
p2 = 3x^5 + 2x^4 + 3x^2
sum = 5x^7 + 3x^5 + 2x^4 + 7x^2 + 3x + 2
p3 = 0

```

Polynomials are created in four ways:

1. The default constructor creates the constant zero.
2. A single term polynomial  $ax^n$  is created by `Poly p(a,n)`;
3. Polynomials can be added together to create new polynomials.
4. A polynomial can be multiplied by a constant to create a new polynomial.



The polynomial class is largely a **wrapper** class around a `CList<Pair>` object where a `Pair` is simply a struct with a coefficient and an exponent. For details on the implementation see `poly.h`, Program G.13 in *How to G*. A `Poly` object has a leading term, obtained via `Poly::Head` and all the other terms, obtained via `Poly::Tail`. Accessor functions supply the degree and leading coefficient of a polynomial as shown in the function `MonoMult` of `polymult.cpp`, Program 10.19. The program also shows how to evaluate a polynomial at a point, how to multiply by a constant, the static function `Poly::TermsAllocated` which reports memory usage for polynomials, and the static constant `Poly::ZERO` which represents the constant 0.

---

#### Program 10.19 `polymult.cpp`

```

#include <iostream>
using namespace std;

#include "poly.h"

Poly MonoMult(const Poly& mono, const Poly& rhs)
// pre: mono is a single term (monomial)
// post: return mono*rhs
//      if mono is a polynomial, returns mono.Head()*rhs
{
    if (rhs.IsPoly())
    {
        return Poly(mono.leadingCoeff()*rhs.leadingCoeff(),
                    mono.degree() + rhs.degree()) + MonoMult(mono,rhs.Tail());
    }
    return Poly::ZERO; // base case accumulated properly
}

int main()

```

502

## Chapter 10 Recursion, Lists, and Matrices

```

{
  Poly p1,p2,p3,p4;
  double x;
  cout << "value of x ";
  cin >> x;

  p1 = Poly(3,2) + Poly(4,1) + Poly(-3,0);
  p2 = Poly(4,2) + Poly(3,1) + Poly(2,0);
  p3 = Poly(5,3);

  cout << "p1 at " << x << ", " << p1.at(x) << "\t : " << p1 << endl;
  cout << "p2 at " << x << ", " << p2.at(x) << "\t : " << p2 << endl;
  cout << "p3 at " << x << ", " << p3.at(x) << "\t : " << p3 << endl;
  p4 = MonoMult(p3,p1);
  cout << "p4 at " << x << ", " << p4.at(x) << "\t : " << p4 << endl;
  cout << "5p4 at " << x << ", " << (5*p4).at(x) << "\t : " << 5*p4 << endl;

  cout << "total # terms used = " << Poly::TermsAllocated() << endl;
  return 0;
}

```

polymult.cpp

## O U T P U T

```

prompt> polymult
value of x 3
p1 at 3, 36      : 3x^2 + 4x + -3
p2 at 3, 47      : 4x^2 + 3x + 2
p3 at 3, 135     : 5x^3
p4 at 3, 4860    : 15x^5 + 20x^4 + -15x^3
5p4 at 3, 24300 : 75x^5 + 100x^4 + -75x^3
total # terms used = 25

```



More details of the implementation are provided in How to G, but we'll reproduce the private section of the class `Poly` and mention three important points of the implementation.

```

class Poly
{
  ...

  private:
    struct Pair          // this is the (a,b) in ax^b
    { double coeff;
      int expo;
      Pair() : coeff(0.0), expo(0) { }
      Pair(double c, int e) : coeff(c), expo(e) { }
    }

```

```
};
typedef CList<Pair> Polist;
typedef CListIterator<Pair> PolistIterator;
static bool ourInitialized;

Poly(Polist p); // poly from list of terms, helper
Polist myPoly; // the list of terms
};
```

- The struct `Pair` that represents a coefficient and an exponent is declared in the private section of `Poly`. It's used only in the implementation of polynomials. In general, it's possible to declare structs and classes inside other classes.
- A private constructor is declared for creating a polynomial from a `CList<Pair>` object, though the alias `Polist` is used for the `CList<Pair>` object. Client programs don't need to know that a list is being used, so the constructor should not be accessible to clients, but it's useful in implementing other member functions.
- (*This is advanced, it's fine to ignore it.*) The static variable `ourInitialized` will be false until the program is run. Then `Poly::ZERO` will be constructed, creating a zero polynomial and making the value of `ourInitialized` true. Then, every time a client calls the default `Poly` constructor, the object `Poly::ZERO` will be used. This means if 10,000 zero polynomials are created, only one cons call is actually made—see Program 10.20, *polycount.cpp*.

---

#### Program 10.20 `polycount.cpp`

---

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

int main()
{
    int k;
    for(k=0; k < 1000; k++)
    {
        Poly p;
    }
    cout << "# terms created = " << Poly::TermsAllocated() << endl;

    for(k=0; k < 1000; k++)
    {
        Poly p(3,4);
    }
    cout << "# terms created = " << Poly::TermsAllocated() << endl;

    return 0;
}
```

---

`polycount.cpp`

---

## O U T P U T

```
prompt> polycount
# terms created = 1
# terms created = 1001
```

## 10.6 The class `tmatrix`

A vector is a one-dimensional structure; an index accesses an element of the array by ranging from zero to one less than the number of elements stored. In some applications two-dimensional arrays are necessary. For example, the pixels on a computer screen are usually identified by a row and column position. Mileage tables that provide distances between cities on a road map also use two dimensions. Positions of pieces in a chess game are usually given by specifying a row and a column.

### 10.6.1 A Simple `tmatrix` Program

A two-dimensional array is sometimes called a **matrix** (the plural is matrices). The Cartesian  $(x, y)$  coordinate system uses two dimensions to specify a point in the plane. The system of latitude and longitude uses two dimensions to specify a location on the earth. Two-dimensional vectors, as we will implement them using the class `tmatrix`, use row and column indices to specify an entry of the matrix. The program *matdemo.cpp*, Program 10.21, defines a matrix, fills it with the equivalent of a multiplication table, and prints the matrix.

As with `tvector` variables, the type used to define the element stored in each matrix cell can be any built-in type or any programmer-defined type that has a default (parameterless) constructor. If a `tmatrix` is defined with the default constructor,

the matrix has zero rows and columns. In that case the member function `resize` should be used to set the number of rows and columns. The first parameter in a constructor or with `resize` is the number

#### Syntax: `tmatrix` definition

```
tmatrix<Type> mat;
tmatrix<Type> mat(row, col);
tmatrix<Type> mat(row, col, fillvalue);
```

of rows, the second parameter is the number of columns. Rows and columns are numbered starting with zero as with `tvector` variables. A third parameter can be used to provide initial values for all the cells of a matrix just as a second parameter provides values for `charCounts` in *letters.cpp*, Program 8.3. For example, the definition `tmatrix<double> chart(3, 5, 1.0);` defines a three-by-five matrix of 15 doubles, all initialized to 1.0.

Complete documentation for the `tmatrix` class is found in the header file *tmatrix.h*, see How to G for details.



---

Program 10.21 matdemo.cpp

```
#include <iostream>
#include <string>
#include <iomanip>      // for setw
using namespace std;
#include "tmatrix.h"

// demonstrate class tmatrix

template <class T>
void Print(const tmatrix<T>& mat)
{
    int j,k;
    int rows = mat.numrows(), cols = mat.numcols();
    for(j=0; j < rows; j++)
    {   for(k=0; k < cols; k++)
        {   cout << setw(4) << mat[j][k];
            }
        cout << endl;
    }
}

int main()
{
    int rows, cols,j,k;
    cout << "row col dimensions: ";
    cin >> rows >> cols;

    tmatrix<int> mat(rows,cols);
    for(j=0; j < rows; j++)           // fill matrix
    {   for(k=0; k < cols; k++)
        {   mat[j][k] = (j+1)*(k+1);
            }
        }
    Print(mat);
    return 0;
}
```

---

matdemo.cpp

```


O U T P U T



```

prompt> matdemo
row col dimensions: 3 5
  1  2  3  4  5
  2  4  6  8 10
  3  6  9 12 15

prompt> matdemo
row col dimensions: 7 4
  1  2  3  4
  2  4  6  8
  3  6  9 12
  4  8 12 16
  5 10 15 20
  6 12 18 24
  7 14 21 28

```


```

### 10.6.2 Case Study: Finding Blobs

In this section we'll use the `tmatrix` class to assist in looking for organisms on the kinds of slides used under microscopes. Of course we'll be using a simulated slide, but the algorithm and design techniques could be used if we were guiding a digital microscope, radiological CT scan or MRI, or a graphics-painting program. We'll use a character-based picture, but a similar class based on the graphics library documented in *How to H* is included in the exercises from this chapter. Slides will be represented by a character **bitmap**, using the class `CharBitMap` declared in *charbitmap.h*. Program 10.22 illustrates most of the `bitmap` class.



Program 10.22 `bitmapdemo.cpp`

```

#include <iostream>
using namespace std;
#include "charbitmap.h"
#include "prompt.h"
#include "randgen.h"

int main()
{
    int rows, cols;
    cout << "enter row col size ";
    cin >> rows >> cols;
    CharBitMap bmap(rows,cols);
    int pixelCount = PromptRange("# pixels on ",1,rows*cols);

```

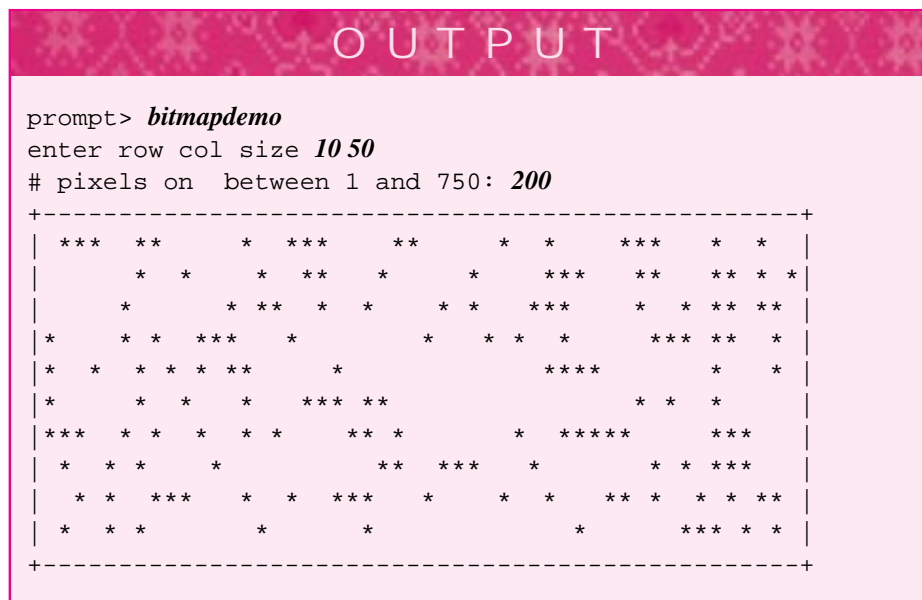
```

int k;
RandGen gen;
for(k=0; k < pixelCount; k++)
{
    bmap.SetPixel(gen.RandInt(0,rows-1),gen.RandInt(0,cols-1),CharBitMap::black);
}
bmap.Display(cout);
return 0;
}

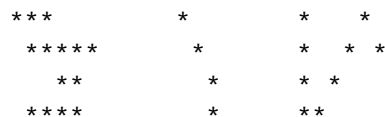
```

`bitmapdemo.cpp`

The pixels in a `CharBitMap` object can have values `CharBitMap::white` and `CharBitMap::black`, the identifiers `black` and `white` are class enum values. `CharBitMap` pixel coordinates are the same as `tmatrix` coordinates, ranging from zero to one less than the number of rows or columns.



We want to identify organisms on the slide. We'll define an organism to be a group of adjacent, black pixels where adjacent means connected horizontally or vertically. In the diagram below, there is a 14-pixel organism on the left; a 2-pixel organism in the middle, and a 5-pixel organism on the right.



The other asterisks in the diagram can be considered 1-pixel organisms or random noise. The middle organism is only a 2-pixel organism because diagonally adjacent pixels are not considered to be part of the same organism.

We want to design a class that counts the organisms in a `CharBitmap` object. The minimal size of an organism will be specified by the user; organisms smaller than this size will be considered noise. In the exercises we'll explore changing the definition of an organism to include diagonally adjacent cells, so we'll design the program to make extensions or modifications as simple as possible. In our initial design we'll need only two member functions other than a constructor in the class `Blobs`.<sup>7</sup>

1. A function `Blobs::Display` to display blobs.
2. A function `Blobs::FindBlobs` to which we pass a bitmap and a minimal size.

The `FindBlobs` function does all the work of finding the organisms/blobs, setting up for a subsequent call of `Display`, and returning the number of organisms found. We'll need some private, helper functions that do most of the work needed to implement `FindBlobs`. Helper functions are useful in general, but are particularly helpful when using recursion. Often, the method called by the client does not have the correct prototype for a recursive call or requires some initializing bookkeeping that's not appropriate in every recursive call. The method called by the client code can perform the initialization and then call the recursive helping function.

**Program Tip 10.6: Many member functions that use a recursive algorithm are most easily implemented by calling a recursive, private helper function.** The public method can set up bookkeeping and sometimes pass private data as an argument to the initial call of the recursive helper method. The bookkeeping should only be done once, and the private data are not available for clients to pass as arguments.

The recursive algorithm for finding an organism can be visualized by thinking of the recursive clones as scouts, sent out by an initial blob-counter to report on adjacent pixels and whether the adjacent pixels are part of the blob being counted.

#### Find a blob containing pixel (x,y), return size of blob

- If (x,y) isn't black, it's not part of a blob, stop and return zero
- Otherwise, (x,y) is part of a blob, send out blob-counting scouts/clones, accumulate results reported back
  - Four clones are sent, one in each direction
  - Each clone reports how many pixels it found that are part of the blob
  - Each clone covers its tracks so that its work won't be duplicated by other clones

Each call that finds a black pixel accumulates the results of the four clones and returns this result plus one for the found black pixel. If you believe that the clones work correctly (see Program Tip 10.2) then the correct result will be returned assuming each clone can cover its tracks.

It's essential that each clone marks where it has been so that clones sent out later don't

<sup>7</sup>We'll use "blobs" rather than "organism" because it's more fun to say "blobs."

count pixels that have already been counted. We'll implement this marking mechanism by using an int matrix. Initially we'll use values for black and white that won't be used as blob-marking values. When we mark blobs, we'll use a different int value for each blob that's found. The recursive, helper function `BlobFill` in *blobs.cpp*, Program 10.23 does all the work. Before looking at the implementation, we'll discuss the interface and how `BlobFill` is called. In the calls below, the instance variable `myBlobCount` is the value of how many blobs have been found so far. The int constants `PIXEL_ON` and `PIXEL_OFF` are used to initialize the Blob grid based on values from the `CharBitmap` parameter passed to `FindBlob`.

```
int Blobs::BlobFill(int row, int col,
                   int lookFor, int fillWith)
// spec: look for blob with pixel-value 'lookFor',
//       color in this blob using 'fillWith' value,
//       return size of blob
// post: returns size of blob at (row,col) and ``colors``
//       blob so that it won't be counted again
//
```

Keeping this specification in mind, `BlobFill` is called as follows. If `BlobFill` reports a blob whose size is more than the minimal being searched for, the number of blobs is incremented; otherwise, the blob doesn't count. If the blob doesn't count, it must be erased since the `BlobFill` call and its clones may have marked the too-small blob.

```
// j and k range over all row, column values
if (BlobFill(j,k, PIXEL_ON, myBlobCount+1)) > minSize)
{ myBlobCount++;
}
else // too small, erase
{ BlobFill(j,k, myBlobCount+1, PIXEL_OFF);
}
```

When a too-small blob is erased, the `lookFor` value is the same as the `fillWith` value from the call to `BlobFill` that just reported the too-small value.

---

#### Program 10.23 `blobs.cpp`

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "tmatrix.h"
#include "randgen.h"
#include "prompt.h"
#include "charbitmap.h"

// find blobs in a two-dimensional grid/bitmap
```

## 510

## Chapter 10 Recursion, Lists, and Matrices

```
class Blobs
{
public:

    Blobs();
    int FindBlobs(const CharBitMap& cbm, int minSize);
    void Display(ostream& out) const;

private:

    tmatrix<int> myGrid;
    int          myBlobCount;

    int BlobFill(int row, int col, int lookFor, int fillWith);
    void Initialize(const CharBitMap& cbm);

    static int PIXEL_OFF, PIXEL_ON;
};

int Blobs::PIXEL_OFF = 0;
int Blobs::PIXEL_ON = -1;

Blobs::Blobs()
    : myBlobCount(0)
{
    // grid is empty
}

void Blobs::Display(ostream& out) const
// post: display the blobs
{
    int j,k;
    int rows = myGrid.numrows();
    int cols = myGrid.numcols();
    for(j=0; j < rows; j++)
    {
        for(k=0; k < cols; k++)
        {
            char ch = '.';
            if (myGrid[j][k] > PIXEL_OFF)
            {
                ch = char ('0' + myGrid[j][k]);
            }
            out << ch;
        }
        out << endl;
    }
}

int Blobs::FindBlobs(const CharBitMap& cbm, int minSize)
// post: return # blobs whose size > minSize
{
    int j,k;
    myGrid.resize(cbm.Rows(), cbm.Cols());
    Initialize(cbm);
    int rows = myGrid.numrows();
    int cols = myGrid.numcols();
    for(j=0; j < rows; j++)
```

```

    {
        for(k=0; k < cols; k++)
        {
            if (myGrid[j][k] == PIXEL_ON)
            {
                if (BlobFill(j,k,PIXEL_ON,myBlobCount+1) > minSize)
                {
                    myBlobCount++;
                }
                else
                {
                    BlobFill(j,k,myBlobCount+1,PIXEL_OFF); // erase it
                }
            }
        }
    }
    return myBlobCount;
}

void Blobs::Initialize(const CharBitMap& cbm)
// post: myGrid initialized from cbm
{
    int j,k;
    int rows = myGrid.numrows();
    int cols = myGrid.numcols();
    for(j=0; j < rows; j++)
    {
        for(k=0; k < cols; k++)
        {
            if (cbm.GetPixel(j,k) == CharBitMap::black)
            {
                myGrid[j][k] = PIXEL_ON;
            }
            else
            {
                myGrid[j][k] = PIXEL_OFF;
            }
        }
    }
    myBlobCount = 0; // no blobs yet
}

int Blobs::BlobFill(int row, int col, int lookFor, int fillWith)
// spec: look for blob with pixel-value 'lookFor', color in this
//       blob using 'fillWith' value and return size of blob
// post: returns size of blob at (row,col) and "colors"
//       blob so that it won't be counted again
{
    static int rowoffset[] = { -1,+1,0,0 }; // north,south,east,west
    static int coloffset[] = { 0,0,+1,-1 };
    const int NBR_COUNT = 4;

    if (0 <= row && row < myGrid.numrows() &&
        0 <= col && col < myGrid.numcols())
    {
        if (myGrid[row][col] != lookFor) // not part of this blob
        {
            return 0;
        }

        // we found a blob element, color it and its neighbors
        myGrid[row][col] = fillWith;
        int k,r,c;
        int size = 1; // count this pixel, add connected counts
        for(k=0; k < NBR_COUNT; k++)

```

512

## Chapter 10 Recursion, Lists, and Matrices

```
{ r = row + rowoffset[k];
  c = col + coloffset[k];
  size += BlobFill(r,c,lookFor,fillWith);
}
return size;
}
return 0; // not on grid, not part of blob
}

int main()
{
  int rows, cols;
  cout << "enter row col size ";
  cin >> rows >> cols;
  CharBitMap bmap(rows,cols);
  int k;
  RandGen gen;
  Blobs blobber;
  int pixelCount = PromptRange("# pixels on: ",1,rows*cols);
  for(k=0; k < pixelCount; k++)
  { bmap.SetPixel(gen.RandInt(0,rows-1),gen.RandInt(0,cols-1),
                  CharBitMap::black);
  }
  bmap.Display(cout);
  int bsize;
  int blobCount;
  do
  { bsize = PromptRange("blob size (0 to exit) ",0,50);
    if (bsize != 0)
    { blobCount = blobber.FindBlobs(bmap,bsize);
      blobber.Display(cout);
      cout << endl << "# blobs = " << blobCount << endl;
    }
  } while (bsize > 0);
  return 0;
}
```

---

blobs.cpp



```

O U T P U T

blob size (0 to exit) between 0 and 50: 5
.....111.....22.....
.....11.1.....2.....
.....1.11...222.....
.....111.....
.....3.....111.....
.....333333.....1.....
.4......5......6.6.....
4444.....55.5.....6666.....
4.....5555.....6.....
.....55555555.....

# blobs = 6
blob size (0 to exit) between 0 and 50: 0
```

Pause to Reflect



**10.20** Write the function `RowSum` that returns the sum of the entries in one row of a matrix and the function `ColSum` that returns the sum of the entries in one column of a matrix.

```
int RowSum(const tmatrix<int>& m, int r)
// pre: 0 <= r < m.numrows()
// post: returns sum of numbers in row r

int ColSum(const tmatrix<int>& m, int c)
// pre: 0 <= c < m.numcols()
// post: returns sum of numbers in column c
```

**10.21** A *magic square* is a square matrix whose rows, columns, and main diagonals all sum to the same number. A  $3 \times 3$  magic square follows.

```
6 1 8
7 5 3
2 9 4
```

Write a boolean-valued function `IsMagic` that returns true if its square matrix parameter is magic and false otherwise. Call the functions `RowSum` and `ColSum` from the previous exercise.

**10.22** The code in *bitmapdemo.cpp*, Program 10.22 prompts the user for the number of pixels to turn on. In fact, fewer than this number will be on in almost every run of the program. Why, and how can you change the code to ensure that exactly `pixelCount` pixels are on?

- 10.23** It's possible to change three lines in `Blobs::BlobFill` so that diagonally adjacent pixels are considered part of an organism. What are the three lines and how should they be changed?
- 10.24** In an  $N \times N$  bitmap, what is the big-Oh complexity of `Blobs::FindBlobs`? Base your answer on the number of pixels that are examined or changed by the function.
- 10.25** Suppose you want to add the capability of reading in a bitmap from data stored in a file. Where's the right place to add this capability and why? Consider additions to `CharBitMap`, to `Blobs`, or writing another class or function.
- 10.26** The class `Blobs` counts blobs and prints them, but there's no way for clients to access the blobs either individually or collectively. Develop two ways to allow client programs to access individual blobs, (i.e., to get a collection of  $(x,y)$  pairs that make up a blob by calling appropriate `Blob` member functions). Consider using vectors or lists. The class `Point` from *point.h*, Program G.10 may help.
- 10.27** Discuss how to add features to the class `Blob` so that client code can find the size of the largest blob (an int value). Develop two methods, one that runs in  $O(N^2)$  time for an  $N \times N$  bitmap and one that runs in  $O(TBP)$  time where  $TBP$  is the total number of blob pixels.

## 10.7 Chapter Review

In this chapter we discussed recursion, a useful programming technique that can be misused. A recursive function does not "call itself," but calls a clone function, an identical copy of itself. Each recursively called function has its own parameters and its own local variables. We also covered variable scope and lifetime. A variable's scope is the part of the program in which the variable can be accessed. A variable's lifetime is how long the variable exists. The class `CList` is useful for representing sparse structures and lists of objects. The class `tmatrix` is like a two-dimensional vector.

Important topics covered include:

- Recursion is an alternative to iteration using loops. Recursive functions iterate by making recursive calls.
- Recursively called functions use memory; there is a limit on the number of recursive calls or recursively called functions. This limit depends on the amount of memory in the computer.
- Some problems are naturally solved with recursive functions and would be difficult to solve using loops.
- Some functions should not be coded recursively. One example is computing Fibonacci numbers.
- Recursive functions are often divided into two cases: a base case that does not involve a recursive call and a recursive case that makes a recursive call. The

recursive call should get closer to the base case so that there are a finite number of recursive calls.

- A variable's scope determines in which part of the program the variable can be accessed. Variables can be defined globally, accessible in all functions, or locally, accessible in the function in which the variable is defined.
- Variables can be defined within the braces, { and }; this means a variable's scope can be restricted to any compound statement, (e.g., accessible only within a loop).
- The scope resolution operator, ::, is used to access global variables when the variable identifier is shadowed by a local variable.
- Static variables maintain values throughout program execution, unlike nonstatic variables, whose lifetime is for the duration of the function in which the variable is defined.
- Static class variables belong to a class rather than to an object. Class variables are useful for keeping track of statistics involving all objects, (e.g., counting the number of objects created).
- The class `CList` represents immutable lists. A list is homogeneous, all elements are the same type. Lists are created using `cons` and processed, usually recursively, using `Head` and `Tail`.
- The class `CList` represents sparse structures efficiently. Representing polynomials provides one example.
- Two-dimensional vectors, or matrices, are useful for representing and manipulating data. We use a `tmatrix` class for two-dimensional arrays.
- Member functions that call for recursion are often most easily implemented using a private, helper function.

## 10.8 Exercises

- 10.1** An integer is printed with commas inserted in the proper positions similarly to the way in which digits in English are printed in *digits3.cpp*, Program 10.2. That is, to print the number 12345678 as 12,345,678, the 678 cannot be printed until *after* the preceding part of the number is printed. Write a recursive function `PrintWithCommas` that will print its `BigInt` parameter with commas inserted properly. The outline of the function is

```

if (number < 1000)
    print normally, no commas needed
else
    recursively print the number
        without the last three digits
    print a comma and the last three digits

```

You'll need to be careful with leading zeroes to ensure, for example, that the number 12,003 is printed properly. Write the function nonrecursively also by creating a string from the `BigInt` value and then printing the string appropriately with commas.

Modify the recursive function to return a string equivalent of the `BigInt`, but with commas properly inserted.

- 10.2** Modify Program 10.5, `subdir.cpp`, so that instead of printing the names of all files and subdirectories, the size of each subdirectory is calculated, returned, and printed. Use the member function `DirEntry::Size()` to calculate the size (usually expressed in bytes) of each file. Print the size of each subdirectory in a format that makes it easy to determine where large files might be found. Do *not* print the names of every file; just print the names of the subdirectories and the size of all the files within the subdirectory.
- 10.3** Pascal's triangle can be used to calculate the number of different ways of choosing  $k$  items from  $n$  different items. The first seven rows of Pascal's triangle are

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1

```

If we use  $C_k^n$  to represent the number of ways of choosing  $k$  items from  $n$ , then  $C_0^n = 1$  and  $C_n^n = 1$  as shown in the outside edges of the triangle. For values of  $k$  other than 0 and  $n$ , the following relationship holds.

$$C_k^n = C_{k-1}^{n-1} + C_k^{n-1} \quad (10.4)$$

Viewed in the triangle, each entry other than the outside 1's is equal to the two entries in the row above it diagonally up and to the left and right. For example,

$$C_2^5 = 10 = (C_1^4 + C_2^4) = (4 + 6)$$

Write a function `Choose`, with two parameters  $n$  and  $k$ , that returns the number of ways that  $k$  items can be chosen from  $n$ . Use this function to print the first 15 rows of Pascal's triangle.

- 10.4** Repeat the previous exercise, but try to develop a mechanism for storing the results of each recursive call using a static local variable so that no calculation is made more than once. This is tricky using `tvector` variables, but it is possible if you can develop a method for calculating a unique index for each pair of values used in the recursive calls. Alternatively, you can also use a `tmatrix`.
- 10.5** The value  $C_k^n$  can also be computed using the factorial function and this equation:

$$C_k^n = \frac{n!}{k! \cdot (n-k)!} \quad (10.5)$$

Write two versions of a function to compute the value of  $C_k^n$ , one based on the factorial function (where factorial is computed iteratively, *not* recursively) and one based on the recursive definition in the previous exercise. Time how long it takes to compute different values of  $C_k^n$ . Use `BigInt` values and compute  $C_k^n$  for large values of  $k$  and  $n$ .

**10.6** Implement multiplication of polynomials. For example,

$$(x^3 + 2x^2 + 3) \times (2x^2 + x - 2) = 2x^5 + 5x^4 + 2x^2 + 3x - 6$$

You should use the function `MONOMULT` from `polymult.cpp`, Program 10.19 as a helper function. Use this function to implement polynomial exponentiation so that you can calculate  $(x^2 + 3x + 4)^3$  efficiently.

**10.7** The function `POLY::at` for evaluating polynomials is implemented inefficiently. Evaluating  $x^{100} + 3x^{99} + 5$  is done by raising  $x^{100}$ , then adding the result of  $3x^{99}$ , then adding 5. It would be more efficiently calculated using  $(x^{99} + 3)x + 5$ . In general this method of evaluating a polynomial is called *Horner's Rule*:

$$\begin{aligned} a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a + 0 = \\ (\dots ((a_n x + a_{n-1})x + \dots + a_1)x + a_0 \end{aligned}$$

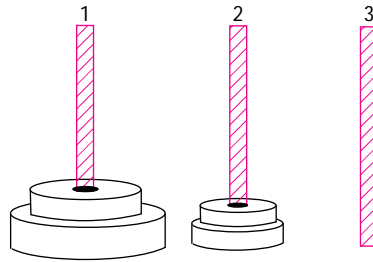
The coefficient of the largest exponent,  $a_n$ , is multiplied by  $x$ . Then  $a_{n-1}$  is added and the result multiplied by  $x$ ; then  $a_{n-2}$  is added and the result multiplied by  $x$  and so on. The simplest way to implement Horner's rule for evaluating polynomials requires a nonsparse representation, i.e., all coefficients, including zero coefficients, are needed. Write a function to produce a nonsparse representation of a polynomial. You won't be able to use a polynomial itself since the addition of polynomials with zero coefficients is ignored by `operator +=` for polynomials. Instead, you'll need to create a `CLIST<int>` object storing exponents and a `CLIST<double>` object storing coefficients; or you can create a list storing exponents and coefficients by declaring a struct like `PAIR` used in the implementation of `POLY`. Use this to evaluate a polynomial using Horner's rule and see if this method is more efficient in practice.

**10.8** The towers of Hanoi puzzle is traditionally studied in computer science courses. The roots of the puzzle are apparently found in the Far East, where a tower of golden disks is said to be used by monks. The puzzle consists of three pegs and a set of disks that fit over the pegs. Each disk is a different size. Initially the disks are on one peg, with the smallest disk on top, the largest on the bottom, and the disks arranged in increasing order. The object is to move the disks, one at a time, to another peg. No disk can be placed on a smaller disk.

If four disks are used and all disks are initially on the leftmost peg, numbered **1** in Figure 10.10, the following sequence of disk moves shows how to reach the configuration of disks shown. A move is indicated by the pegs involved since the topmost disk is always moved.

```
Move 1 to 3
Move 1 to 2
Move 3 to 2
```

To finish moving all the disks from the left peg to the middle peg, the top disk is moved from 1 to 3, then (recursively) the disks are moved from peg 2 to peg 3. The largest disk is then moved from peg 1 to peg 2. Finally (and recursively), the disks from peg 3 are moved to peg 2. Pegs are numbered 1, 2, and 3. To move 7 disks from peg 1 to peg 2, the function call `Hanoi(1, 2, 3, 7)` is used. To move these seven disks, two recursive calls are necessary: `Hanoi(1, 3, 2, 6)`, which moves six disks from peg 1 to peg 3, with peg 2 as the auxiliary peg; followed by a nonrecursive move of the largest disk from peg 1 to peg 2; followed by a recursive `Hanoi(3, 2, 1, 6)` to move the six disks from peg 3 to peg 2, with the now empty peg 1 as the auxiliary peg.



**Figure 10.10** The Towers of Hanoi.

Write the function `Hanoi`. The base case, and the single-disk case, should print the peg moves. For example, the output for a 4-disk tower follows.

```


O U T P U T



```

prompt> hanoi
number of disks:  between 0 and 30: 4
move from 1 to 3
move from 1 to 2
move from 3 to 2
move from 1 to 3
move from 2 to 1
move from 2 to 3
move from 1 to 3
move from 1 to 2
move from 3 to 2
move from 3 to 1
move from 2 to 1
move from 3 to 2
move from 1 to 3
move from 1 to 2
move from 3 to 2

```


```

Consider a function `Hanoi` using the following prototype.

```

void Hanoi(int from, int to, int aux, int numDisks)
// pre:  top numDisks-1 disks on 'from' peg
//        are all smaller than top disk on
//        'aux' peg
// post: top numDisks disks moved from
//        'from' peg to 'to' peg

```

- 10.9** Modify the *hanoi.cpp* program from the previous exercise to time how long it takes for different numbers of disks from 1 to 25. Comment out (put `//` before each statement) the statements that print disk moves so that the number of recursive calls is timed. Use a global variable that is incremented each time `HANOI` executes. Print the value of this variable for each number of disks so that the total number of disk moves is printed, along with the time it takes to move the disks. This can lead to a new measure of computer performance: **DIPS**, for “disks per second.”
- 10.10** A square matrix `a` is symmetric if `a[j][k] == a[k][j]` for all values of `j` and `k`; that is, the matrix is symmetric with respect to the main diagonal from `(0,0)` to `(n-1, n-1)` for an  $n \times n$  matrix. Write a `bool`-valued function that returns true if its matrix parameter is symmetric and false otherwise.
- 10.11** The  $N$ -queens problem has a long history in mathematics and computer science. The problem is posed in two ways:
- Can  $N$  queens be placed on an  $N \times N$  chess board so that no two queens attack each other?
  - How many ways can  $N$  queens be placed on an  $N \times N$  board so that no two queens attack each other?

In chess, queens attack each other if they’re on the same row, the same column, or the same diagonal. The sample output below shows one way to place eight queens so that no two attack each other.

```


O U T P U T



```

prompt> nqueens
size of board:  between 2 and 12: 8
X.....
.....X.
....X...
.....X
.X.....
...X...
.....X..
..X.....

```


```

Solving the  $N$ -queens problem uses an algorithmic technique called **backtracking** that’s related to the method used for generating permutations recursively in Section 10.3.3. The general idea of backtracking is to make a tentative attempt to solve a problem and then proceed recursively. If the tentative attempt fails, it is undone or *backtracked* and the next way of solving the problem is tried.

In the  $N$ -queens problem, we try to place a queen in each column. When the backtracking function is called, queens are successfully placed in columns 0 through `c0l`, and the function tries to place a queen in column `c0l+1`. There are  $N$  possible ways to place a queen, one for each row, and each one is tried in succession. If a queen

can be placed in the row, it is placed and a recursive call for the next column tries to complete the solution. If the recursive call fails, the just-placed queen is “un-placed”, or removed, and the next row tried for a placement. If all rows fail, the function fails. The *backtracking* comes when the function undoes an attempt that doesn’t yield a solution. A partial class declaration for solving the  $N$ -queens problem is given below. Complete the class and then modify it to return the total number of solutions rather than just printing the first solution found.

---

Program 10.24 `nqueenpartial.cpp`

---

```
class Queens
{
public:
    Queens(int size);
    bool Solve(); // return true if solvable
    void Print(ostream& out) const; // print the last board
private:
    // helper functions
    bool NoQueensAttackingAt(int r, int c) const;
    bool SolveAtCol(int col);

    tmatrix<bool> myBoard; // the board
};

bool Queens::NoQueensAttackingAt(int r, int c) const
// post: return true if row clear and diagonals crossing at
// (row,col) clear

bool Queens::Solve()
// post: return true if n queens can be placed
{
    return SolveAtCol(0);
}

bool Queens::SolveAtCol(int col)
// pre: queens placed at columns 0,1,...,col-1
// post: returns true if queen can be placed in column col
// if col == size of board, then n queens are placed
{
    int k;
    int rows = myBoard.numrows();
    if (col == rows) return true; // N queens placed
    for(k=0; k < rows; k++)
    {
        if (NoQueensAttackingAt(k,col)) // can place here?
        {
            myBoard[k][col] = true; // try it
            if (SolveAtCol(col+1)) // recurse
            {
                return true;
            }
            myBoard[k][col] = false; // backtrack
        }
    }
    return false;
}
```

```
int main()
{
    int size = PromptRange("size of board: ",2,12);
    Queens nq(size);
    if (nq.Solve())
    {
        nq.Print(cout);
    }
    else
    {
        cout << "no solution found" << endl;
    }
    return 0;
}
```

nqueenpartial.cpp

**10.12** An image can be represented as a 2-dimensional matrix of pixels, each of which can be off (white) or on (black). Color and gray-scale images can be represented using multivalued pixels; for example, numbers from 0 to 255 can represent different shades of gray. A **bitmap** is a two-dimensional matrix of 0s and 1s, where 0 corresponds to an off pixel and 1 corresponds to an on pixel. Instead of using the class `CharBitMap` for example, the following matrix of ints represents a bitmap that represents a  $9 \times 8$  picture of a < sign.

```
0 0 0 0 0 1 1 0
0 0 0 0 1 1 0 0
0 0 0 1 1 0 0 0
0 0 1 1 0 0 0 0
0 1 1 0 0 0 0 0
0 0 1 1 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 1 1 0 0
0 0 0 0 0 1 1 0
```

You can use the class `CharBitMap` used in Program 10.22, *bitmapdemo.cpp*, or you can create a new version of the class for use with the graphics package in How to H. Write a client program that provides the use with a menu of choices for manipulating an image.



- Read an image from a file
- Write an image to a file
- Invert the current image (change black to white and vice versa)
- Enlarge an image
- Enhance the image using median filtering (described below)

**Enlarging an Image.** A bitmap image can be enlarged by expanding it horizontally, vertically, or in both directions. Expanding an image in place (i.e., without using an auxiliary array) requires some planning. In Figure 10.11 an image is shown partially expanded by three vertically, and by two horizontally. By beginning the expansion in the lower right corner as shown, the image can be expanded in place—that is *without* the use of an auxiliary array or bitmap.

**Enhancing an Image.** Sometimes an image can be “noisy” because of the way in

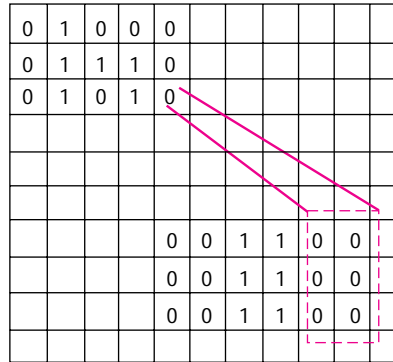


Figure 10.11 Enlarging a bitmap image.

which it is transmitted; for example, a TV picture may have static or “snow.” Image enhancement is a method that takes out noise by changing pixel values according to the values of the neighboring pixels. You should use a method of enhancement based on setting a pixel to the median value of those in its “neighborhood.” Figure 10.12 shows a 3-neighborhood and a 5-neighborhood of the middle pixel whose value is 28. Using **median filtering**, the 28 in the middle is replaced by the median of the values in its neighborhood. The nine values in the 3-neighborhood are (10 10 12 25 25 28 28 32 32). The median, or middle, value is 25—there are four values above 25 and four values below 25. The values in the 5-neighborhood are (10 10 10 10 10 10 12 12 12 18 18 18 25 25 25 25 25 25 32 32 32 32 32 32), and again the median value is 25, because there are 12 values above and 12 values below 25. The easiest way to find the median of a list of values is to sort them and take the middle element.

Pixels near the border of an image don’t have “complete” neighborhoods. These pixels are replaced by the median of the partial neighborhood that is completely on the grid of pixels. One way of thinking about this is to take, for example, a 3 × 3 grid and slide it over an image so that every pixel is centered in the grid. Each pixel is replaced by the median of the pixels of the image that are contained in the sliding grid. This

3-neighborhood

10	12	28
10	28	25
25	32	32

5-neighborhood

10	12	12	10	10
10	10	12	32	32
25	10	28	18	18
25	25	32	32	18
32	32	32	25	25

Figure 10.12 Neighborhoods for median filtering.



**Figure 10.13** Median filtering of a noisy image.

requires using an extra array to store the median values, which are then copied back to the original image when the median filtering has finished. This is necessary so that the pixels are replaced by median values from the original image, not from the partially reconstructed and filtered image.

Applying a  $3 \times 3$  median filter to the image on the left in Figure 10.13 results in the image on the right (these images look better on the screen than they do on paper).